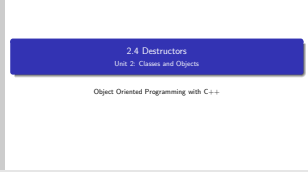
2.4 Destructors

Unit 2: Classes and Objects

Object Oriented Programming with C++



Lecture 17: Destructors

- Unit 2: Classes and Objects
- Topic: 2.4 Destructors - Definition and Use
- Course: Object Oriented Programming with C++

Lecture 17: Destructors
- Unit 2: Classes and Objects
- Topic: 2.4 Destructors - Definition and Use
- Course: Object Oriented Programming with C++

Welcome everyone to Lecture 17. Today we are concluding our discussion on the object lifecycle. We've spent a lot of time talking about how objects are born using constructors. Today, we focus on the end of an object's life: Destructors. We'll explore what they are, why they are essential, and how they help us write safe and efficient C++ code.

What is a Destructor?

- A destructor is a special member function of a class.
- It is executed automatically when an object of that class is destroyed.
- It performs 'cleanup' operations before the object's memory is reclaimed by the system.
- It is the exact counterpart to a constructor.

What is a Destructor?
- A destructor is a special member function of a class.
- It is executed automatically when an object of that class is destroyed.
- It performs 'cleanup' operations before the object's memory is reclaimed by the system.
- It is the exact counterpart to a constructor.

Just as a constructor is called automatically when an object is created, a destructor is called automatically when an object is destroyed. Think of it as a cleanup crew. When you are done using a workspace, you clean it up before leaving. A destructor does exactly that for an object. It ensures that any mess the object made—like borrowing memory or opening files—is properly cleaned up before the object ceases to exist.

Why Do We Need Destructors?

- Resource Management: Objects often acquire resources during their lifetime.
- Examples of resources:
- - Dynamically allocated memory (using 'new')
- - Open files (file handles)
- - Network connections
- - Database locks
- Failure to release these resources leads to Resource Leaks (e.g., Memory Leaks).

Why Do We Need Destructors?
- Resource Management: Objects often acquire resources during their lifetime.
- Examples of resources:
- - Dynamically allocated memory (using 'new')
- - Open files (file handles)
- - Network connections
- - Database locks
- Failure to release these resources leads to Resource Leaks (e.g., Memory Leaks).

You might wonder, if C++ automatically reclaims the memory occupied by the object's standard variables, why do we need to write a destructor? The answer is external resources. If your object simply holds an integer, the default destruction is fine. But if your object dynamically allocates memory on the heap using the 'new' keyword, or opens a file on the hard drive, C++ doesn't automatically know how to close that file or free that heap memory. If the object dies without freeing these, we get a memory leak or a locked file. Destructors give us a guaranteed place to put that cleanup code.

Syntax of a Destructor

- Name: Same name as the class, preceded by a tilde (~).
- No Return Type: Not even void.
- No Parameters: Destructors cannot take arguments.
- Example:
- class MyClass {
- public:
- ~MyClass(); // Destructor declaration
- };

Syntax of a Destructor
- Name: Same name as the class, preceded by a tilde (~).
- No Return Type: Not even void.
- No Parameters: Destructors cannot take arguments.
- Example:
- class MyClass {
- public:
- ~MyClass(); // Destructor declaration
- };

The syntax for a destructor is very strict. It must have the exact same name as the class, but prefixed with a tilde character. The tilde is a bitwise NOT operator in C++, symbolizing the opposite or complement of the constructor. Crucially, destructors do not return values, not even void. And unlike constructors, destructors cannot take any parameters. Because they take no parameters, you cannot overload a destructor. A class can only have one destructor.

Key Properties of Destructors

- 1. Automatic Invocation: Called implicitly by the compiler.
- 2. Uniqueness: A class can have only ONE destructor (no overloading).
- 3. Access Specifier: Almost always declared under 'public'.
- 4. Inheritance: Cannot be inherited, though derived classes invoke base class destructors.
- 5. Cannot have default arguments.

Key Properties of Destructors
- 1. Automatic Invocation: Called implicitly by the compiler.
- 2. Uniqueness: A class can have only ONE destructor (no overloading).
- 3. Access Specifier: Almost always declared under 'public'.
- 4. Inheritance: Cannot be inherited, though derived classes invoke base class destructors.
- 5. Cannot have default arguments.

Let's summarize the key rules. You rarely call a destructor explicitly; the compiler does it for you. There is only one destructor per class because there's only one way an object ceases to exist. They should be public; if a destructor is private, the compiler cannot destroy the object, which usually prevents you from creating the object on the stack entirely. Finally, they cannot be inherited, but in inheritance hierarchies, the derived class destructor will automatically call the base class destructor.

When is a Destructor Called?

- A destructor is invoked automatically when an object goes out of scope or is explicitly deleted.
- Scenarios:
- 1. Local Object: When the function or block containing the object ends.
- 2. Global Object: When the entire program terminates.
- 3. Dynamically Allocated Object: When the 'delete' operator is applied to its pointer.

When is a Destructor Called?
- A destructor is invoked automatically when an object goes out of scope or is explicitly deleted.
- Scenarios:
- 1. Local Object: When the function or block containing the object ends.
- 2. Global Object: When the entire program terminates.
- 3. Dynamically Allocated Object: When the 'delete' operator is applied to its pointer.

Timing is everything. When exactly does the cleanup happen? For standard local objects created inside a function, the destructor is called the moment the execution leaves that function's closing brace. If you create an object inside an 'if' block, it's destroyed when the 'if' block ends. For global objects, they are destroyed when 'main' finishes. And for objects created with 'new', their destructor is only called when you explicitly write 'delete' on their pointer. If you forget to call 'delete', the destructor is never called, and you have a memory leak.

Example 1: Basic Destructor

```
class Demo {
public:
Demo() { cout ¡¡ "Constructor called!\n"; }
~Demo() { cout ¡¡ "Destructor called!\n"; }
};

int main() {
cout ¡¡ "Inside main()\n";
{
Demo d1;
}
cout ¡¡ "Back in main()\n";
return 0;
}
```


Example 1: Basic Destructor
class Demo {
public:
Demo() { cout ¡¡ "Constructor called!\n"; }
~Demo() { cout ¡¡ "Destructor called!\n"; }
};
int main() {
cout ¡¡ "Inside main()\n";
{
Demo d1;
}
cout ¡¡ "Back in main()\n";
return 0;
}

Let's look at a simple trace. We have a class 'Demo' with a constructor and a destructor that simply print messages. In 'main', we have an inner block denoted by the curly braces. We create object 'd1' inside this inner block. Let's think about the output before we look at the next slide. 'Inside main' prints first. Then we enter the block, 'd1' is created, so the constructor prints. Then the block ends. What happens to 'd1'?

Example 1: Output Tracing

- Output:
- Inside main()
- Constructor called!
- Destructor called!
- Back in main()
-
- Explanation:
- - 'd1' is constructed when declared inside the block.
- - When the inner scope '}' is reached, 'd1' goes out of scope.
- - The destructor is automatically invoked BEFORE the program continues to the next line.

Example 1: Output Tracing
- Output:
- Inside main()
- Constructor called!
- Destructor called!
- Back in main()
-
- Explanation:
- - 'd1' is constructed when declared inside the block.
- - When the inner scope '}' is reached, 'd1' goes out of scope.
- - The destructor is automatically invoked BEFORE the program continues to the next line.

As expected, 'Inside main()' prints. Then 'Constructor called!' when 'd1' is instantiated. The moment execution hits that closing brace of the inner block, 'd1' is no longer valid. The compiler automatically inserts a call to the destructor here. So we see 'Destructor called!'. Finally, execution resumes in main, printing 'Back in main()'. This deterministic destruction is a powerful feature of C++, allowing for a concept known as Resource Acquisition Is Initialization (RAII).

The Memory Leak Problem

- If a class contains pointer members dynamically allocated with 'new', the default destructor is insufficient.
- The default destructor will destroy the pointer variable itself, but NOT the memory it points to.
- This results in 'orphaned' memory on the heap that cannot be reclaimed.
- Solution: A custom destructor utilizing the 'delete' keyword.

The Memory Leak Problem
- If a class contains pointer members dynamically allocated with 'new', the default destructor is insufficient.
- The default destructor will destroy the pointer variable itself, but NOT the memory it points to.
- This results in 'orphaned' memory on the heap that cannot be reclaimed.
- Solution: A custom destructor utilizing the 'delete' keyword.

Now let's tackle the main reason we write custom destructors. Suppose a class has a pointer, and in the constructor, we say `pointer = new int[100];`. The compiler-provided default destructor just reclaims the 4 or 8 bytes used by the pointer variable itself on the stack. It has no idea it should free the 100 integers sitting on the heap. That memory becomes a leak. The operating system cannot reclaim it until the program completely shuts down. If your program runs for a long time, this will eventually consume all RAM and crash. To fix this, we write our own destructor.

Example 2: Destructors and Dynamic Memory

- class DynamicArray {
- int* arr;
- public:
- DynamicArray(int size) {
- arr = new int[size];
- cout ¡¡ "Array allocated.\n";
- }
- ~DynamicArray() {
- delete[] arr; // Crucial cleanup step!
- cout ¡¡ "Array deallocated.\n";
- }
- };

Example 2: Destructors and Dynamic Memory
- class DynamicArray {
- int* arr;
- public:
- DynamicArray(int size) {
- arr = new int[size];
- cout ¡¡ "Array allocated.\n";
- }
- ~DynamicArray() {
- delete[] arr; // Crucial cleanup step!
- cout ¡¡ "Array deallocated.\n";
- }
- };

Here is the standard pattern for safe resource management. In the constructor of DynamicArray, we allocate an array of integers on the heap. In the destructor, we use the `delete[]` operator to free that specific memory block. Notice we use `delete[]` with brackets because we allocated an array. Whenever a DynamicArray object goes out of scope, we are 100% guaranteed that the heap memory will be cleanly freed. We don't have to manually remember to clean it up in main().

Order of Constructor and Destructor Execution

- Objects are destroyed in the exact reverse order of their creation.
- "First in, Last out" (LIFO) or Stack order.
- Why?
- Because newer objects might depend on older objects. Destroying older objects first could leave newer objects with dangling references.

Order of Constructor and Destructor Execution
- Objects are destroyed in the exact reverse order of their creation.
- "First in, Last out" (LIFO) or Stack order.
- Why?
- Because newer objects might depend on older objects. Destroying older objects first could leave newer objects with dangling references.

When multiple objects exist in the same scope, C++ has strict rules about the order of destruction. Objects are destroyed in the reverse order of their construction. Think of stacking plates. The first plate you put down is at the bottom, and it's the last one you pick up. If you have objects A, B, and C created in that order, they will be destroyed as C, then B, then A. This is crucial for safety because object C might have been created using a reference to object A. If A were destroyed first, C would be left holding a dangerous, invalid reference.

Example 3: Verifying Execution Order

```
class Test {
int id;
public:
Test(int i) { id = i; cout ¡¡ "Created " ¡¡ id ¡¡ "\n"; }
~Test() { cout ¡¡ "Destroyed " ¡¡ id ¡¡ "\n"; }
};
int main() {
Test t1(1);
Test t2(2);
return 0;
}
```


Example 3: Verifying Execution Order
class Test {
int id;
public:
Test(int i) { id = i; cout ¡¡ "Created " ¡¡ id ¡¡ "\n"; }
~Test() { cout ¡¡ "Destroyed " ¡¡ id ¡¡ "\n"; }
};
int main() {
Test t1(1);
Test t2(2);
return 0;
}

Let's prove the LIFO order. In this code, we create 't1' with ID 1, then 't2' with ID 2. When 'main' ends, both objects go out of scope simultaneously. The compiler steps in. It will print 'Created 1', then 'Created 2'. When returning, it destroys 't2' first, printing 'Destroyed 2', and finally destroys 't1', printing 'Destroyed 1'. This deterministic behavior makes reasoning about C++ programs much easier compared to languages relying on non-deterministic garbage collection.

Default Destructor

- If you do not define a destructor, the C++ compiler automatically provides a Default Destructor.
- The default destructor works perfectly for classes containing only built-in types (int, float, char) or statically sized arrays.
- It also automatically calls the destructors of any class-type member variables.
- Rule of thumb: Only write a destructor if you actually have resources to manually manage.

Default Destructor
- If you do not define a destructor, the C++ compiler automatically provides a Default Destructor.
- The default destructor works perfectly for classes containing only built-in types (int, float, char) or statically sized arrays.
- It also automatically calls the destructors of any class-type member variables.
- Rule of thumb: Only write a destructor if you actually have resources to manually manage.

It's important to note that you don't always have to write a destructor. If your class is just a collection of integers, doubles, or even other objects that manage their own resources (like std::string or std::vector), the compiler-generated default destructor is perfect. It does nothing for primitives, and correctly calls the destructors of member objects. You should only explicitly define a destructor when you have a raw pointer and need to call `delete`, or you have an open file handle and need to call `close()`.

Preview: The Rule of Three

- A foundational C++ design rule regarding resource management.
- If a class requires a user-defined destructor (e.g., to manage dynamic memory), it almost certainly requires:
- 1. A user-defined Copy Constructor
- 2. A user-defined Copy Assignment Operator
- (We will cover this in detail in upcoming lectures!)

Preview: The Rule of Three
- A foundational C++ design rule regarding resource management.
- If a class requires a user-defined destructor (e.g., to manage dynamic memory), it almost certainly requires:
- 1. A user-defined Copy Constructor
- 2. A user-defined Copy Assignment Operator
- (We will cover this in detail in upcoming lectures!)

Before we wrap up, I want to plant a seed for a future lecture. There is a famous principle in C++ called the Rule of Three. It states that if your class design forces you to write a custom destructor—meaning you are managing raw memory or resources—then you must also write your own Copy Constructor and Copy Assignment Operator. If you don't, copying your objects will result in double-free errors or memory corruption. We will explore exactly why this happens and how to fix it in our advanced memory management units.

Summary and Best Practices

- Destructors (~ClassName) are called automatically to clean up objects.
- They have no return type and take no parameters.
- Essential for preventing memory leaks when using dynamic allocation.
- Execution order is strictly the reverse of construction order.
- Best Practice: Use Smart Pointers (std::unique_ptr) instead of raw pointers to avoid writing manual destructors whenever possible.

Summary and Best Practices
- Destructors (~ClassName) are called automatically to clean up objects.
- They have no return type and take no parameters.
- Essential for preventing memory leaks when using dynamic allocation.
- Execution order is strictly the reverse of construction order.
- Best Practice: Use Smart Pointers (std::unique_ptr) instead of raw pointers to avoid writing manual destructors whenever possible.

To summarize, destructors are your cleanup crew. They are invoked automatically, take no arguments, and are essential for avoiding memory and resource leaks. They operate in reverse construction order. As a modern C++ best practice, we usually try to avoid writing manual destructors by using smart pointers or standard library containers, which manage their own memory automatically. However, understanding destructors is absolutely fundamental to understanding how C++ works under the hood.

Q&A and Next Steps

- Any questions on Destructor syntax or behavior?
- Try writing a small program testing destructor execution order.
- Next Lecture: We will introduce Operator Overloading, giving standard operators (+, -, *) new meaning for our custom classes.

Q&A and Next Steps
- Any questions on Destructor syntax or behavior?
- Try writing a small program testing destructor execution order.
- Next Lecture: We will introduce Operator Overloading, giving standard operators (+, -, *) new meaning for our custom classes.

That concludes our lecture on destructors. Are there any questions? I highly recommend you open your IDEs tonight, write a class with a constructor and destructor containing print statements, and create objects in different scopes to watch exactly when the prints happen. It's the best way to internalize this. In our next lecture, we transition to a very fun topic: Operator Overloading, which allows us to add objects together using the plus sign. See you then!