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++

Lecture 23: Constructors and Destructors in Derived Classes

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

Lecture 23: Constructors and Destructors in Derived Classes
- Course: Object Oriented Programming with C++
- Unit 3: Inheritance and Polymorphism
- Topic 3.4: Constructors and Destructors in Derived Classes

Welcome everyone. Today we are focusing on a critical aspect of C++ inheritance: how objects of derived classes are initialized and destroyed. We will explore the mechanics behind constructors and destructors in an inheritance hierarchy, which is fundamental to writing robust and leak-free object-oriented systems.

The Need for Constructors in Derived Classes

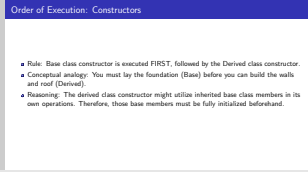
- A derived class inherently contains a sub-object of its base class.
- When a derived class object is created, it needs to initialize both its own new members and the inherited base class members.
- Since a derived class cannot directly initialize private members of its base class, it must rely on the base class's constructor.
- This cooperative initialization process requires a strict order of execution.

The Need for Constructors in Derived Classes
- A derived class inherently contains a sub-object of its base class.
- When a derived class object is created, it needs to initialize both its own new members and the inherited base class members.
- Since a derived class cannot directly initialize private members of its base class, it must rely on the base class's constructor.
- This cooperative initialization process requires a strict order of execution.

When we instantiate a derived class object, it physically contains a sub-object of the base class in memory. Because the derived class doesn't have direct access to private members of the base class, it cannot initialize them directly. It must trigger the base class constructor to handle that part of the initialization. This is why understanding the sequence of constructor execution is crucial.

Order of Execution: Constructors

- Rule: Base class constructor is executed FIRST, followed by the Derived class constructor.
- Conceptual analogy: You must lay the foundation (Base) before you can build the walls and roof (Derived).
- Reasoning: The derived class constructor might utilize inherited base class members in its own operations. Therefore, those base members must be fully initialized beforehand.



The golden rule of constructors in inheritance is 'Base before Derived'. Think of it like constructing a building: the foundation must be completed before the upper floors can be added. If the derived class constructor tries to use a base class member, that member must already be fully formed and initialized. Hence, C++ enforces top-down initialization.

Example: Implicit Default Constructor Execution

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

Example: Implicit Default Constructor Execution
- class Base { public: Base() { cout ¡¡ "Base Constructor\n"; } };
- class Derived : public Base { public: Derived() { cout ¡¡ "Derived Constructor\n"; } };
- int main() { Derived d; return 0; }
- Output: Base Constructor Derived Constructor

In this simple example, we have a Base class and a Derived class, both featuring default constructors. When we create an object 'd' of type Derived in main(), the C++ compiler automatically calls the Base class default constructor before it executes the Derived class constructor body. The output clearly proves the Base-then-Derived sequence.

Parameterized Constructors in Inheritance

- If a base class does not have a default constructor (only parameterized), the compiler cannot automatically invoke one.
- In this scenario, the derived class MUST explicitly call the base class's parameterized constructor.
- This explicit call is achieved using the constructor initialization list (initializer list).
- The derived class passes the necessary arguments up to the base class.

Parameterized Constructors in Inheritance
- If a base class does not have a default constructor (only parameterized), the compiler cannot automatically invoke one.
- In this scenario, the derived class MUST explicitly call the base class's parameterized constructor.
- This explicit call is achieved using the constructor initialization list (initializer list).
- The derived class passes the necessary arguments up to the base class.

What happens if the base class requires arguments to be initialized? The compiler doesn't magically know what arguments to pass! In this case, the derived class constructor takes on the responsibility of providing those arguments to the base class. If it fails to explicitly do so, the compiler will throw an error. We use the initializer list for this exact purpose.

Syntax: Passing Arguments to Base Constructor

- DerivedClass(arg1, arg2, ...) : BaseClass(arg1) {
- // Body of the Derived class constructor
- // Initialization of derived members using arg2, etc.
- }
- The colon (:) begins the constructor initialization list.
- The call to BaseClass(arg1) dictates which base constructor runs before the Derived block.

Syntax: Passing Arguments to Base Constructor
- DerivedClass(arg1, arg2, ...) : BaseClass(arg1) {
- // Body of the Derived class constructor
- // Initialization of derived members using arg2, etc.
- }
- The colon (:) begins the constructor initialization list.
- The call to BaseClass(arg1) dictates which base constructor runs before the Derived block.

Let's review the syntax. Notice the colon placed after the derived class constructor's parameter list. This marks the start of the initializer list. Inside it, we specify the base class name and pass the required arguments. This explicit directive ensures the base sub-object is correctly constructed before the derived class's own constructor body begins execution.

Example: Explicitly Calling Base Constructor

- class Base { int x; public: Base(int i) { x = i; cout ¡¡ "Base: " ¡¡ x ¡¡ " "; } };
- class Derived : public Base { int y;
- public: Derived(int i, int j) : Base(i) { y = j; cout ¡¡ "Derived: " ¡¡ y ¡¡ "\n"; } };
- int main() { Derived d(10, 20); }
- Output: Base: 10 Derived: 20

Example: Explicitly Calling Base Constructor
- class Base { int x; public: Base(int i) { x = i; cout ¡¡ "Base: " ¡¡ x ¡¡ " "; } };
- class Derived : public Base { int y;
- public: Derived(int i, int j) : Base(i) { y = j; cout ¡¡ "Derived: " ¡¡ y ¡¡ "\n"; } };
- int main() { Derived d(10, 20); }
- Output: Base: 10 Derived: 20

Look closely at this snippet. The Derived constructor takes two arguments: 'i' and 'j'. It routes 'i' up to the Base constructor via the initializer list : `Base(i)`. It then utilizes 'j' to initialize its own member 'y' inside the body. When `Derived d(10, 20)` executes, 10 goes to Base, and 20 goes to Derived. The output confirms our theory.

Constructors in Multiple Inheritance

- In multiple inheritance, a single derived class inherits from two or more direct base classes.
- Rule: Base class constructors are called in the exact order they appear in the class derivation list.
- Example: class Derived : public Base1, public Base2 { ... };
- Execution sequence: Base1(), then Base2(), then Derived().

Constructors in Multiple Inheritance
- In multiple inheritance, a single derived class inherits from two or more direct base classes.
- Rule: Base class constructors are called in the exact order they appear in the class derivation list.
- Example: class Derived : public Base1, public Base2 { ... };
- Execution sequence: Base1(), then Base2(), then Derived().

C++ supports multiple inheritance, where a class can have multiple parents. When dealing with multiple base classes, how does the compiler decide which base constructor runs first? The rule is strict: it follows the order declared in the class derivation list, reading left to right. It completely ignores the order in which you might list them in the initializer list.

Example: Multiple Inheritance Constructor Order

- class A { public: A() { cout ¡¡ "A "; } };
- class B { public: B() { cout ¡¡ "B "; } };
- class C : public B, public A { public: C() { cout ¡¡ "C"; } };
- int main() { C obj; }
- Output: B A C

Example: Multiple Inheritance Constructor Order
- class A { public: A() { cout ¡¡ "A "; } };
- class B { public: B() { cout ¡¡ "B "; } };
- class C : public B, public A { public: C() { cout ¡¡ "C"; } };
- int main() { C obj; }
- Output: B A C

Notice the declaration `class C : public B, public A`. Because B is listed first in the inheritance list, B's constructor is executed before A's. Finally, C's constructor is executed. This output, 'B A C', frequently catches students off guard in exams. Always trust the derivation list order, not alphabetical or logical assumptions.

Constructors in Multilevel Inheritance

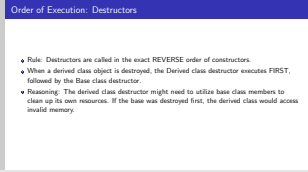
- In multilevel inheritance (e.g., Class A -¿ Class B -¿ Class C), constructors execute in a cascading top-down flow.
- The highest base class is constructed first, down to the most specialized derived class.
- Execution order for an object of Class C: A() -¿ B() -¿ C().
- Delegation: Class C passes arguments to B, and B is responsible for passing arguments to A.

Constructors in Multilevel Inheritance
- In multilevel inheritance (e.g., Class A -¿ Class B -¿ Class C), constructors execute in a cascading top-down flow.
- The highest base class is constructed first, down to the most specialized derived class.
- Execution order for an object of Class C: A() -¿ B() -¿ C().
- Delegation: Class C passes arguments to B, and B is responsible for passing arguments to A.

For multilevel inheritance, think of a grandparent-parent-child relationship. Construction happens top-down. The grandparent is constructed, then the parent, then the child. If parameterized constructors are involved, class C must hand parameters to B, and B must hand parameters to A. C cannot bypass B to directly pass parameters to A's constructor (unless dealing with virtual base classes, an advanced topic).

Order of Execution: Destructors

- Rule: Destructors are called in the exact REVERSE order of constructors.
- When a derived class object is destroyed, the Derived class destructor executes FIRST, followed by the Base class destructor.
- Reasoning: The derived class destructor might need to utilize base class members to clean up its own resources. If the base was destroyed first, the derived class would access invalid memory.



Now let's switch gears to destruction. The rule is elegantly simple: Reverse order of construction, or Bottom-Up. When an object is destroyed, the derived class cleans up its own specialized resources first. Once the derived part is safely dismantled, the compiler automatically triggers the base part's destruction. This guarantees the derived class doesn't attempt to access base members that have already been deallocated.

Example: Destructor Execution Order

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

Example: Destructor Execution Order
- class Base { public: ~Base() { cout ¡¡ "~Base() "; } };
- class Derived : public Base { public: ~Derived() { cout ¡¡ "~Derived() "; } };
- int main() { Derived d; return 0; }
- Output: ~Derived() ~Base()

In this block, when the main function reaches the end, the object 'd' goes out of scope, and its destruction is initiated. First, `~Derived()` executes and prints to the console. Immediately after, the compiler implicitly invokes `~Base()`. Note that you never explicitly call a base class destructor in your code; the compiler manages this automatically.

Best Practice: Virtual Destructors

- When using pointers to base classes to manage derived objects (polymorphism), you MUST make the base class destructor `virtual`.
- Scenario: `Base* ptr = new Derived(); delete ptr;`
- If `~Base()` is non-virtual, only `~Base()` executes! This causes a memory leak for Derived's unique resources.
- If `virtual ~Base()`, then `~Derived()` is correctly called first, followed by `~Base()`.

Best Practice: Virtual Destructors
- When using pointers to base classes to manage derived objects (polymorphism), you MUST make the base class destructor virtual.
- Scenario: Base* ptr = new Derived(); delete ptr;
- If ~Base() is non-virtual, only ~Base() executes! This causes a memory leak for Derived's unique resources.
- If virtual ~Base(), then ~Derived() is correctly called first, followed by ~Base().

While we will cover polymorphism more deeply soon, this rule is vital to introduce now. If you ever dynamically allocate a derived class object and point to it with a base class pointer, and then `delete` it, the base class MUST have a virtual destructor. Without the `virtual` keyword, C++ uses early binding and only executes the base destructor, leaving derived resources orphaned in memory. Always declare virtual destructors in base classes meant for polymorphic use.

Summary

- Constructors execute Top-Down (Base then Derived).
- Destructors execute Bottom-Up (Derived then Base).
- Derived classes explicitly call parameterized base constructors using initialization lists.
- Multiple inheritance base constructors run based on the derivation list order.
- Virtual destructors are critical for memory safety in polymorphism.

Summary
- Constructors execute Top-Down (Base then Derived).
- Destructors execute Bottom-Up (Derived then Base).
- Derived classes explicitly call parameterized base constructors using initialization lists.
- Multiple inheritance base constructors run based on the derivation list order.
- Virtual destructors are critical for memory safety in polymorphism.

To wrap up today's lecture, memorize the top-down construction and bottom-up destruction rules. Mastering the constructor initializer list is essential for managing parameterized constructors across class hierarchies. Finally, keep the virtual destructor rule in the back of your mind as it will become incredibly important in our upcoming units on Polymorphism. Are there any questions?