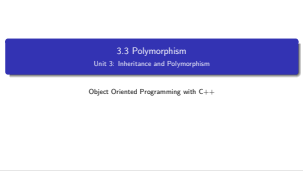
3.3 Polymorphism

Unit 3: Inheritance and Polymorphism

Object Oriented Programming with C++



Lecture 22: Polymorphism in C++

- Unit 3: Inheritance and Polymorphism
- Topic 3.3: Polymorphism
- Compile-time Polymorphism
- Run-time Polymorphism

Lecture 22: Polymorphism in C++
- Unit 3: Inheritance and Polymorphism
- Topic 3.3: Polymorphism
- Compile-time Polymorphism
- Run-time Polymorphism

Welcome everyone to Lecture 22. Today we are tackling one of the most powerful and crucial pillars of Object-Oriented Programming: Polymorphism. We have already covered Encapsulation and Inheritance. Polymorphism is the final piece of the core OOP triad that gives C++ its incredible flexibility.

What is Polymorphism?

- The word 'Polymorphism' is derived from Greek words: 'poly' (many) and 'morphs' (forms).
- In OOP, it refers to the ability of a single function, object, or operator to take on multiple forms.
- It allows you to define one interface and have multiple implementations.
- Core Idea: A single name can represent different behaviors based on the context.

What is Polymorphism?
- The word 'Polymorphism' is derived from Greek words: 'poly' (many) and 'morphs' (forms).
- In OOP, it refers to the ability of a single function, object, or operator to take on multiple forms.
- It allows you to define one interface and have multiple implementations.
- Core Idea: A single name can represent different behaviors based on the context.

Let's break down the word itself. 'Poly' means many, and 'morphs' means forms. So, polymorphism literally translates to 'many forms'. In programming, this means we can use the same name for different things depending on the context. Imagine a generic command like 'speak'. If you issue the command 'speak' to a dog, it barks. If you issue it to a cat, it meows. The command (interface) is the same, but the behavior (implementation) differs based on the object receiving the command.

Types of Polymorphism in C++

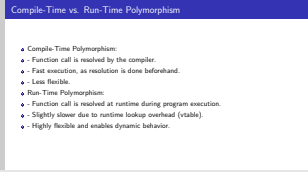
- Polymorphism in C++ is broadly categorized into two types:
- 1. Compile-Time Polymorphism (Static Binding / Early Binding)
- - Achieved via: Function Overloading
- - Achieved via: Operator Overloading
- 2. Run-Time Polymorphism (Dynamic Binding / Late Binding)
- - Achieved via: Virtual Functions

Types of Polymorphism in C++
- Polymorphism in C++ is broadly categorized into two types:
- 1. Compile-Time Polymorphism (Static Binding / Early Binding)
- - Achieved via: Function Overloading
- - Achieved via: Operator Overloading
- 2. Run-Time Polymorphism (Dynamic Binding / Late Binding)
- - Achieved via: Virtual Functions

C++ supports polymorphism at two different stages of a program's lifecycle: during compilation and during execution. Compile-time polymorphism, also known as early binding, happens when the compiler determines which function to call based on the arguments provided. We do this using function overloading and operator overloading. Run-time polymorphism, or late binding, happens when the program is running. The decision of which function to call is delayed until runtime, and this is achieved using inheritance and virtual functions.

Compile-Time vs. Run-Time Polymorphism

- Compile-Time Polymorphism:
- - Function call is resolved by the compiler.
- - Fast execution, as resolution is done beforehand.
- - Less flexible.
- Run-Time Polymorphism:
- - Function call is resolved at runtime during program execution.
- - Slightly slower due to runtime lookup overhead (vtable).
- - Highly flexible and enables dynamic behavior.



Why do we have both? It's a trade-off between performance and flexibility. Compile-time polymorphism is highly efficient because the compiler figures out exactly what code to execute before the program even runs. However, you must know all the types at compile time. Run-time polymorphism adds a tiny bit of overhead because the program has to figure out the object's type on the fly, but it allows you to write highly generic code that can handle new types of objects you haven't even written yet.

Compile-Time: Function Overloading

- Feature where multiple functions can have the same name but different parameters.
- The compiler differentiates them based on the function signature.
- A function signature consists of the function name and the number, type, and sequence of its parameters.
- Return type alone is NOT sufficient for overloading.

Compile-Time: Function Overloading
- Feature where multiple functions can have the same name but different parameters.
- The compiler differentiates them based on the function signature.
- A function signature consists of the function name and the number, type, and sequence of its parameters.
- Return type alone is NOT sufficient for overloading.

Let's dive into Compile-Time polymorphism first, starting with Function Overloading. This is something you might have used already. C++ allows you to create multiple functions with the exact same name, as long as their parameter lists are different. The compiler acts like a detective; it looks at the arguments you pass when you call the function and matches them to the correct function definition. Remember, the return type doesn't count towards the signature for overloading purposes. You can't have two functions with the same parameters but different return types.

Example: Function Overloading

- class MathOps {
- 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; }
- };
- // Usage:
- MathOps math;
- math.add(5, 10); // Calls int add(int, int)
- math.add(5.5, 2.3); // Calls double add(double, double)
- math.add(1, 2, 3); // Calls int add(int, int, int)

Example: Function Overloading
class MathOps {
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; }
};
// Usage:
MathOps math;
math.add(5, 10); // Calls int add(int, int)
math.add(5.5, 2.3); // Calls double add(double, double)
math.add(1, 2, 3); // Calls int add(int, int, int)

Here is a simple example. Our MathOps class has three functions named 'add'. One takes two integers, one takes two doubles, and one takes three integers. When we call math.add(5, 10), the compiler sees two integers and immediately knows to wire that call directly to the first 'add' function. It's clean, intuitive, and happens completely at compile time.

Compile-Time: Operator Overloading Basics

- Provides a way to redefine the way standard C++ operators work with user-defined types (classes).
- Allows objects of custom classes to be used with operators like +, -, ==, etc.
- Syntax involves a special function named 'operator' followed by the symbol.
- Cannot create new operators, only overload existing ones (with a few exceptions like ::, ., .*, ?: which cannot be overloaded).

Compile-Time: Operator Overloading Basics
- Provides a way to redefine the way standard C++ operators work with user-defined types (classes).
- Allows objects of custom classes to be used with operators like +, -, ==, etc.
- Syntax involves a special function named 'operator' followed by the symbol.
- Cannot create new operators, only overload existing ones (with a few exceptions like ::, ., .*, ?: which cannot be overloaded).

The second type of compile-time polymorphism is Operator Overloading. In C++, operators like the plus sign naturally know how to add two integers or two floats. But what if you create a class called 'ComplexNumber'? The compiler doesn't know how to add two complex numbers together. Operator overloading allows you to teach the compiler how to apply operators to your custom objects. This makes your custom objects feel like built-in types, making your code much more readable.

Example: Operator Overloading

- class Complex {
- private: int real, imag;
- public:
- Complex(int r = 0, int i = 0) : real(r), imag(i) {}
- // Overloading the '+' operator
- Complex operator + (const Complex& obj) {
- Complex res;
- res.real = real + obj.real;
- res.imag = imag + obj.imag;
- return res;
- }
- };
- // Usage: Complex c1(10, 5), c2(2, 4); Complex c3 = c1 + c2;

Example: Operator Overloading
class Complex {
private: int real, imag;
public:
Complex(int r = 0, int i = 0) : real(r), imag(i) {}
// Overloading the '+' operator
Complex operator + (const Complex& obj) {
Complex res;
res.real = real + obj.real;
res.imag = imag + obj.imag;
return res;
}
};
// Usage: Complex c1(10, 5), c2(2, 4); Complex c3 = c1 + c2;

Look at this Complex class. We define a special function named 'operator+'. It takes another Complex object as a parameter. When the compiler sees 'c1 + c2', it effectively translates this to a function call: 'c1.operator+(c2)'. It takes the real parts and imaginary parts, adds them, and returns a new Complex object. Now you can use the '+' operator naturally with your own types.

Transition to Run-Time Polymorphism

- Compile-time polymorphism is powerful, but restricted by the compiler's need to know everything in advance.
- What if we have an array of different objects and we want to process them generically?
- To achieve dynamic behavior, we need three ingredients:
- 1. Inheritance (Base and Derived classes)
- 2. Base class pointers pointing to derived class objects.
- 3. Virtual Functions.

Transition to Run-Time Polymorphism
- Compile-time polymorphism is powerful, but restricted by the compiler's need to know everything in advance.
- What if we have an array of different objects and we want to process them generically?
- To achieve dynamic behavior, we need three ingredients:
- 1. Inheritance (Base and Derived classes)
- 2. Base class pointers pointing to derived class objects.
- 3. Virtual Functions.

While compile-time polymorphism is great, it has limits. If you have a zoo simulation with Animal objects, and Derived classes like Dog and Cat, you might want an array of Animals and tell them all to 'speak'. The compiler cannot know in advance whether array index 3 holds a Dog or a Cat. For this dynamic behavior, we must move to Run-Time Polymorphism. To make this work in C++, we absolutely require inheritance, pointers, and a special keyword called 'virtual'.

Pointers to Base Class

- A pointer of a base class type can point to an object of any of its derived classes.
- Base* ptr = new Derived(); // This is valid!
- However, through this base class pointer, you can ONLY access members defined in the Base class.
- If a function is overridden in the Derived class, calling it via the Base pointer normally executes the Base class version (Early Binding).

Pointers to Base Class
- A pointer of a base class type can point to an object of any of its derived classes.
- Base* ptr = new Derived(); // This is valid!
- However, through this base class pointer, you can ONLY access members defined in the Base class.
- If a function is overridden in the Derived class, calling it via the Base pointer normally executes the Base class version (Early Binding).

The cornerstone of run-time polymorphism is a specific rule regarding pointers: A pointer of type Base Class can legally point to an object of a Derived Class. Think of it as 'a Dog IS AN Animal', so an Animal pointer can hold a Dog. However, there's a catch. If you use that Base pointer to call a function, the compiler, doing early binding, looks at the type of the *pointer* (Base), not the object it points to. So it calls the Base class's function.

The Problem: Early Binding

- class Animal {
- public: void speak() { cout ¡¡ "Animal sound" ¡¡ endl; }
- };
- class Dog : public Animal {
- public: void speak() { cout ¡¡ "Woof!" ¡¡ endl; }
- };
- // In main:
- Animal* ptr = new Dog();
- ptr-¿speak(); // Output: "Animal sound"
- // The compiler looked at ptr's type (Animal*) and bound it early.

The Problem: Early Binding
- class Animal {
- public: void speak() { cout ¡¡ "Animal sound" ¡¡ endl; }
- };
- class Dog : public Animal {
- public: void speak() { cout ¡¡ "Woof!" ¡¡ endl; }
- };
- // In main:
- Animal* ptr = new Dog();
- ptr-¿speak(); // Output: "Animal sound"
- // The compiler looked at ptr's type (Animal*) and bound it early.

Let's see this problem in code. We have Animal and Dog. Dog overrides 'speak'. We create a new Dog, but store it in an Animal pointer. When we say ptr-¿speak(), we expect 'Woof!'. But we get 'Animal sound'. Why? Because at compile time, the compiler sees 'ptr' is an Animal pointer, so it hardwires the call to Animal::speak. This is early binding, and it breaks our desired polymorphic behavior.

The Solution: Virtual Functions

- A virtual function is a member function in the base class that is declared using the keyword 'virtual'.
- It tells the compiler to perform Late Binding (Dynamic Binding) for this function.
- When you call a virtual function through a base pointer, the program looks at the actual type of the object pointed to at runtime.
- It then calls the derived class's overridden version of the function.

The Solution: Virtual Functions
- A virtual function is a member function in the base class that is declared using the keyword 'virtual'.
- It tells the compiler to perform Late Binding (Dynamic Binding) for this function.
- When you call a virtual function through a base pointer, the program looks at the actual type of the object pointed to at runtime.
- It then calls the derived class's overridden version of the function.

To fix this, we introduce the 'virtual' keyword. When you place 'virtual' before a function declaration in the Base class, you are giving the compiler a special instruction. You are saying: 'Do not bind this function call at compile time. Wait until runtime, look at the actual object the pointer is pointing to in memory, and call that object's version of the function.' This is Late Binding.

Example: Run-Time Polymorphism

- class Animal {
- public: virtual void speak() { cout ¡¡ "Animal sound" ¡¡ endl; }
- };
- class Dog : public Animal {
- public: void speak() override { cout ¡¡ "Woof!" ¡¡ endl; }
- };
- // In main:
- Animal* ptr = new Dog();
- ptr-¿speak(); // Output: "Woof!" (Late Binding!)

Example: Run-Time Polymorphism
- class Animal {
- public: virtual void speak() { cout ¡¡ "Animal sound" ¡¡ endl; }
- };
- class Dog : public Animal {
- public: void speak() override { cout ¡¡ "Woof!" ¡¡ endl; }
- };
- // In main:
- Animal* ptr = new Dog();
- ptr-¿speak(); // Output: "Woof!" (Late Binding!)

Now we simply add the 'virtual' keyword to the Animal's speak function. Notice I also added 'override' in the Dog class—this is a best practice in modern C++ to ensure you actually are overriding a virtual function, but it's optional. Now, when we run the exact same main code, ptr-¿speak() outputs 'Woof!'. At runtime, the program detects that 'ptr' actually points to a Dog, and dynamically routes the call to Dog::speak. This is true polymorphism.

Behind the Scenes: Virtual Tables (vtable)

- How does late binding work?
- If a class contains a virtual function, the compiler creates a Virtual Table (vtable) for that class.
- The vtable is an array of function pointers pointing to the virtual functions for that class.
- Every object of that class gets a hidden pointer (vptr) that points to the class's vtable.
- At runtime, the program follows the vptr to the vtable to find the correct function address.

Behind the Scenes: Virtual Tables (vtable)
- How does late binding work?
- If a class contains a virtual function, the compiler creates a Virtual Table (vtable) for that class.
- The vtable is an array of function pointers pointing to the virtual functions for that class.
- Every object of that class gets a hidden pointer (vptr) that points to the class's vtable.
- At runtime, the program follows the vptr to the vtable to find the correct function address.

You might wonder how C++ achieves this runtime magic. It uses a mechanism called Virtual Tables, or vtables. When a class has virtual functions, the compiler silently creates a table mapping function names to their actual memory addresses for that specific class. Every object instantiated gets a hidden pointer, called the vptr, pointing to its class's table. When you call a virtual function, the program follows the vptr, looks up the function in the table, and jumps to it. This lookup causes a very small performance hit, which is why C++ doesn't make functions virtual by default.

Pure Virtual Functions & Abstract Classes

- Sometimes a base class represents a generic concept and has no meaningful implementation for a function.
- We can define a Pure Virtual Function using '$= 0$':
- virtual void draw() $= 0$;
- A class with at least one pure virtual function becomes an Abstract Class.
- You CANNOT create objects of an Abstract Class.
- Derived classes MUST override the pure virtual function, otherwise they too become abstract.

Pure Virtual Functions & Abstract Classes
- Sometimes a base class represents a generic concept and has no meaningful implementation for a function.
- We can define a Pure Virtual Function using '$= 0$':
- virtual void draw() $= 0$;
- A class with at least one pure virtual function becomes an Abstract Class.
- You CANNOT create objects of an Abstract Class.
- Derived classes MUST override the pure virtual function, otherwise they too become abstract.

Finally, let's talk about Abstract Classes. Sometimes your base class is just a template. For instance, a 'Shape' class might have a 'draw()' function. But how do you draw a generic shape? You can't. It only makes sense for a Circle or a Square. We can make 'draw()' a Pure Virtual Function by appending '$= 0$' to it. This forces any derived class to provide its own implementation. Any class containing a pure virtual function is an Abstract Class, meaning it exists solely to be inherited from, and you cannot instantiate it directly.

Summary

- Polymorphism allows objects to take many forms and share interfaces.
- Compile-Time Polymorphism (Early Binding): Resolved by compiler. Fast. Uses Function/Operator Overloading.
- Run-Time Polymorphism (Late Binding): Resolved at execution. Flexible. Uses Inheritance, Base Pointers, and Virtual Functions.
- Virtual functions ensure the derived class implementation is executed even via a base class pointer.
- Pure virtual functions create Abstract Classes used as foundational interfaces.

Summary
- Polymorphism allows objects to take many forms and share interfaces.
- Compile-Time Polymorphism (Early Binding): Resolved by compiler. Fast. Uses Function/Operator Overloading.
- Run-Time Polymorphism (Late Binding): Resolved at execution. Flexible. Uses Inheritance, Base Pointers, and Virtual Functions.
- Virtual functions ensure the derived class implementation is executed even via a base class pointer.
- Pure virtual functions create Abstract Classes used as foundational interfaces.

To summarize, polymorphism is what gives C++ object-oriented design its true power. You must know when to use compile-time overloading for simple type variations, and when to design rich inheritance hierarchies using virtual functions to achieve run-time flexibility. Remember that run-time polymorphism requires you to interact with your objects through pointers or references. Review the vtable concept, as it's a common interview question for C++ roles.