3.4 Constructors and Destructors in Derived Classes

Unit 3: Inheritance and Polymorphism

Object Oriented Programming with C++

3.4 Constructors and Destructors in Derived Classes
Unit 3: Inheritance and Polymorphism
Object Oriented Programming with C++

Constructors and Destructors in Derived Classes

- Object Oriented Programming with C++
- Lecture 27: 3.4 Constructors and Destructors in Derived Classes
- Unit 3: Inheritance and Polymorphism

Constructors and Destructors in Derived Classes
- Object Oriented Programming with C++
- Lecture 27: 3.4 Constructors and Destructors in Derived Classes
- Unit 3: Inheritance and Polymorphism

Welcome back to our journey through Object Oriented Programming with C++. Today, we are diving deep into the lifecycle of objects in an inheritance hierarchy. We'll explore exactly what happens when you create or destroy an object of a derived class.

The Lifecycle of a Derived Object

- When an object of a derived class is created, it contains within it a sub-object of its base class.
- Therefore, the base class portion must be initialized before the derived class portion.
- Similarly, when the object is destroyed, the derived class portion must be cleaned up before the base class portion.
- This is handled automatically by C++ through the strict ordering of constructors and destructors.

The Lifecycle of a Derived Object
- When an object of a derived class is created, it contains within it a sub-object of its base class.
- Therefore, the base class portion must be initialized before the derived class portion.
- Similarly, when the object is destroyed, the derived class portion must be cleaned up before the base class portion.
- This is handled automatically by C++ through the strict ordering of constructors and destructors.

Think of building a house. You have to lay the foundation (the base class) before you can build the walls and roof (the derived class). If you don't initialize the base class first, the derived class might try to use uninitialized data inherited from the base class. The reverse is true for demolition: tear down the roof before digging up the foundation.

Order of Constructor Execution

- Constructors are executed from the top of the inheritance hierarchy down to the bottom.
- Order: Base Class Constructor -¿ Derived Class Constructor.
- If there are multiple levels (Multilevel Inheritance): Grandparent -¿ Parent -¿ Child.
- The compiler implicitly inserts a call to the base class's default constructor if no explicit call is made.

Order of Constructor Execution
- Constructors are executed from the top of the inheritance hierarchy down to the bottom.
- Order: Base Class Constructor -¿ Derived Class Constructor.
- If there are multiple levels (Multilevel Inheritance): Grandparent -¿ Parent -¿ Child.
- The compiler implicitly inserts a call to the base class's default constructor if no explicit call is made.

The rule is simple: top-down construction. The most fundamental class in the hierarchy gets built first. This ensures that every derived class has a fully constructed base class to rely upon. If your base class doesn't have a default constructor, you will get a compiler error unless you explicitly call another constructor.

Example: Default Constructors

- class Base { public: Base() { cout ¡¡ "Base Constructor" ¡¡ endl; } };
- class Derived : public Base { public: Derived() { cout ¡¡ "Derived Constructor" ¡¡ endl; } };
- int main() { Derived d; return 0; }
- Output: Base Constructor Derived Constructor

Example: Default Constructors
- class Base { public: Base() { cout ¡¡ "Base Constructor" ¡¡ endl; } };
- class Derived : public Base { public: Derived() { cout ¡¡ "Derived Constructor" ¡¡ endl; } };
- int main() { Derived d; return 0; }
- Output: Base Constructor Derived Constructor

Let's look at this simple example. When we create object 'd' of type Derived, the compiler first allocates memory for the entire object. Then, it calls the Base class constructor. Once that finishes, it executes the body of the Derived class constructor. The output clearly shows this sequence.

Passing Arguments to Base Constructors

- What if the Base class requires arguments for initialization? (No default constructor).
- The Derived class constructor must explicitly call the Base class constructor.
- This is done using the Constructor Initializer List.
- Syntax: Derived(args) : Base(args_for_base) { ... }

Passing Arguments to Base Constructors
- What if the Base class requires arguments for initialization? (No default constructor).
- The Derived class constructor must explicitly call the Base class constructor.
- This is done using the Constructor Initializer List.
- Syntax: Derived(args) : Base(args_for_base) { ... }

Often, a base class is designed to require specific initial state, meaning it only has parameterized constructors. In this case, the compiler cannot automatically call a default constructor. The derived class constructor takes on the responsibility of passing the necessary arguments up the chain using the initializer list.

Example: Parameterized Constructors

- class Base { int x; public: Base(int i) : x(i) { cout ¡¡ "Base initialized with " ¡¡ x ¡¡ endl; } };
- class Derived : public Base { int y; public: // Explicitly call Base constructor Derived(int i, int j) : Base(i), y(j) { cout ¡¡ "Derived initialized with " ¡¡ y ¡¡ endl; } };

Example: Parameterized Constructors
- class Base { int x; public: Base(int i) : x(i) { cout ¡¡ "Base initialized with " ¡¡ x ¡¡ endl; } };
- class Derived : public Base { int y; public: // Explicitly call Base constructor Derived(int i, int j) : Base(i), y(j) { cout ¡¡ "Derived initialized with " ¡¡ y ¡¡ endl; } };

Notice the syntax on the Derived constructor. After the parameter list `(int i, int j)`, we place a colon, followed by the call to the Base class constructor `Base(i)`. This happens before the body of the Derived constructor executes. It's the only way to initialize the `x` member of the Base class.

Constructors in Multiple Inheritance

- In multiple inheritance, a class inherits from more than one base class.
- class Derived : public Base1, public Base2 { ... };
- Order of execution: Determined by the order of inheritance in the class declaration.
- Base1 constructor -¿ Base2 constructor -¿ Derived constructor.

Constructors in Multiple Inheritance
- In multiple inheritance, a class inherits from more than one base class.
- class Derived : public Base1, public Base2 { ... };
- Order of execution: Determined by the order of inheritance in the class declaration.
- Base1 constructor -¿ Base2 constructor -¿ Derived constructor.

When dealing with multiple inheritance, the C++ standard dictates that base class constructors are called in the order they appear in the class declaration list, NOT in the order they appear in the initializer list. It's best practice to make your initializer list match the declaration order to avoid confusion.

Example: Multiple Inheritance

- class B1 { public: B1() { cout ¡¡ "B1\n"; } };
- class B2 { public: B2() { cout ¡¡ "B2\n"; } };
- // B1 is listed first, then B2
- class D : public B1, public B2 { public: D() { cout ¡¡ "D\n"; } };
- Output: B1 B2 D

Example: Multiple Inheritance
- class B1 { public: B1() { cout ¡¡ "B1\n"; } };
- class B2 { public: B2() { cout ¡¡ "B2\n"; } };
- // B1 is listed first, then B2
- class D : public B1, public B2 { public: D() { cout ¡¡ "D\n"; } };
- Output: B1 B2 D

Because the declaration is `class D : public B1, public B2`, B1's constructor is called first, followed by B2's, and finally D's. If we changed the declaration to `class D : public B2, public B1`, the output would be B2, B1, D.

Order of Destructor Execution

- Destructors are executed in the exact reverse order of constructors.
- Order: Derived Class Destructor -¿ Base Class Destructor.
- Bottom-up destruction.
- The compiler implicitly inserts a call to the base class destructor at the end of the derived destructor's execution.

Order of Destructor Execution
- Destructors are executed in the exact reverse order of constructors.
- Order: Derived Class Destructor -¿ Base Class Destructor.
- Bottom-up destruction.
- The compiler implicitly inserts a call to the base class destructor at the end of the derived destructor's execution.

Destruction is tearing down the house. You have to remove the new additions (derived class) before you can touch the foundation (base class). The derived class might hold pointers or resources that depend on the base class still being intact. Therefore, the derived class must be cleaned up first.

Example: Destructor Order

- class Base { public: ~Base() { cout ¡¡ "Base Destructor" ¡¡ endl; } };
- class Derived : public Base { public: ~Derived() { cout ¡¡ "Derived Destructor" ¡¡ endl; } };
- Output when Derived goes out of scope: Derived Destructor Base Destructor

Example: Destructor Order
- class Base { public: ~Base() { cout ¡¡ "Base Destructor" ¡¡ endl; } };
- class Derived : public Base { public: ~Derived() { cout ¡¡ "Derived Destructor" ¡¡ endl; } };
- Output when Derived goes out of scope: Derived Destructor Base Destructor

As we can see, when the `Derived` object is destroyed, the `~Derived()` destructor runs first. Once its body completes, the compiler automatically calls `~Base()`. You do not explicitly call base class destructors.

A Crucial Polymorphism Problem

- Consider this scenario: Base* ptr = new Derived();
- What happens when we call: delete ptr; ?
- Only the Base class destructor is called!
- The Derived class destructor is skipped, leading to a memory leak or resource leak if Derived allocated memory.

A Crucial Polymorphism Problem
- Consider this scenario: Base* ptr = new Derived();
- What happens when we call: delete ptr; ?
- Only the Base class destructor is called!
- The Derived class destructor is skipped, leading to a memory leak or resource leak if Derived allocated memory.

This is a classic C++ interview question and a very common source of bugs. If you delete an object of a derived class through a pointer to a base class, and the base class destructor is non-virtual, the behavior is undefined by the C++ standard. In practice, most compilers will only invoke the Base class destructor.

The Problem in Action

- class Base { public: ~Base() { cout¡¡"~Base\n"; } };
- class Derived : public Base { int* data; public: Derived() { data = new int[100]; } ~Derived() { delete[] data; cout¡¡"~Derived\n"; } };
- Base* ptr = new Derived(); delete ptr; // Output: ~Base (Memory Leak!)

The Problem in Action
- class Base { public: ~Base() { cout¡¡"~Base\n"; } };
- class Derived : public Base { int* data; public: Derived() { data = new int[100]; } ~Derived() { delete[] data; cout¡¡"~Derived\n"; } };
- Base* ptr = new Derived(); delete ptr; // Output: ~Base (Memory Leak!)

Here, `Derived` allocates memory. But because we delete it through a `Base*`, the compiler statically binds the destructor call to `~Base()`. The `~Derived()` destructor never runs, and the array `data` is leaked.

The Solution: Virtual Destructors

- To ensure proper destruction, declare the Base class destructor as virtual.
- virtual ~Base() { ... }
- This tells the compiler to use dynamic binding (late binding) for the destructor.
- When 'delete ptr' is called, it looks at the actual object type at runtime and calls the correct Derived destructor first.

The Solution: Virtual Destructors
- To ensure proper destruction, declare the Base class destructor as virtual.
- virtual ~Base() { ... }
- This tells the compiler to use dynamic binding (late binding) for the destructor.
- When 'delete ptr' is called, it looks at the actual object type at runtime and calls the correct Derived destructor first.

By making the base class destructor virtual, you guarantee that the destruction process starts at the most-derived class, exactly as it should. The virtual mechanism ensures the correct destructor is found via the vtable, and from there, the normal bottom-up destruction sequence proceeds automatically.

Virtual Destructors in Action

- class Base { public: virtual ~Base() { cout¡¡"~Base\n"; } };
- class Derived : public Base { public: ~Derived() { cout¡¡"~Derived\n"; } };
- Base* ptr = new Derived(); delete ptr; // Output: // ~Derived // ~Base

Virtual Destructors in Action
- class Base { public: virtual ~Base() { cout¡¡"~Base\n"; } };
- class Derived : public Base { public: ~Derived() { cout¡¡"~Derived\n"; } };
- Base* ptr = new Derived(); delete ptr; // Output: // ~Derived // ~Base

Adding the `virtual` keyword fixes the issue completely. Now, `delete ptr` correctly identifies that `ptr` points to a `Derived` object, calls `~Derived()`, which in turn automatically calls `~Base()`. No memory leaks, safe polymorphic cleanup.

Summary

- Constructors execute Top-Down: Base first, then Derived.
- Destructors execute Bottom-Up: Derived first, then Base.
- Derived classes must use the Initializer List to pass arguments to parameterized Base constructors.
- Multiple Inheritance constructor order is dictated by the class declaration order.
- ALWAYS use a virtual destructor in a base class if you intend to manipulate derived objects via base pointers.

Summary
- Constructors execute Top-Down: Base first, then Derived.
- Destructors execute Bottom-Up: Derived first, then Base.
- Derived classes must use the Initializer List to pass arguments to parameterized Base constructors.
- Multiple Inheritance constructor order is dictated by the class declaration order.
- ALWAYS use a virtual destructor in a base class if you intend to manipulate derived objects via base pointers.

To wrap up: understanding the sequence of construction and destruction is vital for managing resources effectively in C++. Remember the top-down / bottom-up rule, master the initializer list for passing arguments, and never forget your virtual destructors when using polymorphism.