

Object Oriented Programming C++

Chapter 6: Polymorphism

Exam Oriented Questions & Answers

(Short Answer, Broad Answer & Coding Questions)

Section A: Short Answer Questions

Q1. What is polymorphism?

Ans: Polymorphism is the ability of a function, operator, or interface to behave differently depending on the object or data type it is acting upon. It means “many forms” and allows the same interface to produce different results for different classes.

Q2. Name the two types of polymorphism in C++.

Ans: Compile-time (static) polymorphism, achieved through function/operator overloading, and run-time (dynamic) polymorphism, achieved through function overriding using virtual functions.

Q3. What is function overloading?

Ans: Function overloading is defining multiple functions with the same name in the same scope but with different parameter lists (different number, type, or order of parameters).

Q4. What is operator overloading?

Ans: Operator overloading is giving additional meaning to a built-in C++ operator (like +, -, ==) so that it works with user-defined data types such as classes.

Q5. What is function overriding?

Ans: Function overriding is redefining a base class's virtual function in a derived class using the exact same name, return type, and parameter list, so that it executes in place of the base version when called through a base pointer/reference.

Q6. Differentiate overloading and overriding in one line each.

Ans: Overloading is same name with different parameters within one class (compile-time); overriding is same signature redefined in a derived class (run-time).

Q7. What is a virtual function?

Ans: A virtual function is a member function declared with the keyword virtual in a base class, which can be redefined in a derived class, enabling run-time polymorphism via dynamic binding.

Q8. Can virtual functions be overloaded?

Ans: Yes. A class can declare multiple virtual functions with the same name but different parameter lists; each is independently virtual and can be separately overridden by derived classes.

Q9. What happens if a derived class overrides only one overloaded virtual function?

Ans: The other overloaded versions from the base class get hidden in the derived class due to name hiding, unless explicitly made visible using a using-declaration.

Q10. What is static binding?

Ans: Static binding (early binding) is when the function call is resolved by the compiler at compile time itself, as in normal function calls and overloaded functions.

Q11. What is dynamic binding?

Ans: Dynamic binding (late binding) is when the function call is resolved at run time, based on the actual object type, implemented through virtual functions using the vtable and vptr mechanism.

Q12. What are vtable and vptr?

Ans: The vtable (virtual table) is a table of function pointers maintained per class that has virtual functions; the vptr (virtual pointer) is a hidden pointer stored in each object that points to its class's vtable, enabling dynamic dispatch.

Q13. What is a pure virtual function?

Ans: A pure virtual function is a virtual function with no body in the base class, declared by assigning it 0, e.g. virtual void draw() = 0;. It forces derived classes to provide their own implementation.

Q14. What is an abstract class?

Ans: An abstract class is a class that contains at least one pure virtual function. It cannot be instantiated directly, though pointers and references to it can be created.

Q15. Why can a constructor not be virtual?

Ans: Because the vptr of an object is only set up once the object begins construction; since the constructor's job is to build the object, dynamic dispatch cannot apply to it.

Q16. Why should a destructor be virtual?

Ans: So that deleting a derived class object through a base class pointer correctly calls the derived class destructor followed by the base class destructor, avoiding resource leaks and undefined behaviour.

Q17. What is the Virtual Constructor Idiom?

Ans: A design technique where a virtual function (commonly named clone() or create()) is defined in the base class and overridden in each derived class to return a new object of its own derived type, simulating the effect of a “virtual constructor.”

Section B: Broad / Long Answer Questions

Q1. Explain the concept and types of polymorphism in C++ with suitable examples.

Ans: Polymorphism is a core OOP feature meaning “many forms” — it allows the same function name or operator to behave differently depending on context. C++ supports two types: (i) Compile-time polymorphism, resolved during compilation, achieved via function overloading (e.g., `add(int,int)` and `add(double,double)` in the same class) and operator overloading (e.g., overloading `+` for a `Complex` class); and (ii) Run-time polymorphism, resolved during execution, achieved via function overriding with virtual functions accessed through base class pointers/references (e.g., a `Shape` base class with an overridden `draw()` in `Circle` and `Square`). Compile-time polymorphism is faster since there is no run-time lookup, while run-time polymorphism offers greater flexibility since the correct function is chosen based on the actual object type at execution time.

Q2. Differentiate between function overloading and function overriding with a comparison table.

- **Scope:** Overloading happens in the same class/scope; overriding happens between a base and a derived class.
- **Signature:** Overloading requires a different parameter list; overriding requires an identical signature.
- **Binding time:** Overloading is resolved at compile time; overriding (with virtual functions) is resolved at run time.
- **Inheritance requirement:** Overloading does not need inheritance; overriding necessarily involves inheritance.
- **Polymorphism type:** Overloading gives static/compile-time polymorphism; overriding gives dynamic/run-time polymorphism.

Q3. What are virtual functions? Explain with a program how they enable run-time polymorphism.

Ans: A virtual function is a base class member function declared with the `virtual` keyword that a derived class is expected to override. When called through a base class pointer or reference, the version actually executed is determined by the real (derived) type of the object at run time, not by the declared type of the pointer. This is implemented internally through a vtable (a per-class table of function addresses) and a `vp`tr stored in each object, which points to its class's vtable.

```
class Animal {
public:
    virtual void sound() { cout << "Some generic sound"; }
};
class Dog : public Animal {
public:
    void sound() override { cout << "Bark"; }
};
class Cat : public Animal {
public:
    void sound() override { cout << "Meow"; }
};
```

```
int main() {
    Animal *a;
    Dog d; Cat c;
    a = &d; a->sound();    // Bark
    a = &c; a->sound();    // Meow
}
```

Output:

Bark
Meow

Q4. Explain static binding and dynamic binding. How does C++ implement dynamic binding internally?

Ans: Static (early) binding resolves a function call at compile time; it applies to normal function calls, overloaded functions and operator overloading, and is fast since no run-time decision is needed. Dynamic (late) binding resolves the call at run time and is used for virtual functions accessed via base class pointers/references. Internally, every class that has at least one virtual function gets a vtable containing pointers to its virtual functions' actual implementations. Every object of such a class carries a hidden vptr pointing to its class's vtable. When a virtual function is called through a pointer, the compiler generates code to look up the correct function address in the vtable via the vptr at run time, ensuring the derived class's overridden version executes.

Q5. What is a pure virtual function and an abstract class? Write a program using an abstract class to calculate area of different shapes.

Ans: A pure virtual function has no definition in the base class and is declared as virtual returnType functionName() = 0;. A class containing at least one such function is an abstract class; it cannot be instantiated, but can be used to declare pointers/references, enabling a common interface for all derived shapes.

```
class Shape {
public:
    virtual double area() = 0;    // pure virtual
    virtual ~Shape() {}
};
class Rectangle : public Shape {
    double l, w;
public:
    Rectangle(double l, double w) : l(l), w(w) {}
    double area() override { return l * w; }
};
class Circle : public Shape {
    double r;
public:
    Circle(double r) : r(r) {}
    double area() override { return 3.14159 * r * r; }
};
int main() {
    Shape *s;
    Rectangle rect(4, 5); Circle cir(3);
    s = &rect; cout << s->area() << endl;    // 20
    s = &cir;  cout << s->area() << endl;    // 28.27
}
```

Output:

20
28.2743

Q6. Why can constructors not be virtual, and why should destructors be virtual? Illustrate with a program what goes wrong without a virtual destructor.

Ans: A constructor cannot be virtual because the vptr that enables virtual dispatch is only established once the object's construction begins — there is no complete object or valid vptr available before or during construction to base a dynamic call on. Destructors, on the other hand, should be declared virtual whenever a class is meant to be used polymorphically (i.e., deleted through a base class pointer), because without a virtual destructor, deleting a derived object via a base pointer only invokes the base class destructor, skipping the derived class destructor and any cleanup (like freeing dynamically allocated memory) it performs — this causes resource/memory leaks and undefined behaviour.

```
class Base {
public:
    ~Base() { cout << "Base destructor\n"; }    // NOT virtual
};
class Derived : public Base {
    int *data;
public:
    Derived() { data = new int[10]; }
    ~Derived() { delete[] data; cout << "Derived destructor\n"; }
};
int main() {
    Base *b = new Derived();
    delete b;    // Only Base destructor runs -> memory leak of 'data'
}
```

Output:

Base destructor
(Derived destructor never runs – memory leak)

Making the base destructor virtual (virtual ~Base(){...}) fixes this: it ensures Derived's destructor executes first, followed by Base's, so data is correctly freed.

Section C: Coding Questions & Answers

Q1. Write a C++ program to overload the '+' operator to add two Complex numbers.

```
#include <iostream>
using namespace std;
class Complex {
    float real, imag;
public:
    Complex(float r = 0, float i = 0) : real(r), imag(i) {}
    Complex operator+(const Complex &c) {
        return Complex(real + c.real, imag + c.imag);
    }
    void display() { cout << real << " + " << imag << "i"; }
};
int main() {
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;
    c3.display();
}
```

Output:

4 + 6i

Q2. Write a program demonstrating function overloading to find the maximum of two integers and two floats.

```
#include <iostream>
using namespace std;
int max(int a, int b) { return (a > b) ? a : b; }
float max(float a, float b) { return (a > b) ? a : b; }
int main() {
    cout << max(10, 20) << endl;
    cout << max(3.5f, 2.1f) << endl;
}
```

Output:

20

3.5

Q3. Write a program to demonstrate run-time polymorphism using virtual functions for an Employee payroll hierarchy.

```
#include <iostream>
using namespace std;
class Employee {
public:
    virtual float calculateSalary() { return 20000; }
};
class Manager : public Employee {
public:
    float calculateSalary() override { return 50000; }
};
int main() {
    Employee *e;
    Manager m;
    e = &m;
}
```

```
    cout << "Salary: " << e->calculateSalary();  
}
```

Output:

Salary: 50000

Q4. Write a program showing a pure virtual function and an abstract class Vehicle with derived classes Car and Bike.

```
#include <iostream>  
using namespace std;  
class Vehicle {  
public:  
    virtual void start() = 0;  
};  
class Car : public Vehicle {  
public:  
    void start() override { cout << "Car starts with a key/button\n"; }  
};  
class Bike : public Vehicle {  
public:  
    void start() override { cout << "Bike starts with a kick/self-start\n"; }  
};  
int main() {  
    Vehicle *v;  
    Car c; Bike b;  
    v = &c; v->start();  
    v = &b; v->start();  
}
```

Output:

Car starts with a key/button

Bike starts with a kick/self-start

Q5. Write a program demonstrating the Virtual Constructor Idiom using a clone() function.

```
#include <iostream>  
using namespace std;  
class Shape {  
public:  
    virtual Shape* clone() = 0;  
    virtual void draw() = 0;  
    virtual ~Shape() {}  
};  
class Square : public Shape {  
public:  
    Shape* clone() override { return new Square(*this); }  
    void draw() override { cout << "Drawing a Square"; }  
};  
int main() {  
    Shape *s1 = new Square();  
    Shape *s2 = s1->clone();    // creates a new Square without knowing type  
    directly  
    s2->draw();  
    delete s1; delete s2;  
}
```

Output:

Drawing a Square

Q6. Write a program that demonstrates the problem of a non-virtual destructor and then fixes it.

```
#include <iostream>
using namespace std;
class Base {
public:
    virtual ~Base() { cout << "Base destructor\n"; }    // fixed: virtual
};
class Derived : public Base {
public:
    ~Derived() { cout << "Derived destructor\n"; }
};
int main() {
    Base *b = new Derived();
    delete b;    // Now correctly calls both destructors
}
```

Output:

Derived destructor
Base destructor