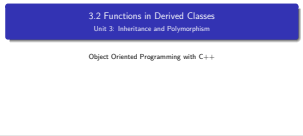
3.2 Functions in Derived Classes

Unit 3: Inheritance and Polymorphism

Object Oriented Programming with C++



Lecture 21: Functions in Derived Classes

- Unit 3: Inheritance and Polymorphism
- Topic: 3.2 Functions in Derived Classes
- Overriding Base Class Functions
- Virtual Functions
- Abstract Classes

Lecture 21: Functions in Derived Classes
- Unit 3: Inheritance and Polymorphism
- Topic: 3.2 Functions in Derived Classes
- Overriding Base Class Functions
- Virtual Functions
- Abstract Classes

Welcome class to Lecture 21. Today we are diving deeper into Unit 3, specifically focusing on how functions behave when we start dealing with derived classes. This lecture is critical because it forms the very foundation of runtime polymorphism in C++, which is one of the most powerful features of object-oriented programming.

Recap & Learning Objectives

- Recap: Inheritance allows a derived class to inherit properties and behaviors from a base class.
- Objective 1: Understand how to override base class functions.
- Objective 2: Grasp the difference between static (early) and dynamic (late) binding.
- Objective 3: Learn to implement polymorphism using virtual functions.
- Objective 4: Understand the concept and application of abstract classes.

Recap & Learning Objectives
- Recap: Inheritance allows a derived class to inherit properties and behaviors from a base class.
- Objective 1: Understand how to override base class functions.
- Objective 2: Grasp the difference between static (early) and dynamic (late) binding.
- Objective 3: Learn to implement polymorphism using virtual functions.
- Objective 4: Understand the concept and application of abstract classes.

Before we begin, let's briefly recall that inheritance is an 'is-a' relationship. By the end of this session, you should be perfectly comfortable with rewriting base class behaviors in a derived class, understanding when the compiler decides which function to call versus when it's decided at runtime, and designing abstract interfaces.

Introduction: Functions in Derived Classes

- Derived classes inherit non-private member functions of the base class.
- A derived class can use inherited functions as they are.
- A derived class can also define its own new functions.
- Crucially, a derived class can provide a new implementation for an existing base class function.

Introduction: Functions in Derived Classes
- Derived classes inherit non-private member functions of the base class.
- A derived class can use inherited functions as they are.
- A derived class can also define its own new functions.
- Crucially, a derived class can provide a new implementation for an existing base class function.

When we create a derived class, it gets access to the public and protected members of the base class. Most of the time, the derived class will just use those inherited functions directly. However, what if a generic behavior defined in the base class doesn't quite fit the specialized derived class? This is where function overriding comes into play.

Overriding Base Class Functions

- Function Overriding occurs when a derived class provides a specific implementation of a function that is already provided by its base class.
- The function in the derived class must have the exact same signature (name, return type, and parameters).
- When the function is called on an object of the derived class, the derived class's version is executed.
- The base class version becomes 'hidden' for that object.

Overriding Base Class Functions
- Function Overriding occurs when a derived class provides a specific implementation of a function that is already provided by its base class.
- The function in the derived class must have the exact same signature (name, return type, and parameters).
- When the function is called on an object of the derived class, the derived class's version is executed.
- The base class version becomes 'hidden' for that object.

Function overriding is redefining a base class function in a derived class. It must have the exact same signature. If you change the parameters, you aren't overriding; you are hiding the base class function and creating a new overloaded function in the derived class scope. When you call an overridden function using a derived object, the derived version executes.

Overriding vs. Overloading

- Overloading:
- - Functions have the same name but different parameters.
- - Occurs within the same scope (same class).
- - Resolved at compile-time (early binding).
- Overriding:
- - Functions have the exact same name and same parameters.
- - Occurs between base and derived classes (different scopes).
- - Can be resolved at runtime (late binding) if virtual functions are used.

Overriding vs. Overloading
- Overloading:
- - Functions have the same name but different parameters.
- - Occurs within the same scope (same class).
- - Resolved at compile-time (early binding).
- Overriding:
- - Functions have the exact same name and same parameters.
- - Occurs between base and derived classes (different scopes).
- - Can be resolved at runtime (late binding) if virtual functions are used.

It's a common interview question to ask the difference between overriding and overloading. Remember, overloading is about providing multiple ways to call a function within the same class, differentiated by parameters. Overriding is about providing a different implementation for an inherited function across class hierarchies.

Example: Function Overriding

- class Animal {
- public:
- void makeSound() { cout ¡¡ "Generic animal sound" ¡¡ endl; }
- };
- class Dog : public Animal {
- public:
- void makeSound() { cout ¡¡ "Woof!" ¡¡ endl; }
- };
- Dog myDog;
- myDog.makeSound(); // Outputs: Woof!

Example: Function Overriding
class Animal {
public:
void makeSound() { cout ¡¡ "Generic animal sound" ¡¡ endl; }
};
class Dog : public Animal {
public:
void makeSound() { cout ¡¡ "Woof!" ¡¡ endl; }
};
Dog myDog;
myDog.makeSound(); // Outputs: Woof!

Let's look at a concrete example. We have a base class Animal with a makeSound function. The Dog class inherits from Animal and overrides makeSound. When we create a Dog object and call makeSound, it correctly outputs 'Woof!'. The Animal's version is ignored for the Dog object.

The Problem with Static Binding

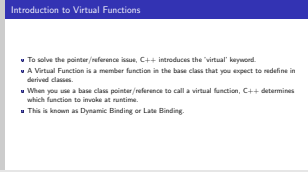
- What happens when we use pointers or references?
- Animal* ptr = new Dog();
- ptr-¿makeSound(); // Outputs: Generic animal sound! (WHY?)
- Early Binding (Static Binding): The compiler matches the function call based on the pointer type, not the object type.
- Since 'ptr' is of type 'Animal*', the compiler binds to Animal::makeSound() at compile time.

The Problem with Static Binding
- What happens when we use pointers or references?
- Animal* ptr = new Dog();
- ptr-¿makeSound(); // Outputs: Generic animal sound! (WHY?)
- Early Binding (Static Binding): The compiler matches the function call based on the pointer type, not the object type.
- Since 'ptr' is of type 'Animal*', the compiler binds to Animal::makeSound() at compile time.

Here's where things get tricky. If we use a base class pointer to point to a derived class object—which is perfectly legal and common in OOP—and call the overridden function, it calls the BASE class version. Why? Because of static binding. The compiler looks at the type of the pointer (which is Animal) and sets up the call to Animal's function during compilation, ignoring the actual object type (Dog) in memory.

Introduction to Virtual Functions

- To solve the pointer/reference issue, C++ introduces the 'virtual' keyword.
- A Virtual Function is a member function in the base class that you expect to redefine in derived classes.
- When you use a base class pointer/reference to call a virtual function, C++ determines which function to invoke at runtime.
- This is known as Dynamic Binding or Late Binding.



To achieve true polymorphism, where the function called depends on the actual object the pointer is pointing to, we use virtual functions. By prepending the keyword 'virtual' to the function declaration in the base class, we tell the compiler not to bind the function call at compile time. Instead, it defers the decision until runtime.

How Virtual Functions Work (Under the Hood)

- Virtual Table (vtable): A static array of function pointers created for classes containing virtual functions.
- Virtual Pointer (vptr): A hidden pointer added by the compiler to the class, pointing to the vtable for that class.
- At runtime, the program follows the object's vptr to the vtable to find the correct overridden function to execute.
- This adds a slight memory and performance overhead but enables powerful polymorphism.

How Virtual Functions Work (Under the Hood)
- Virtual Table (vtable): A static array of function pointers created for classes containing virtual functions.
- Virtual Pointer (vptr): A hidden pointer added by the compiler to the class, pointing to the vtable for that class.
- At runtime, the program follows the object's vptr to the vtable to find the correct overridden function to execute.
- This adds a slight memory and performance overhead but enables powerful polymorphism.

You might wonder how C++ knows which function to call at runtime. It uses a mechanism involving a Virtual Table, or vtable, and a virtual pointer, vptr. Every class with virtual functions gets a vtable. Every object of that class gets a vptr pointing to it. At runtime, the program uses the object's vptr to look up the correct function address. This is the magic behind dynamic binding.

Example: Virtual Functions in Action

- class Animal {
- public:
- virtual void makeSound() { cout ¡¡ "Generic animal sound" ¡¡ endl; }
- };
- class Dog : public Animal {
- public:
- void makeSound() { cout ¡¡ "Woof!" ¡¡ endl; }
- };
- Animal* ptr = new Dog();
- ptr-¿makeSound(); // Outputs: Woof!

Example: Virtual Functions in Action
- class Animal {
- public:
- virtual void makeSound() { cout ¡¡ "Generic animal sound" ¡¡ endl; }
- };
- class Dog : public Animal {
- public:
- void makeSound() { cout ¡¡ "Woof!" ¡¡ endl; }
- };
- Animal* ptr = new Dog();
- ptr-¿makeSound(); // Outputs: Woof!

Let's revisit our earlier code, but now we've added the 'virtual' keyword to Animal's makeSound function. Notice that we don't strictly need to repeat 'virtual' in the Dog class, though it's good practice. Now, when we call ptr-¿makeSound(), the program checks the actual object type at runtime, sees it's a Dog, and outputs 'Woof!'.

The 'override' Specifier (C++11)

- Introduced in C++11, 'override' is placed at the end of a derived class function declaration.
- It tells the compiler: 'I intend to override a virtual function from a base class.'
- If there is a typo, or the signatures don't match exactly, the compiler generates an error.
- Highly recommended for preventing bugs where you accidentally overload instead of override.

The 'override' Specifier (C++11)
- Introduced in C++11, 'override' is placed at the end of a derived class function declaration.
- It tells the compiler: 'I intend to override a virtual function from a base class.'
- If there is a typo, or the signatures don't match exactly, the compiler generates an error.
- Highly recommended for preventing bugs where you accidentally overload instead of override.

Modern C++ introduced the 'override' specifier. It's a lifesaver. Before C++11, if you made a slight typo in the function signature in the derived class, the compiler would quietly treat it as a new overloaded function. Your polymorphic code would fail silently. By using 'override', you force the compiler to verify that you are actually overriding a base class virtual function.

Using the 'override' Specifier

```
class Animal {
public:
virtual void makeSound() const { ... }
};
class Dog : public Animal {
public:
// void makeSound() override; // ERROR: missing 'const'
void makeSound() const override {
cout ¡¡ "Woof!" ¡¡ endl;
}
};
```


Using the 'override' Specifier
class Animal {
public:
virtual void makeSound() const { ... }
};
class Dog : public Animal {
public:
// void makeSound() override; // ERROR: missing 'const'
void makeSound() const override {
cout ¡¡ "Woof!" ¡¡ endl;
}
};

Here is an example. If Animal's makeSound is marked 'const', the overriding function in Dog must also be 'const'. If we forget the 'const' but use 'override', the compiler throws an error immediately, saving us hours of debugging.

Pure Virtual Functions

- Sometimes, a base class has no meaningful implementation for a virtual function.
- A Pure Virtual Function is a virtual function declared in a base class that has no body.
- Syntax: virtual void functionName() = 0;
- It forces any derived class to provide an implementation for this function.

Pure Virtual Functions
- Sometimes, a base class has no meaningful implementation for a virtual function.
- A Pure Virtual Function is a virtual function declared in a base class that has no body.
- Syntax: virtual void functionName() = 0;
- It forces any derived class to provide an implementation for this function.

Moving on to pure virtual functions. Think about a base class called 'Shape'. What is the 'area' of a generic Shape? It doesn't make sense. We know all shapes have an area, but we can't calculate it until we know if it's a Circle or a Rectangle. We declare 'area' as a pure virtual function by assigning it to 0. This enforces a contract: any concrete derived class must implement it.

Abstract Classes

- An Abstract Class is a class containing at least one pure virtual function.
- Key Property: You cannot instantiate (create objects of) an abstract class.
- Animal obj; // ERROR if Animal is abstract.
- Abstract classes are designed solely to act as base classes (interfaces).
- We can still create pointers and references of an abstract class type.

Abstract Classes
- An Abstract Class is a class containing at least one pure virtual function.
- Key Property: You cannot instantiate (create objects of) an abstract class.
- Animal obj; // ERROR if Animal is abstract.
- Abstract classes are designed solely to act as base classes (interfaces).
- We can still create pointers and references of an abstract class type.

The moment a class contains a pure virtual function, it becomes an abstract class. Because it has incomplete functions, the compiler forbids you from creating an object of that class. Abstract classes serve as blueprints or interfaces. They define what derived classes must do, without dictating how they do it.

Example: Abstract Classes and Interfaces

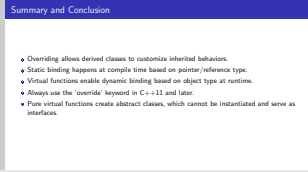
- class Shape { // Abstract class
- public:
- virtual void draw() = 0; // Pure virtual
- };
- class Circle : public Shape {
- public:
- void draw() override { cout ¡¡ "Drawing a Circle"; }
- };
- // Shape s; // ERROR
- Shape* ptr = new Circle();
- ptr-¿draw(); // Valid, outputs Drawing a Circle

Example: Abstract Classes and Interfaces
- class Shape { // Abstract class
- public:
- virtual void draw() = 0; // Pure virtual
- };
- class Circle : public Shape {
- public:
- void draw() override { cout ¡¡ "Drawing a Circle"; }
- };
- // Shape s; // ERROR
- Shape* ptr = new Circle();
- ptr-¿draw(); // Valid, outputs Drawing a Circle

In this final code example, Shape is abstract because of the pure virtual draw() function. We cannot say 'Shape s'. But we can create a Circle, and point to it with a Shape pointer. This allows us to have arrays or collections of Shape pointers, pointing to various circles, rectangles, and triangles, drawing them all polymorphically in a loop.

Summary and Conclusion

- Overriding allows derived classes to customize inherited behaviors.
- Static binding happens at compile time based on pointer/reference type.
- Virtual functions enable dynamic binding based on object type at runtime.
- Always use the 'override' keyword in C++11 and later.
- Pure virtual functions create abstract classes, which cannot be instantiated and serve as interfaces.



To summarize, today we've covered the critical transition from static inheritance to dynamic polymorphism. We learned how to override functions, how virtual functions tell the compiler to wait until runtime to bind the function call, and how abstract classes enforce architectural design patterns. These are essential skills for advanced C++ development.