Object Oriented Programming C++

Chapter 6: Polymorphism

Engineering Notes

Topics: 6.1 Concepts & Types of Polymorphism, Overloading & Overriding, Virtual Functions, Static & Dynamic Binding | 6.2 Pure Virtual Functions, Virtual Constructors and Destructors

6.1 Concepts of Polymorphism, Types of Polymorphism, Overloading & Overriding, Overloading Virtual Functions, Static & Dynamic Binding

Concept of Polymorphism

Polymorphism is one of the four fundamental pillars of Object Oriented Programming, alongside encapsulation, inheritance and abstraction. The word itself comes from Greek, meaning "many forms." In the context of C++, polymorphism refers to the ability of a single interface, function name, or operator to behave differently depending on the context in which it is used, or on the type of object that invokes it.

In practical terms, polymorphism allows a program to decide, either at compile time or at run time, which version of a function to execute. This gives the programmer the ability to write generic code that works with objects of different classes through a common interface, while each class provides its own specific implementation of the behaviour. This greatly improves code reusability, extensibility and maintainability, since new derived classes can be added without modifying existing calling code.

Types of Polymorphism

C++ supports polymorphism in two broad categories, based on when the binding of a function call to its definition is resolved:

- **Compile-Time Polymorphism (Static Polymorphism):** The function or operator to be executed is determined by the compiler at compile time itself. It is achieved through function overloading and operator overloading. It is also called early binding or static binding.
- **Run-Time Polymorphism (Dynamic Polymorphism):** The function to be executed is determined during program execution, based on the actual type of the object being referred to. It is achieved through function overriding using virtual functions, along with base class pointers or references. It is also called late binding or dynamic binding.

Function Overloading

Function overloading allows two or more functions to share the same name within the same scope, provided their parameter lists differ — either in the number of parameters, the type of parameters, or the order of parameter types. The return type alone is not sufficient to distinguish overloaded functions.

```
class Calculator {
public:
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
    int add(int a, int b, int c) { return a + b + c; }
};
```

Here, the compiler selects the appropriate add() function to call based on the number and types of arguments passed — this decision is made entirely at compile time.

Operator Overloading

C++ also allows most built-in operators (+, -, *, ==, <<, and so on) to be redefined for user-defined types such as classes. This is called operator overloading, and it is another form of compile-time polymorphism. It allows objects of a class to be used with natural, intuitive syntax.

```
class Complex {
    int real, imag;
public:
    Complex operator+(const Complex &c) {
        Complex temp;
        temp.real = real + c.real;
        temp.imag = imag + c.imag;
        return temp;
    }
};
```

Function Overriding

Function overriding occurs when a derived class provides its own implementation of a function that is already defined in its base class, using the exact same name, return type and parameter list. Unlike overloading, overriding is a run-time concept and requires an inheritance relationship between classes. Overriding forms the foundation of dynamic polymorphism in C++ and is typically implemented using virtual functions.

```
class Shape {
public:
    virtual void draw() { cout << "Drawing shape"; }
};
class Circle : public Shape {
public:
    void draw() override { cout << "Drawing circle"; }
};
```

Overloading vs Overriding — Key Differences

- Overloading happens within the same class (or scope); overriding happens between a base class and a derived class.
- Overloading requires different parameter lists; overriding requires an identical function signature.
- Overloading is resolved at compile time; overriding is resolved at run time (when using virtual functions).
- Overloading is a form of static polymorphism; overriding is a form of dynamic polymorphism.

Virtual Functions and Overloading Virtual Functions

A virtual function is a member function declared in a base class using the keyword virtual, which the programmer expects to be redefined in derived classes. When a virtual function is called through a base class pointer or reference, C++ determines at run time which version of the function to invoke, based on the actual type of the object pointed to — not the type of the pointer itself. This mechanism is the basis of run-time polymorphism.

Virtual functions themselves can also be overloaded within the same class. That is, a class may declare multiple virtual functions with the same name but different parameter lists. Each overloaded version is independently virtual, and derived classes may override any one or more of these overloaded versions. However, if a derived class overrides only one overloaded version, the remaining overloaded versions from the base class become hidden in the derived class scope due to C++'s name-hiding rules, unless explicitly brought into scope using a using-declaration.

```
class Base {
public:
    virtual void show(int x) { cout << "Base int: " << x; }
    virtual void show(double x) { cout << "Base double: " << x; }
};
class Derived : public Base {
public:
    using Base::show;           // brings both overloads into scope
    void show(int x) override { cout << "Derived int: " << x; }
};
```

Static Binding and Dynamic Binding

Binding refers to the process of associating a function call with the actual function definition (the code) that will be executed. C++ supports two kinds of binding:

- **Static Binding (Early Binding):** The compiler resolves the function call at compile time itself. This applies to normal (non-virtual) function calls, overloaded functions, and operator overloading. It is faster because there is no run-time overhead, but it is less flexible.
- **Dynamic Binding (Late Binding):** The function call is resolved at run time, based on the type of object actually being pointed to or referred to. This is implemented using virtual functions accessed through base class pointers or references. C++ implements this internally using a Virtual Table (vtable) and a hidden vptr (virtual pointer) stored in every object of a polymorphic class.

In dynamic binding, when a base class pointer holding the address of a derived class object calls a virtual function, the vptr of the object is used to look up the correct function address in the class's vtable, and that specific overridden version is executed — this is exactly what enables run-time polymorphism to work correctly.

```
Base *bptr;
Derived d;
bptr = &d;
bptr->show(5);   // Dynamic binding: Derived::show(int) is called
```

6.2 Pure Virtual Functions, Virtual Constructors and Destructors

Pure Virtual Functions

A pure virtual function is a virtual function that has no implementation (definition) in the base class itself. It is declared by assigning it the value 0 in its declaration. A pure virtual function forces every derived class to provide its own implementation, or else the derived class itself becomes abstract.

```
class Shape {
public:
    virtual double area() = 0;   // pure virtual function
};
```

A class that contains at least one pure virtual function is called an Abstract Class. Abstract classes cannot be instantiated directly — no object of an abstract class can be created — but pointers and references to an abstract class can be created, which is precisely what allows run-time polymorphism to be used through them. Abstract classes are commonly used to define a common interface that all derived classes must follow.

Key Points about Pure Virtual Functions and Abstract Classes

- An abstract class can still contain regular (non-pure) member functions, data members, and constructors.
- If a derived class does not override all the pure virtual functions of its base class, it also becomes abstract.
- Abstract classes are typically used to design a base class purely as an interface, defining what derived classes must do without specifying how.

```
class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() override { return 3.14159 * radius * radius; }
};
```

Virtual Constructors

In C++, a constructor can never be declared virtual. This is because the virtual mechanism relies on the vptr (virtual pointer), which is set up inside an object only after the object has been constructed. Since a constructor's job is to create and initialise the object in the first place, there is no vptr available yet at the point the constructor executes, so dynamic dispatch on a constructor is not possible.

Even though true “virtual constructors” do not exist in C++, the same effect — creating an object of the correct derived type without knowing its exact type in advance — is commonly achieved using the Virtual Constructor Idiom, also known as the Factory Method or Clone pattern. In this idiom, a virtual function (often named clone() or create()) is defined in the base class and overridden in each derived class to return a new object of that derived class's own type.

```
class Shape {
public:
    virtual Shape* clone() = 0;   // acts like a "virtual constructor"
    virtual ~Shape() {}
};
class Circle : public Shape {
public:
    Shape* clone() override { return new Circle(*this); }
};
```

Virtual Destructors

Unlike constructors, destructors in C++ can, and very often should, be declared virtual. This is essential whenever a base class pointer is used to delete an object of a derived class. If the base class destructor is not virtual, deleting a derived class object through a base class pointer results in undefined behaviour: only the base class portion of the object is destroyed, and the derived class's destructor is never called, which leads to resource leaks.

```
class Base {
public:
    virtual ~Base() { cout << "Base destructor"; }
};
class Derived : public Base {
public:
    ~Derived() { cout << "Derived destructor"; }
};
int main() {
    Base *bptr = new Derived();
    delete bptr;   // Correctly calls Derived::~Derived() then Base::~Base()
}
```

The general rule of thumb followed by experienced C++ programmers is: if a class has even a single virtual function, or if it is intended to be used as a base class in an inheritance hierarchy, its destructor should always be declared virtual. This ensures that the complete destructor chain of a derived object executes correctly when destroyed through a base class pointer or reference.

Summary Table — Chapter 6 at a Glance

- Compile-time polymorphism: function overloading, operator overloading — resolved by the compiler.
- Run-time polymorphism: function overriding, virtual functions — resolved at execution using vptr/vtable.
- Static binding: fast, resolved early; Dynamic binding: flexible, resolved late through virtual functions.
- Pure virtual function (= 0) makes a class abstract; abstract classes cannot be instantiated.
- Constructors cannot be virtual; the Virtual Constructor Idiom (clone/create) simulates the effect.
- Destructors should be virtual whenever polymorphic deletion through a base pointer is possible.

Multiple Choice Questions (MCQs)

1. Which of the following best defines polymorphism in C++?

A) Hiding data members from outside access

B) The ability of a function/operator to take many forms depending on context

C) Deriving one class from another

D) Wrapping data and functions into a single unit

2. Compile-time polymorphism in C++ is also known as:

A) Late binding

B) Dynamic binding

C) Early binding

D) Run-time binding

3. Which mechanism is used to achieve compile-time polymorphism?

A) Virtual functions

B) Function overloading and operator overloading

C) Pure virtual functions

D) Abstract classes

4. Function overloading is distinguished by:

A) Return type only

B) Function name only

C) Number, type, or order of parameters

D) Access specifier

5. Which of these is required for function overriding to occur?

A) Same function name, same signature, in base and derived class

B) Different parameter lists in the same class

C) Only friend functions

D) Only static functions

6. Run-time polymorphism in C++ is implemented using:

A) Operator overloading

B) Virtual functions

C) Templates

D) Macros

7. A virtual function is declared using which keyword?

A) virtual

B) override

C) static

D) friend

8. The internal mechanism C++ compilers use to implement dynamic binding is called:

A) Symbol table

B) Virtual Table (vtable) and vptr

C) Activation record

D) Heap allocation table

9. If a derived class overrides only one overloaded virtual function from the base class, the remaining overloaded versions become:

A) Automatically inherited and callable

B) Hidden, unless brought into scope with a using-declaration

C) Deleted permanently

D) Converted to pure virtual functions

10. A pure virtual function is declared by:

A) Assigning it the value 0 in the declaration

B) Using the keyword pure

C) Marking it as static

D) Leaving out the return type

11. A class containing at least one pure virtual function is called:

A) A final class

B) A friend class

C) An abstract class

D) A static class

12. Which of the following is true about abstract classes in C++?

A) They can be instantiated directly

B) Pointers/references to them can be created but objects cannot

C) They cannot contain any data members

D) They must have a virtual destructor only

13. Why can a constructor never be declared virtual in C++?

A) Constructors do not return values

B) The vptr is not set up until after object construction begins

C) Constructors cannot take arguments

D) Constructors are always private

14. The commonly used technique to simulate a “virtual constructor” in C++ is called:

A) The Singleton pattern

B) The Virtual Constructor Idiom (clone/factory method)

C) The Observer pattern

D) Static polymorphism

15. A base class destructor should be declared virtual when:

A) The class has no derived classes

B) Objects of derived classes may be deleted through a base class pointer

C) The class has only static members

D) The class is never inherited

Answer Key

1-B 2-C 3-B 4-C 5-A 6-B 7-A 8-B

9-B 10-A 11-C 12-B 13-B 14-B 15-B

Broad / Long Answer Questions

1. Define polymorphism. Explain its significance in Object Oriented Programming with suitable examples.
2. Differentiate between compile-time (static) polymorphism and run-time (dynamic) polymorphism, giving one example of each.
3. What is function overloading? Write a C++ program to overload a function to compute the area of a square, rectangle, and circle.
4. Explain operator overloading with an example of overloading the '+' operator for a Complex number class.
5. Distinguish between function overloading and function overriding. Support your answer with a comparison table.
6. What are virtual functions? Explain, with a program, how virtual functions achieve run-time polymorphism using base class pointers.
7. Explain how a virtual function can itself be overloaded in a base class. What issue arises when a derived class overrides only one overloaded version, and how is it resolved?
8. Describe the concepts of static binding and dynamic binding. Explain the role of vptr and vtable in implementing dynamic binding.
9. What is a pure virtual function? What is an abstract class? Explain with an example why abstract classes cannot be instantiated.
10. Why can constructors not be declared virtual in C++, while destructors can be? Explain the Virtual Constructor Idiom and the importance of virtual destructors with a program illustrating memory/behaviour issues if the destructor is not virtual.