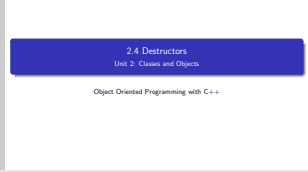
2.4 Destructors

Unit 2: Classes and Objects

Object Oriented Programming with C++



Unit 2: Classes and Objects 2.4 Destructors

- Course: DI05011021 - Object Oriented Programming with C++
- Topic: Destructors - Definition and Use
- Estimated Time: 50 minutes

Unit 2: Classes and Objects 2.4 Destructors
- Course: DI05011021 - Object Oriented Programming with C++
- Topic: Destructors - Definition and Use
- Estimated Time: 50 minutes

Welcome back everyone. Today we are continuing our deep dive into the 'Classes and Objects' unit by discussing destructors. As you know, objects take up resources when they are created, and C++ gives us a very elegant way to clean up those resources when the object is no longer needed. By the end of this session, you'll understand exactly how to manage an object's end-of-life.

Introduction to Destructors

- A destructor is a special member function of a class.
- It is executed automatically when an object goes out of scope or is explicitly deleted.
- Its primary purpose is to free up resources (like memory, file handles, or network connections) that the object may have acquired during its lifetime.
- It is the exact opposite of a constructor.

Introduction to Destructors
- A destructor is a special member function of a class.
- It is executed automatically when an object goes out of scope or is explicitly deleted.
- Its primary purpose is to free up resources (like memory, file handles, or network connections) that the object may have acquired during its lifetime.
- It is the exact opposite of a constructor.

Just as a constructor breathes life into an object by allocating resources and setting initial states, a destructor takes life away by cleaning up. In managed languages like Java or Python, the garbage collector handles memory cleanup. But in C++, you are in the driver's seat. If your object opens a file or asks for memory, it is your object's responsibility to close that file or give back that memory. The destructor is where this happens.

Syntax and Naming Convention

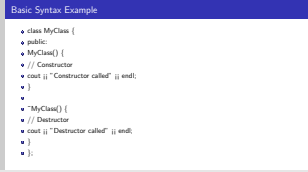
- A destructor has the exact same name as the class.
- It is preceded by a tilde symbol (~).
- It has no return type, not even void.
- It accepts NO parameters.
- A class can only have exactly ONE destructor.

Syntax and Naming Convention
- A destructor has the exact same name as the class.
- It is preceded by a tilde symbol (~).
- It has no return type, not even void.
- It accepts NO parameters.
- A class can only have exactly ONE destructor.

The syntax is strict but simple. If your class is named 'Student', the destructor is named '~Student()'. The tilde represents a bitwise NOT operation in C++, symbolizing that the destructor is the complement to the constructor. Because it takes no parameters, it cannot be overloaded. There is only one way to destroy an object.

Basic Syntax Example

- class MyClass {
- public:
- MyClass() {
- // Constructor
- cout ¡¡ "Constructor called" ¡¡ endl;
- }
-
- ~MyClass() {
- // Destructor
- cout ¡¡ "Destructor called" ¡¡ endl;
- }
- };



Here we have a simple class definition. Notice the public access specifier. While destructors can technically be private, in 99% of cases they must be public because the compiler needs to be able to call them from outside the class when an object goes out of scope. If it's private, you'll get a compile-time error when trying to instantiate the class on the stack.

When is a Destructor Called?

- 1. When a local (automatic) object goes out of its defining block or scope.
- 2. When a dynamically allocated object is explicitly deleted using the 'delete' keyword.
- 3. When a program terminates (for global or static objects).
- 4. When an enclosing object containing member objects is destroyed.

When is a Destructor Called?
- 1. When a local (automatic) object goes out of its defining block or scope.
- 2. When a dynamically allocated object is explicitly deleted using the 'delete' keyword.
- 3. When a program terminates (for global or static objects).
- 4. When an enclosing object containing member objects is destroyed.

The compiler is very diligent about this. If you create an object inside an 'if' statement, the destructor runs the moment the 'if' block ends. If you create an object globally, it is destroyed right as the main function finishes and the program exits. This deterministic behavior is one of the strongest features of C++.

Example: Local Scope Destruction

- void someFunction() {
- MyClass obj; // Constructor called here
- cout ¡¡ "Inside function" ¡¡ endl;
- } // Destructor called here automatically
-
- int main() {
- cout ¡¡ "Before function" ¡¡ endl;
- someFunction();
- cout ¡¡ "After function" ¡¡ endl;
- return 0;
- }

Example: Local Scope Destruction
- void someFunction() {
- MyClass obj; // Constructor called here
- cout ¡¡ "Inside function" ¡¡ endl;
- } // Destructor called here automatically
-
- int main() {
- cout ¡¡ "Before function" ¡¡ endl;
- someFunction();
- cout ¡¡ "After function" ¡¡ endl;
- return 0;
- }

Let's trace this code. We print 'Before function', then call 'someFunction'. Inside, 'obj' is created, so the constructor prints its message. Then 'Inside function' is printed. Right at the closing brace of 'someFunction', the lifetime of 'obj' ends. The compiler inserts a call to the destructor here. Finally, 'After function' is printed in main.

The Default Destructor

- If you do not define a destructor in your class, the C++ compiler automatically provides a default destructor.
- The default destructor is inline and has an empty body.
- It does not free dynamically allocated memory.
- It is sufficient for classes that only contain basic data types or other objects that manage their own resources.

The Default Destructor
- If you do not define a destructor in your class, the C++ compiler automatically provides a default destructor.
- The default destructor is inline and has an empty body.
- It does not free dynamically allocated memory.
- It is sufficient for classes that only contain basic data types or other objects that manage their own resources.

Much like the default constructor, the compiler helps you out if you don't write a destructor. If your class just holds a few ints and floats, the default destructor is perfectly fine. The memory for the object itself (the raw bytes holding those integers) is reclaimed from the stack automatically.

Why write a Custom Destructor?

- To prevent Memory Leaks.
- If a class uses pointers to allocate memory dynamically (using 'new'), the default destructor will NOT free that memory.
- Only the pointer variable is destroyed, not the memory it points to.
- A custom destructor must be written to call 'delete' on those pointers.

Why write a Custom Destructor?
- To prevent Memory Leaks.
- If a class uses pointers to allocate memory dynamically (using 'new'), the default destructor will NOT free that memory.
- Only the pointer variable is destroyed, not the memory it points to.
- A custom destructor must be written to call 'delete' on those pointers.

This is the most critical slide of the lecture. When you declare a pointer inside an object and allocate memory on the heap using 'new', the object owns that memory. But the compiler doesn't know that! If you rely on the default destructor, it will just destroy the pointer, leaving the heap memory orphaned. This is a memory leak.

Example: Dynamic Memory and Destructors

```
class String {
private:
char* buffer;
public:
String(const char* text) {
buffer = new char[strlen(text) + 1];
strcpy(buffer, text);
}
~String() {
delete[] buffer; // Crucial for preventing leaks!
}
};
```


Example: Dynamic Memory and Destructors
class String {
private:
char* buffer;
public:
String(const char* text) {
buffer = new char[strlen(text) + 1];
strcpy(buffer, text);
}
~String() {
delete[] buffer; // Crucial for preventing leaks!
}
};

Here we have a simple String class. The constructor dynamically allocates an array of characters based on the input text. If we didn't write the destructor shown here, every time a String object goes out of scope, we would lose that block of memory. By explicitly writing `delete[] buffer` in the destructor, we ensure a clean, leak-free system.

Destructors and 'delete' Keyword

- When an object is created on the heap using 'new', it does not go out of scope automatically.
- It remains in memory until explicitly destroyed.
- Calling 'delete' on the object pointer explicitly triggers the object's destructor.
- Then, the memory occupied by the object itself is freed.

Destructors and 'delete' Keyword
- When an object is created on the heap using 'new', it does not go out of scope automatically.
- It remains in memory until explicitly destroyed.
- Calling 'delete' on the object pointer explicitly triggers the object's destructor.
- Then, the memory occupied by the object itself is freed.

We've discussed local variables. But what if the object itself is created on the heap? For example, `String* s = new String("Hello");`. In this case, `s` is just a pointer. When `s` goes out of scope, only the pointer is destroyed. To destroy the object and trigger its destructor, you must explicitly call `delete s;`.

Example: Explicit 'delete'

- int main() {
- // Object created on the heap
- MyClass* ptr = new MyClass();
-
- cout ¡¡ "Doing some work..." ¡¡ endl;
-
- // Explicitly destroy the object
- delete ptr; // Triggers ~MyClass()
-
- return 0;
- }

Example: Explicit 'delete'
- int main() {
- // Object created on the heap
- MyClass* ptr = new MyClass();
-
- cout ¡¡ "Doing some work..." ¡¡ endl;
-
- // Explicitly destroy the object
- delete ptr; // Triggers ~MyClass()
-
- return 0;
- }

Notice how the destructor call is manually controlled here. If we omitted the 'delete ptr;' line, the program would end, the 'ptr' variable would be destroyed, but the actual 'MyClass' instance on the heap would remain, causing a leak. The destructor would never be printed or executed.

Order of Execution: Constructors vs. Destructors

- Objects are destroyed in the reverse order of their creation.
- LIFO: Last In, First Out.
- If you create Object A, then Object B, then Object C...
- ...when the scope ends, C is destroyed first, then B, then A.

Order of Execution: Constructors vs. Destructors
- Objects are destroyed in the reverse order of their creation.
- LIFO: Last In, First Out.
- If you create Object A, then Object B, then Object C...
- ...when the scope ends, C is destroyed first, then B, then A.

This is a fundamental rule in C++. The stack unwinds in reverse. Why? Because later objects might depend on earlier objects. If Object B takes a pointer to Object A, we cannot destroy A before B! By destroying them in reverse order (LIFO), C++ guarantees that an object's dependencies remain valid for its entire lifetime.

Example: Order of Execution

```
class Test {
int id;
public:
Test(int i) { id = i; cout ¡¡ "Constructed " ¡¡ id ¡¡ endl; }
~Test() { cout ¡¡ "Destroyed " ¡¡ id ¡¡ endl; }
};

int main() {
Test t1(1);
Test t2(2);
return 0;
}
```


Example: Order of Execution
class Test {
int id;
public:
Test(int i) { id = i; cout ¡¡ "Constructed " ¡¡ id ¡¡ endl; }
~Test() { cout ¡¡ "Destroyed " ¡¡ id ¡¡ endl; }
};
int main() {
Test t1(1);
Test t2(2);
return 0;
}

Can anyone guess the output? The program will output: Constructed 1, Constructed 2, Destroyed 2, Destroyed 1. It operates just like a stack. This becomes highly relevant when you have multiple objects managing complex resources like database connections or thread locks.

Best Practices and RAII

- Resource Acquisition Is Initialization (RAII)
- A core C++ programming idiom.
- Bind the lifecycle of a resource (memory, network, file) to the lifecycle of a local object.
- Let the constructor acquire the resource.
- Let the destructor release the resource.
- Guarantees cleanup even if an exception is thrown.

Best Practices and RAII
- Resource Acquisition Is Initialization (RAII)
- A core C++ programming idiom.
- Bind the lifecycle of a resource (memory, network, file) to the lifecycle of a local object.
- Let the constructor acquire the resource.
- Let the destructor release the resource.
- Guarantees cleanup even if an exception is thrown.

This slide touches on advanced but crucial concepts. RAII is what makes C++ robust. If you acquire a lock, do it in a constructor. Release it in the destructor. Because C++ guarantees destructors are called when objects go out of scope—even if an error or exception occurs—your resources are always safely cleaned up.

Summary

- Destructors are special functions called ~ClassName().
- They take no arguments and return no values.
- They are invoked automatically upon scope exit or explicitly via 'delete'.
- Crucial for cleaning up dynamically allocated memory and other resources.
- Objects are destroyed in the reverse order of their construction (LIFO).

Summary
- Destructors are special functions called ~ClassName().
- They take no arguments and return no values.
- They are invoked automatically upon scope exit or explicitly via 'delete'.
- Crucial for cleaning up dynamically allocated memory and other resources.
- Objects are destroyed in the reverse order of their construction (LIFO).

To wrap up, a destructor is your object's last will and testament. It cleans up the mess. Remember the Rule of Three (which we will cover more later): if your class needs a custom destructor to manage memory, it almost certainly needs a custom Copy Constructor and Copy Assignment Operator as well.

Questions & Answers

- Any questions regarding Destructors, Memory Management, or Scope?
- Next time: We will dive into Operator Overloading.

Questions & Answers
- Any questions regarding Destructors, Memory Management, or Scope?
- Next time: We will dive into Operator Overloading.

Thank you for your attention. Let's open the floor to any questions. Does everyone understand why the reverse order of destruction is necessary? Are there any questions on how the delete keyword interacts with heap objects?