15.4 — Introduction to destructors

ALEX[1] SEPTEMBER 23, 2024

The cleanup problem

Let's say that you are writing a program that needs to send some data over a network. However, establishing a connection to the server is expensive, so you want to collect a bunch of data and then send it all at once. Such a class might be structured like this:

```
// This example won't compile because it is (intentionally) incomplete
class NetworkData
{
private:
    std::string m_serverName{};
    DataStore m_dataQueue{};

public:
    NetworkData(std::string_view serverName)
        : m_serverName { serverName }
    {
    }

    void addData(std::string_view data)
    {
        m_dataQueue.add(data);
    }

    void sendData()
    {
        // connect to server
        // send all data
        // clear data
    }
};

int main()
{
    NetworkData n("someipAddress");

    n.addData("somedata1");
    n.addData("somedata2");

    n.sendData();

    return 0;
}
```

However, this `NetworkData` has a potential issue. It relies on `sendData()` being explicitly called before the program is shut down. If the user of `NetworkData` forgets to do this, the data will not be sent to the server, and will be lost when the program exits. Now, you might say, "well, it's not hard to remember to do this!", and in this particular case, you'd be right. But consider a slightly more complex example, like this function:

```
bool someFunction()
{
    NetworkData n("someipAddress");

    n.addData("somedata1");
    n.addData("somedata2");

    if (someCondition)
        return false;

    n.sendData();
    return true;
}
```

In this case, if `someCondition` is `true`, then the function will return early, and `sendData()` will not be called. This is an easier mistake to make, because the `sendData()` call is present, the program just isn't pathing to it in all cases.

To generalize this issue, classes that use a resource (most often memory, but sometimes files, databases, network connections, etc...) often need to be explicitly sent or closed before the class object using them is destroyed. In other cases, we may want to do some record-keeping prior to the destruction of the object, such as writing information to a log file, or sending a piece of telemetry to a server. The term "clean up" is often used to refer to any set of tasks that a class must perform before an object of the class is destroyed in order to behave as expected. If we have to rely on the user of such a class to ensure that the function that performs clean up is called prior to the object being destroyed, we are likely to run into errors somewhere.

But why are we even requiring the user to ensure this? If the object is being destroyed, then we know that cleanup needs to be performed at that point. Should that cleanup happen automatically?

Destructors to the rescue

In lesson 14.9 -- Introduction to constructors (https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/)[2] we covered constructors, which are special member functions that are called when an object of a non-aggregate class type is created. Constructors are used to initialize members variables, and do any other set up tasks required to ensure objects of the class are ready for use.

Analogously, classes have another type of special member function that is called automatically when an object of a non-aggregate class type is destroyed. This function is called a **destructor**. Destructors are designed to allow a class to do any necessary clean up before an object of the class is destroyed.

Destructor naming

Like constructors, destructors have specific naming rules:

1. The destructor must have the same name as the class, preceded by a tilde (~).
2. The destructor can not take arguments.
3. The destructor has no return type.

A class can only have a single destructor.

In C++, local objects (objects defined inside a function) are destructed in the reverse order of their construction when the function (like main()) ends.

Generally you should not call a destructor explicitly (as it will be called automatically when the object is destroyed), since there are rarely cases where you'd want to clean up an object more than once.

Destructors may safely call other member functions since the object isn't destroyed until after the destructor executes.

A destructor example

```
#include <iostream>

class Simple
{
private:
    int m_id {};

public:
    Simple(int id)
        : m_id { id }
    {
        std::cout << "Constructing Simple " << m_id << '\n';
    }

    ~Simple() // here's our destructor
    {
        std::cout << "Destructing Simple " << m_id << '\n';
    }

    int getID() const { return m_id; }
};

int main()
{
    // Allocate a Simple
    Simple simple1{ 1 };
    {
        Simple simple2{ 2 };
    } // simple2 dies here

    return 0;
} // simple1 dies here
```

This program produces the following result:

```
Constructing Simple 1
Constructing Simple 2
Destructing Simple 2
Destructing Simple 1
```

Note that when each `Simple` object is destroyed, the destructor is called, which prints a message. "Destructing Simple 1" is printed after "Destructing Simple 2" because `simple2` was destroyed before the end of the function, whereas `simple1` was not destroyed until the end of `main()`.

Remember that static variables (including global variables and static local variables) are constructed at program startup and destroyed at program shutdown.

Improving the NetworkData program

Back to our example at the top of the lesson, we can remove the need for the user to explicitly call `sendData()` by having a destructor call that function:

```
class NetworkData
{
private:
    std::string m_serverName{};
    DataStore m_dataQueue{};

public:
    NetworkData(std::string_view serverName)
        : m_serverName { serverName }
    {
    }

    ~NetworkData()
    {
        sendData(); // make sure all data is sent before object is destroyed
    }

    void addData(std::string_view data)
    {
        m_dataQueue.add(data);
    }

    void sendData()
    {
        // connect to server
        // send all data
        // clear data
    }
};

int main()
{
    NetworkData n("someipAddress");

    n.addData("somedata1");
    n.addData("somedata2");

    return 0;
}
```

With such a destructor, our `NetworkData` object will always send whatever data it has before the object is destroyed! The cleanup happens automatically, which means less chance for errors, and less things to think about.

An implicit destructor

If a non-aggregate class type object has no user-declared destructor, the compiler will generate a destructor with an empty body. This destructor is called an implicit destructor, and it is effectively just a placeholder.

If your class does not need to do any cleanup on destruction, it's fine to not define a destructor at all, and let the compiler generate an implicit destructor for your class.

A warning about the `std::exit()` function

In lesson 8.12 -- Halts (exiting your program early) (https://www.learncpp.com/cpp-tutorial/halts-exiting-your-program-early/)[3], we discussed the `std::exit()` function, can be used to terminate your program immediately. When the program is terminated immediately, the program just ends. Local variables are not destroyed first, and because of this, no destructors will be called. Be wary if you're relying on your destructors to do necessary cleanup work in such a case.

For advanced readers

Unhandled exceptions will also cause the program to terminate, and may not unwind the stack before doing so. If stack unwinding does not happen, destructors will not be called prior to the termination of the program.

Next lesson
15.5 Class templates with member functions

4

Back to table of contents

5

Previous lesson
15.3 Nested types (member types)

6

7

Mr. F

Mr. F

May 30, 2025 1:04 pm PDT

ay that wasnt bad at all XD

0 Reply

Nidhi Gupta

April 15, 2025 9:34 pm PDT

Destructor does not have a return type , must have the same name as the class, and is expressed with a tilde ~

1 Reply

Leni

March 9, 2025 9:34 pm PDT

Destructors automate the cleanup process for class objects, ensuring that necessary final actions, such as releasing resources or saving data, occur before an object is destroyed. How can improper use of destructors lead to resource leaks or undefined behavior in C++ programs?

1 Reply

EmtyC

December 24, 2024 1:04 pm PST

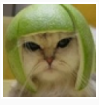
I came back here to ask: what happens when a destructor is deleted ?

```
class SomeType:
{
public:
~SomeType() = delete;
};
```

I asked chatgpt (instead of doing web research, like my peers), but he gets high on me :>

Last edited 6 months ago by EmtyC

1 Reply



Alex Author

Reply to EmtyC [11] January 1, 2025 10:19 pm PST

If the destructor is deleted, then objects of the class can't be destroyed. This includes both explicitly deleted objects and stack allocated objects.

There's very little reason to do this.

2 Reply

EmtyC

Reply to Alex [12] January 1, 2025 11:50 pm PST

tnx :>

0 Reply

EmtyC

Reply to EmtyC [11] December 28, 2024 11:15 am PST

An indirect explanation of consequences is given in chapter 25 https://www.learncpp.com/cpp-tutorial/virtual-destructors-virtual-assignment-and-overriding-virtualization/
(using inaccessible destructor through access specifiers instead)

0 Reply

k4040

November 10, 2024 6:18 am PST

Should probably add a section for explaining the usefulness of Resource Acquisition Is Initialization (RAII)

0 Reply

Alex Author

Reply to k4040 [13] November 10, 2024 10:14 pm PST

I intend to cover this in detail in the rewrite of the chapter on dynamic memory, since that's when we'll have a compelling use-case to better illustrate the principle.

3 Reply

Estelyen

October 21, 2024 2:15 am PDT

I just had a revelation. I've been programming C++ for years now, event teaching it to newcomers as part of my job. One of them just asked me why we use a tilde key to mark the destructor. Had to look that up myself. Since I rarely had to use bitwise operators before, I never figured out this connection:

Tilde key is the bitwise operator NOT.
Writing a destructor `~MyClass();` is basically like saying `if (NOT MyClass)`.

```
1 Mind = Blown
```

3 Reply

Alex Author

Reply to Estelyen [14] October 21, 2024 2:26 pm PDT

`Operator~` is also sometimes called the bitwise complement operator. One definition of "complement" is "something that completes another thing when used together". The destructor complements the constructor in managing the lifecycle of an object...

4 Reply

frank

September 19, 2024 6:44 pm PDT

what is this .add that appears in the sample program?

```
1 | m_data.add(data);
```

it doesn't seem to be defined anywhere? what is it?

also, why is this not included in the constructor?

```
1 | DataStore m_data{};
```

1 Reply

Steins;Pointer

Reply to frank [15] September 22, 2024 8:38 am PDT

```
1 | m_data.add(data);
```

in this line .add is part of the DataStore class type. Sample program isn't a complete program, but just a code snippet, and knowledge of how Datastore::add works isn't required to understand how destructors work.

DataStore m_data{}; wasn't in constructor, because it was already initialized at point of the declaration, and if user could directly assign some value to it, they could possibly create some invariant.

Last edited 9 months ago by Steins;Pointer

1 Reply

LVNA

July 27, 2024 6:38 am PDT

The UserSettings program? It a leftover from some content changed?

0 Reply

Alex Author

Reply to LVNA [16] July 27, 2024 12:14 pm PDT

It got renamed to NetworkData and I missed updating the section title. Fixed.

1 Reply

Swaminathan

July 16, 2024 11:31 pm PDT

My program and output below... My doubt is, I manually deleted a by calling `delete &a`. In that case, why would still the destructor of A be called after the block end? I expected the output to only call the destructor once when I manually delete a.

```
#include <iostream>

class A
{
public:
    ~A()
    {
        std::cout << "Destructor is called";
    }
};

int main()
{
    A a;
    delete &a;
    std::cout <<std::endl<< "before exit"<<std::endl;

}


OUTPUT:
Destructor is called
before exit
Destructor is called
```

Last edited 11 months ago by Swaminathan

0 Reply

Alex Author

Reply to Swaminathan [17] July 18, 2024 1:37 pm PDT

Because `a` is a local variable with automatic duration, and it gets destroyed at the end of the block.

You should not be manually deleting `&a` here, as you did not dynamically allocate `a`.

0 Reply

Phargelm

April 23, 2024 4:36 pm PDT

Are destructors called when a user exits a program normally? Not when we call `std::exit()` from a program but when let's say a user closes a window, or closes a console or ends a process through task manager.

0 Reply

Alex Author

Reply to Phargelm [18] April 26, 2024 10:20 am PDT

Destructors are called when the objects containing those destructors are destroyed. This typically happens when they go out of scope (if stack allocated) or when explicitly deleted (if heap allocated).

Closing a window, console, or ending a process through task manager are not normal terminations. The executable is killed at that point, and destructors are not called.

A normal termination is letting `main()` return.

2 Reply

OldCoder

Reply to Alex [19] July 12, 2024 12:22 pm PDT

As a side note, when using some UI frameworks, there may be ways to intercept abnormal program terminations.

For example, using .NET Windows Forms, you can handle the Form.Closing event. This allows you to identify and handle (IIRC) FormClosed, WindowsShuttingDown, and TaskCancelled situations, among others, including the ability to abort powering off the system.

I've used this technique in C#, but I don't know how well it works in combined managed (.NET) and unmanaged (C++) code.

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/
3. https://www.learncpp.com/cpp-tutorial/halts-exiting-your-program-early/
4. https://www.learncpp.com/cpp-tutorial/class-templates-with-member-functions/
5. https://www.learncpp.com/
6. https://www.learncpp.com/cpp-tutorial/nested-types-member-types/
7. https://www.learncpp.com/introduction-to-destructors/
8. https://www.learncpp.com/cpp-tutorial/in-and-out-parameters/
9. https://www.learncpp.com/cpp-tutorial/introduction-to-classes/
10. https://gravatar.com/

11. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-605679
12. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-606087
13. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-604009
14. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-603407
15. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-602138
16. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-600158
17. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-599684
18. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-596119
19. https://www.learncpp.com/cpp-tutorial/introduction-to-destructors/#comment-596237
20. https://g.ezoic.net/privacy/learncpp.com