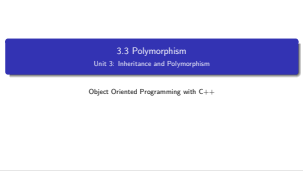
3.3 Polymorphism

Unit 3: Inheritance and Polymorphism

Object Oriented Programming with C++



Lecture 26: 3.3 Polymorphism

- Unit 3: Inheritance and Polymorphism
- Topics: Compile-time Polymorphism, Run-time Polymorphism, Function & Operator Overloading, Virtual Functions
- Course: Object Oriented Programming with C++ (DI05011021)

Lecture 26: 3.3 Polymorphism
- Unit 3: Inheritance and Polymorphism
- Topics: Compile-time Polymorphism, Run-time Polymorphism, Function & Operator Overloading, Virtual Functions
- Course: Object Oriented Programming with C++ (DI05011021)

Welcome class to Lecture 26. Today we are diving into one of the most powerful pillars of Object-Oriented Programming: Polymorphism. We will build upon our knowledge of classes and inheritance to see how C++ allows us to write flexible, scalable, and reusable code by treating different objects through a unified interface.

What is Polymorphism?

- The word 'Polymorphism' is derived from two Greek words: 'poly' (meaning many) and 'morphs' (meaning forms).
- In OOP, polymorphism refers to the ability of a message or function to be displayed or executed in more than one form.
- Real-world analogy: A person behaves differently depending on the context. A woman might be a mother at home, an employee at work, and a customer at a store.
- In C++, it allows a single function name, operator, or object reference to behave differently based on the data types or object classes they are interacting with.

What is Polymorphism?
- The word 'Polymorphism' is derived from two Greek words: 'poly' (meaning many) and 'morphs' (meaning forms).
- In OOP, polymorphism refers to the ability of a message or function to be displayed or executed in more than one form.
- Real-world analogy: A person behaves differently depending on the context. A woman might be a mother at home, an employee at work, and a customer at a store.
- In C++, it allows a single function name, operator, or object reference to behave differently based on the data types or object classes they are interacting with.

Let's start with the definition. Polymorphism literally means 'many forms'. Think about how you behave. The way you interact with your friends is different from how you interact with your professors, yet you are the same person responding to different 'inputs' or contexts. In C++, polymorphism allows our code to be context-aware. A single function call can trigger entirely different underlying behaviors depending on the exact object it is invoked upon. This reduces cognitive load because we can use one name for a general action.

Types of Polymorphism in C++

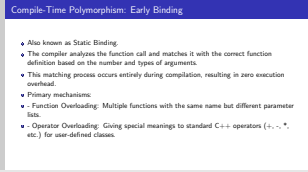
- C++ supports two primary categories of polymorphism:
- 1. Compile-Time Polymorphism (Early Binding / Static Binding)
- - The compiler determines which function to call at compile time.
- - Achieved through: Function Overloading and Operator Overloading.
- 2. Run-Time Polymorphism (Late Binding / Dynamic Binding)
- - The function call is resolved during the execution of the program.
- - Achieved through: Inheritance and Virtual Functions.

Types of Polymorphism in C++
- C++ supports two primary categories of polymorphism:
- 1. Compile-Time Polymorphism (Early Binding / Static Binding)
- - The compiler determines which function to call at compile time.
- - Achieved through: Function Overloading and Operator Overloading.
- 2. Run-Time Polymorphism (Late Binding / Dynamic Binding)
- - The function call is resolved during the execution of the program.
- - Achieved through: Inheritance and Virtual Functions.

C++ handles polymorphism in two distinct phases: during compilation and during execution. Compile-time polymorphism is fast because the compiler figures out exactly which memory address to jump to before the program even runs. This is also known as early binding. Run-time polymorphism, or late binding, happens while the program is actually running. The compiler doesn't know the exact object type beforehand, so the decision of which function to execute is delayed until the last possible moment.

Compile-Time Polymorphism: Early Binding

- Also known as Static Binding.
- The compiler analyzes the function call and matches it with the correct function definition based on the number and types of arguments.
- This matching process occurs entirely during compilation, resulting in zero execution overhead.
- Primary mechanisms:
- - Function Overloading: Multiple functions with the same name but different parameter lists.
- - Operator Overloading: Giving special meanings to standard C++ operators (+, -, *, etc.) for user-defined classes.



Let's look closer at compile-time polymorphism. The term 'early binding' means the linkage between the function call and the actual code block is established early—by the compiler. The compiler looks at the signature of the function you are trying to call. The signature includes the name of the function and the exact types and number of its arguments. If it finds a perfect match, it statically binds them. This means your compiled program is highly optimized.

Function Overloading: The Rules

- You can define multiple functions with the exact same name within the same scope.
- Rule 1: The parameter list (signature) must differ. This means a different number of parameters OR different data types of parameters.
- Rule 2: The return type alone is NOT sufficient to overload a function. If two functions have the same name and parameters but different return types, the compiler will throw an error.
- Compiler uses name mangling (decorating the function name with parameter types) internally to distinguish them.

Function Overloading: The Rules
- You can define multiple functions with the exact same name within the same scope.
- Rule 1: The parameter list (signature) must differ. This means a different number of parameters OR different data types of parameters.
- Rule 2: The return type alone is NOT sufficient to overload a function. If two functions have the same name and parameters but different return types, the compiler will throw an error.
- Compiler uses name mangling (decorating the function name with parameter types) internally to distinguish them.

Function overloading is extremely common. For instance, you might want an 'add' function. Sometimes you add two integers, sometimes two floats, sometimes three integers. Instead of naming them addInts, addFloats, addThreeInts, you just name them all 'add'. The compiler is smart enough to look at the arguments you pass in. However, remember the golden rule: the parameter list must be unique. The compiler ignores the return type when resolving overloads because you might call a function and ignore its return value, leaving the compiler guessing which one you meant.

Code Example: Function Overloading

```
class MathOps {
public:
// Overload 1: Two integers
int add(int a, int b) { return a + b; }

// Overload 2: Three integers
int add(int a, int b, int c) { return a + b + c; }

// Overload 3: Two doubles
double add(double a, double b) { return a + b; }
};
// Usage: MathOps m; m.add(5, 10); m.add(2.5, 3.1);
```


Code Example: Function Overloading
class MathOps {
public:
// Overload 1: Two integers
int add(int a, int b) { return a + b; }
// Overload 2: Three integers
int add(int a, int b, int c) { return a + b + c; }
// Overload 3: Two doubles
double add(double a, double b) { return a + b; }
};
// Usage: MathOps m; m.add(5, 10); m.add(2.5, 3.1);

Here is a concrete example. We have a class MathOps with three methods, all named 'add'. The first takes two ints, the second takes three ints, and the third takes two doubles. When we write m.add(5, 10), the compiler sees two integer literals, matches them to the first overload, and statically binds that specific block of code. If we pass 2.5 and 3.1, it binds to the double version. The interface remains clean and intuitive.

Operator Overloading: Basics

- C++ allows you to redefine how standard operators work when applied to your own custom classes (objects).
- For example, the '+' operator normally adds two numbers. With operator overloading, you can make '+' concatenate two custom 'String' objects or add two 'Complex' number objects.
- Syntax involves the 'operator' keyword followed by the symbol.
- Restrictions: You cannot change the precedence or associativity of an operator, nor can you create entirely new operator symbols.

Operator Overloading: Basics
- C++ allows you to redefine how standard operators work when applied to your own custom classes (objects).
- For example, the '+' operator normally adds two numbers. With operator overloading, you can make '+' concatenate two custom 'String' objects or add two 'Complex' number objects.
- Syntax involves the 'operator' keyword followed by the symbol.
- Restrictions: You cannot change the precedence or associativity of an operator, nor can you create entirely new operator symbols.

Moving on to Operator Overloading. This is what makes C++ feel so natural when working with user-defined types. If you create a Matrix class, it makes sense to write Matrix C = A + B instead of Matrix C = A.add(B). Operator overloading lets you do exactly this. You define a special function using the keyword 'operator' followed by the symbol you want to overload. It's essentially just 'syntactic sugar' over function overloading.

Code Example: Operator Overloading

```
class Complex {
private:
float real, imag;
public:
Complex(float r = 0, float i = 0) : real(r), imag(i) {}
// Overloading the '+' operator
Complex operator+(const Complex& obj) {
Complex temp;
temp.real = real + obj.real;
temp.imag = imag + obj.imag;
return temp;
}
};
// Usage: Complex c1(3, 4), c2(1, 2); Complex c3 = c1 + c2;
```


Code Example: Operator Overloading
class Complex {
private:
float real, imag;
public:
Complex(float r = 0, float i = 0) : real(r), imag(i) {}
// Overloading the '+' operator
Complex operator+(const Complex& obj) {
Complex temp;
temp.real = real + obj.real;
temp.imag = imag + obj.imag;
return temp;
}
};
// Usage: Complex c1(3, 4), c2(1, 2); Complex c3 = c1 + c2;

In this example, we model a Complex number with real and imaginary parts. We define the 'operator+' member function. Notice that it takes another Complex object as a constant reference. When the compiler sees 'c1 + c2', it actually translates it to the function call 'c1.operator+(c2)'. It adds the respective real and imaginary components and returns a new Complex object. This makes mathematical code involving objects incredibly readable.

Run-Time Polymorphism: Late Binding

- Also known as Dynamic Binding.
- The specific method to execute is determined at the exact moment the code is executed, not during compilation.
- Prerequisites for Run-Time Polymorphism in C++:
- 1. A class hierarchy created through Inheritance.
- 2. Base class pointer (or reference) pointing to a derived class object.
- 3. The use of Virtual Functions in the base class.

Run-Time Polymorphism: Late Binding
- Also known as Dynamic Binding.
- The specific method to execute is determined at the exact moment the code is executed, not during compilation.
- Prerequisites for Run-Time Polymorphism in C++:
- 1. A class hierarchy created through Inheritance.
- 2. Base class pointer (or reference) pointing to a derived class object.
- 3. The use of Virtual Functions in the base class.

Now we reach the heart of OOP: Run-Time Polymorphism. Unlike compile-time, here the compiler cannot know which function to call. Why? Because we might have a pointer of the Base class type, but it might be pointing to a Derived class object in memory. Since memory allocation can happen dynamically at runtime based on user input, the decision of which code to execute must be deferred until runtime. This requires inheritance, pointers or references, and the 'virtual' keyword.

The Role of Virtual Functions

- A virtual function is a member function in the base class that you declare using the 'virtual' keyword.
- It acts as a signal to the compiler: 'Do not use static binding for this function. Wait until runtime to see what exact type of object the pointer is looking at.'
- When a derived class redefines a virtual function from its base class, it is called 'Overriding'.
- Function Overriding requires the exact same signature (name, parameters, and return type) in both base and derived classes.

The Role of Virtual Functions
- A virtual function is a member function in the base class that you declare using the 'virtual' keyword.
- It acts as a signal to the compiler: 'Do not use static binding for this function. Wait until runtime to see what exact type of object the pointer is looking at.'
- When a derived class redefines a virtual function from its base class, it is called 'Overriding'.
- Function Overriding requires the exact same signature (name, parameters, and return type) in both base and derived classes.

The magic key for dynamic binding is the 'virtual' keyword. When you place 'virtual' in front of a base class method, you are telling the C++ compiler to set up a mechanism for dynamic dispatch. If a derived class provides its own version of this virtual function, we say the derived class is 'overriding' the base class function. Note the difference: overloading is same name, different signature. Overriding is same name, exact same signature, across an inheritance boundary.

Under the Hood: vTable and vPtr

- How does C++ actually achieve run-time polymorphism?
- 1. vTable (Virtual Table): A static array of function pointers created by the compiler for any class containing virtual functions.
- 2. vPtr (Virtual Pointer): A hidden pointer added by the compiler to every object of a class with virtual functions.
- At runtime, the vPtr of the specific object being pointed to is used to look up the correct function address in the vTable.
- This lookup is what causes a slight performance overhead compared to compile-time binding.

Under the Hood: vTable and vPtr
- How does C++ actually achieve run-time polymorphism?
- 1. vTable (Virtual Table): A static array of function pointers created by the compiler for any class containing virtual functions.
- 2. vPtr (Virtual Pointer): A hidden pointer added by the compiler to every object of a class with virtual functions.
- At runtime, the vPtr of the specific object being pointed to is used to look up the correct function address in the vTable.
- This lookup is what causes a slight performance overhead compared to compile-time binding.

You might wonder how the program knows which function to call at runtime. C++ implements this using virtual tables, or vTables. Every class with at least one virtual function gets a vTable—an array of function pointers. Every object instantiated from these classes gets a hidden pointer, the vPtr, which points to the class's vTable. When you call a virtual function through a base pointer, the program follows the object's vPtr to the vTable and executes the correct overridden function. This lookup takes a tiny fraction of time, which is the overhead of dynamic binding.

Code Example: Run-Time Polymorphism Setup

```
class Animal {
public:
virtual void makeSound() {
cout ¡¡ "Some generic animal sound" ¡¡ endl;
}
};

class Dog : public Animal {
public:
void makeSound() override { // 'override' is optional but good practice
cout ¡¡ "Woof! Woof!" ¡¡ endl;
}
};
```


Code Example: Run-Time Polymorphism Setup
class Animal {
public:
virtual void makeSound() {
cout ¡¡ "Some generic animal sound" ¡¡ endl;
}
};
class Dog : public Animal {
public:
void makeSound() override { // 'override' is optional but good practice
cout ¡¡ "Woof! Woof!" ¡¡ endl;
}
};

Let's look at the code. We have a base class Animal with a virtual function makeSound(). We then have a derived class Dog that publicly inherits from Animal. Dog provides its own implementation of makeSound(). The 'override' keyword in the derived class is a C++11 feature that asks the compiler to double-check that we are actually overriding a base class virtual function correctly.

Code Example: Dynamic Dispatch in Action

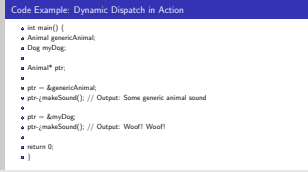
```
int main() {
Animal genericAnimal;
Dog myDog;

Animal* ptr;

ptr = &genericAnimal;
ptr-¿makeSound(); // Output: Some generic animal sound

ptr = &myDog;
ptr-¿makeSound(); // Output: Woof! Woof!

return 0;
}
```

Here is where the magic happens. We create a generic Animal object and a Dog object. Crucially, we create an Animal pointer 'ptr'. When ptr points to genericAnimal, ptr-¿makeSound() calls the base class version. But when we point ptr to myDog (which is allowed because a Dog IS-A Animal), calling ptr-¿makeSound() executes the Dog's version! The compiler didn't know at compile time what ptr would point to, but at runtime, the vPtr ensures 'Woof! Woof!' is printed.

Pure Virtual Functions and Abstract Classes

- Sometimes a base class shouldn't have an implementation for a virtual function.
- Pure Virtual Function: A virtual function assigned a value of 0.
- Syntax: virtual void draw() = 0;
- Abstract Class: Any class containing at least one pure virtual function.
- Properties of Abstract Classes:
- - You cannot instantiate an abstract class (cannot create objects of it).
- - It exists solely to act as a base interface for derived classes.
- - Derived classes MUST override all pure virtual functions, otherwise they too become abstract.

Pure Virtual Functions and Abstract Classes
- Sometimes a base class shouldn't have an implementation for a virtual function.
- Pure Virtual Function: A virtual function assigned a value of 0.
- Syntax: virtual void draw() = 0;
- Abstract Class: Any class containing at least one pure virtual function.
- Properties of Abstract Classes:
- - You cannot instantiate an abstract class (cannot create objects of it).
- - It exists solely to act as a base interface for derived classes.
- - Derived classes MUST override all pure virtual functions, otherwise they too become abstract.

Often, a base class is too generic to implement a function. For example, a 'Shape' class might have a 'draw()' method, but how do you draw a generic shape? You can't. You can only draw specific shapes like Circles or Squares. We make 'draw()' a pure virtual function by appending '= 0'. This makes 'Shape' an abstract class. You cannot create a 'Shape' object directly. It forces all inheriting classes to provide their own specific implementation of 'draw()', guaranteeing a consistent interface across all shapes.

Comparison: Compile-Time vs Run-Time Polymorphism

- — Feature — Compile-Time Polymorphism — Run-Time Polymorphism —
- ————————————————————————————
- — Binding — Early / Static — Late / Dynamic —
- — Mechanism — Function & Operator Overloading — Virtual Functions & Inheritance —
- — Speed — Faster (no run-time overhead) — Slower (vTable lookup overhead) —
- — Flexibility— Less flexible — Highly flexible —
- — Memory — Resolved at compile time — Requires pointers/references to base class —

Comparison: Compile-Time vs Run-Time Polymorphism
— Feature — Compile-Time Polymorphism — Run-Time Polymorphism —
————————————————————————————
— Binding — Early / Static — Late / Dynamic —
— Mechanism — Function & Operator Overloading — Virtual Functions & Inheritance —
— Speed — Faster (no run-time overhead) — Slower (vTable lookup overhead) —
— Flexibility— Less flexible — Highly flexible —
— Memory — Resolved at compile time — Requires pointers/references to base class —

To summarize, compile-time polymorphism is about overloaded names resolved during compilation, offering maximum speed. Run-time polymorphism relies on inheritance and virtual functions, resolved during execution, offering maximum flexibility and making large codebases much easier to maintain and extend without modifying existing code. Both are essential tools in the C++ programmer's toolkit.

Summary & Q&A

- Today we explored:
- - The core concept of Polymorphism ('many forms').
- - How to overload functions and operators statically.
- - How to use virtual functions to achieve dynamic, late-binding behavior.
- - The mechanics of vTables and abstract classes.
- Any questions?

Summary & Q&A
- Today we explored:
- - The core concept of Polymorphism ('many forms').
- - How to overload functions and operators statically.
- - How to use virtual functions to achieve dynamic, late-binding behavior.
- - The mechanics of vTables and abstract classes.
- Any questions?

That wraps up our deep dive into Polymorphism. You now understand how C++ allows identical function calls to behave differently based on context, both statically and dynamically. Please review the vTable concept as it is a frequent topic in technical interviews. I will now open the floor to any questions regarding function overriding versus overloading, or how abstract classes enforce design patterns.