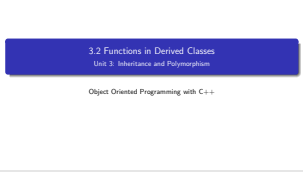
3.2 Functions in Derived Classes

Unit 3: Inheritance and Polymorphism

Object Oriented Programming with C++



Lecture 25: Functions in Derived Classes

- Course: Object Oriented Programming with C++
- Unit 3: Inheritance and Polymorphism
- Topics: Function Overriding, Virtual Functions, Abstract Classes

Lecture 25: Functions in Derived Classes
- Course: Object Oriented Programming with C++
- Unit 3: Inheritance and Polymorphism
- Topics: Function Overriding, Virtual Functions, Abstract Classes

Welcome back to Unit 3. Today, we are going deeper into inheritance. In our last class, we saw how a derived class inherits members from a base class. But what if the derived class needs to change the behavior of an inherited function? Or what if we want to write generic code that can work with any derived class automatically? That's what we'll cover today.

The Need to Redefine Behavior

- Inheritance provides base functionality.
- Derived classes often need to specialize or modify this inherited behavior.
- Example: A 'Shape' base class has a 'draw()' function. A 'Circle' derived class needs its own specific 'draw()' logic.
- C++ allows derived classes to redefine functions inherited from the base class.

The Need to Redefine Behavior
- Inheritance provides base functionality.
- Derived classes often need to specialize or modify this inherited behavior.
- Example: A 'Shape' base class has a 'draw()' function. A 'Circle' derived class needs its own specific 'draw()' logic.
- C++ allows derived classes to redefine functions inherited from the base class.

Think about a base class called 'Animal' with a function 'makeSound()'. If we inherit 'Dog' and 'Cat' from 'Animal', they shouldn't just make a generic animal sound. The Dog needs to bark, and the Cat needs to meow. So, the derived classes need a way to provide their own specific implementation for the 'makeSound' function. This brings us to the concept of Function Overriding.

Overriding Base Class Functions

- Definition: When a derived class provides a specific implementation for a member function that is already provided by its base class.
- The function in the derived class must have the exact same name, return type, and parameter list as the base class function.
- This is also known as Function Redefinition (when not using virtual functions).

Overriding Base Class Functions
- Definition: When a derived class provides a specific implementation for a member function that is already provided by its base class.
- The function in the derived class must have the exact same name, return type, and parameter list as the base class function.
- This is also known as Function Redefinition (when not using virtual functions).

Function overriding is straight-forward in syntax. You just declare a function in the derived class with the exact same signature—same name, same arguments, same return type—as the one in the base class. When you call this function on an object of the derived class, the derived class's version is executed, effectively 'hiding' or 'overriding' the base class's version.

Code Example: Simple Overriding

- class Base {
- public:
- void display() { cout ¡¡ "Base display" ¡¡ endl; }
- };
- class Derived : public Base {
- public:
- void display() { cout ¡¡ "Derived display" ¡¡ endl; }
- };
- Derived obj;
- obj.display(); // Outputs: Derived display

Code Example: Simple Overriding
class Base {
public:
void display() { cout ¡¡ "Base display" ¡¡ endl; }
};
class Derived : public Base {
public:
void display() { cout ¡¡ "Derived display" ¡¡ endl; }
};
Derived obj;
obj.display(); // Outputs: Derived display

Here is a simple example. Base has a display() function. Derived inherits from Base but defines its own display() function. When we create a Derived object 'obj' and call obj.display(), it prints 'Derived display'. The Base class version is hidden. If we wanted to explicitly call the base class version, we would have to use the scope resolution operator: obj.Base::display().

The Problem with Pointers (Early Binding)

- Consider using a Base class pointer pointing to a Derived class object.
- Base* ptr = new Derived();
- ptr-¿display(); // Which function is called?
- Result: The Base class function is called, NOT the Derived class function!

The Problem with Pointers (Early Binding)
- Consider using a Base class pointer pointing to a Derived class object.
- Base* ptr = new Derived();
- ptr-¿display(); // Which function is called?
- Result: The Base class function is called, NOT the Derived class function!

Now we encounter a critical issue. One of the main powers of inheritance is being able to treat derived objects as base objects. We do this using base class pointers. However, if we point a Base pointer to a Derived object, and call the overridden function... C++ calls the Base version! Why? Because the compiler looks at the type of the pointer (Base*) at compile time, not the actual object it points to (Derived). This is called early binding or static binding.

Introduction to Polymorphism

- Polymorphism means 'many forms'.
- In C++, it means a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- Compile-time (Early Binding): Function overloading, operator overloading.
- Run-time (Late Binding/Dynamic Binding): Virtual functions.

Introduction to Polymorphism
- Polymorphism means 'many forms'.
- In C++, it means a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.
- Compile-time (Early Binding): Function overloading, operator overloading.
- Run-time (Late Binding/Dynamic Binding): Virtual functions.

To solve the pointer problem, we need Polymorphism. Specifically, run-time polymorphism. We want the program to decide which function to call while it's running, based on the actual object the pointer is pointing to, not just the pointer's declared type. This decision process at runtime is called Late Binding or Dynamic Binding.

Virtual Functions

- A virtual function is a member function in the base class that you expect to redefine in derived classes.
- Declared using the `virtual` keyword in the base class.
- It tells the compiler to perform dynamic binding (late binding) for this function.
- Ensures the correct function is called for an object, regardless of the type of pointer used for the function call.

Virtual Functions
- A virtual function is a member function in the base class that you expect to redefine in derived classes.
- Declared using the virtual keyword in the base class.
- It tells the compiler to perform dynamic binding (late binding) for this function.
- Ensures the correct function is called for an object, regardless of the type of pointer used for the function call.

This is where the 'virtual' keyword comes in. By placing the word 'virtual' before a function declaration in the base class, we are explicitly telling the C++ compiler: 'Do not use early binding for this function. Wait until runtime, look at the actual object being pointed to, and call that object's version of the function.'

Code Example: Virtual Functions

- class Base {
- public:
- virtual void display() { cout ¡¡ "Base display" ¡¡ endl; }
- };
- class Derived : public Base {
- public:
- void display() override { cout ¡¡ "Derived display" ¡¡ endl; }
- };
- Base* ptr = new Derived();
- ptr-¿display(); // Outputs: Derived display! (Dynamic Binding)

Code Example: Virtual Functions
class Base {
public:
virtual void display() { cout ¡¡ "Base display" ¡¡ endl; }
};
class Derived : public Base {
public:
void display() override { cout ¡¡ "Derived display" ¡¡ endl; }
};
Base* ptr = new Derived();
ptr-¿display(); // Outputs: Derived display! (Dynamic Binding)

Look at the same example, but now we've added 'virtual' to the Base class display() function. Notice I also added the 'override' keyword in the derived class. This is optional but highly recommended in modern C++ (C++11 onwards) as it makes the compiler check that you are actually overriding a virtual function. Now, when ptr-¿display() is called, it correctly outputs 'Derived display'. This is runtime polymorphism in action.

Rules for Virtual Functions

- Virtual functions must be members of some class.
- They cannot be static members.
- They are accessed through object pointers or references to achieve polymorphism.
- The prototype (signature) must be identical in base and derived classes.
- Constructors cannot be virtual, but destructors should be virtual if a class has virtual functions.

Rules for Virtual Functions
- Virtual functions must be members of some class.
- They cannot be static members.
- They are accessed through object pointers or references to achieve polymorphism.
- The prototype (signature) must be identical in base and derived classes.
- Constructors cannot be virtual, but destructors should be virtual if a class has virtual functions.

There are some important rules to remember. You can't have a virtual static function. You only get polymorphic behavior when using pointers or references. If you just use normal object variables, early binding happens. And very importantly, if your base class has any virtual functions, you must make its destructor virtual too. We will discuss virtual destructors in detail in a future lecture, but it prevents memory leaks when deleting derived objects through base pointers.

The 'override' Specifier (C++11)

- Added in C++11 to prevent subtle bugs.
- Placed at the end of a derived class function declaration.
- Forces the compiler to check if a base class has a matching virtual function.
- If signatures don't match exactly (e.g., const mismatch, slight typo), compilation fails.

The 'override' Specifier (C++11)
- Added in C++11 to prevent subtle bugs.
- Placed at the end of a derived class function declaration.
- Forces the compiler to check if a base class has a matching virtual function.
- If signatures don't match exactly (e.g., const mismatch, slight typo), compilation fails.

Before C++11, if you made a typo in the derived class function—say, you forgot a 'const' qualifier—the compiler wouldn't override the base function; it would just create a brand new function, hiding the base one. This caused terrible bugs. The 'override' specifier tells the compiler 'I intend to override a virtual function here. If I didn't get the signature exactly right, give me an error.' Always use it.

Pure Virtual Functions

- Sometimes a base class cannot provide a meaningful implementation for a virtual function.
- A pure virtual function is declared by assigning 0 to it.
- Syntax: `virtual return_type function_name(parameters) = 0;`
- It forces any derived class to provide its own implementation.

Pure Virtual Functions
- Sometimes a base class cannot provide a meaningful implementation for a virtual function.
- A pure virtual function is declared by assigning 0 to it.
- Syntax: virtual return_type function_name(parameters) = 0;
- It forces any derived class to provide its own implementation.

Consider our 'Shape' class. What does a generic 'Shape' look like? It doesn't. You can draw a Circle or a Square, but you can't just draw a 'Shape'. Therefore, the 'draw()' function in the Shape base class has no meaningful code. Instead of leaving it empty, we make it a Pure Virtual Function by appending '= 0' to the declaration. This acts as a strict contract.

Abstract Classes

- An Abstract Class is a class that contains at least one pure virtual function.
- You CANNOT instantiate objects of an abstract class.
- It serves solely as a blueprint or interface for derived classes.
- Derived classes must override all pure virtual functions to become instantiable (concrete classes).

Abstract Classes
- An Abstract Class is a class that contains at least one pure virtual function.
- You CANNOT instantiate objects of an abstract class.
- It serves solely as a blueprint or interface for derived classes.
- Derived classes must override all pure virtual functions to become instantiable (concrete classes).

The moment a class contains even one pure virtual function, it becomes an Abstract Class. Because it has incomplete functions, the compiler will not allow you to create objects of this class. 'Shape s;' would cause a compiler error. Abstract classes exist purely to be inherited from, providing a common interface—a set of rules—that all derived classes must follow.

Code Example: Abstract Classes

- class Shape { // Abstract Class
- public:
- virtual void draw() = 0; // Pure virtual
- };
- class Circle : public Shape {
- public:
- void draw() override { cout ¡¡ "Drawing Circle"; }
- };
- // Shape s; // ERROR: Cannot instantiate
- Shape* ptr = new Circle();
- ptr-¿draw(); // Valid, dynamic binding

Code Example: Abstract Classes
- class Shape { // Abstract Class
- public:
- virtual void draw() = 0; // Pure virtual
- };
- class Circle : public Shape {
- public:
- void draw() override { cout ¡¡ "Drawing Circle"; }
- };
- // Shape s; // ERROR: Cannot instantiate
- Shape* ptr = new Circle();
- ptr-¿draw(); // Valid, dynamic binding

Here is the code representation. Shape is abstract because of the `= 0` on draw(). We cannot say `Shape s;`. However, we can create pointers to abstract classes! We can have a `Shape*`. We then point that Shape to a `Circle` object, and when we call `ptr->draw()`, it magically calls the Circle's draw function. This is the ultimate power of OOP interfaces.

Summary of Polymorphism

- 1. Base class pointer to Derived class object.
- 2. Base class function is `virtual`.
- 3. Result: Call resolves at RUNTIME (Dynamic Binding) to the actual object type.
- 4. Abstract classes enforce interfaces using pure virtual functions (= 0).

Summary of Polymorphism
- 1. Base class pointer to Derived class object.
- 2. Base class function is virtual.
- 3. Result: Call resolves at RUNTIME (Dynamic Binding) to the actual object type.
- 4. Abstract classes enforce interfaces using pure virtual functions (= 0).

To summarize, to achieve true runtime polymorphism in C++, you need three ingredients: inheritance, a base class pointer or reference pointing to a derived object, and virtual functions. If you want to force an interface without providing a default implementation, you use pure virtual functions, which turn your class into an abstract class.