

19.3 — Destructors

👤 [ALEX](#)¹ ⌚ NOVEMBER 30, 2023

A **destructor** is another special kind of class member function that is executed when an object of that class is destroyed. Whereas constructors are designed to initialize a class, destructors are designed to help clean up.

When an object goes out of scope normally, or a dynamically allocated object is explicitly deleted using the `delete` keyword, the class destructor is automatically called (if it exists) to do any necessary clean up before the object is removed from memory. For simple classes (those that just initialize the values of normal member variables), a destructor is not needed because C++ will automatically clean up the memory for you.

However, if your class object is holding any resources (e.g. dynamic memory, or a file or database handle), or if you need to do any kind of maintenance before the object is destroyed, the destructor is the perfect place to do so, as it is typically the last thing to happen before the object 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.

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

A destructor example

Let's take a look at a simple class that uses a destructor:

```

1  #include <iostream>
2  #include <cassert>
3  #include <cstddef>
4
5  class IntArray
6  {
7  private:
8      int* m_array{};
9      int m_length{};
10
11 public:
12     IntArray(int length) // constructor
13     {
14         assert(length > 0);
15
16         m_array = new int[static_cast<std::size_t>(length)]{};
17         m_length = length;
18     }
19
20     ~IntArray() // destructor
21     {
22         // Dynamically delete the array we allocated earlier
23         delete[] m_array;
24     }
25
26     void setValue(int index, int value) { m_array[index] = value; }
27     int getValue(int index) { return m_array[index]; }
28
29     int getLength() { return m_length; }
30 };
31
32 int main()
33 {
34     IntArray ar ( 10 ); // allocate 10 integers
35     for (int count{ 0 }; count < ar.getLength(); ++count)
36         ar.setValue(count, count+1);
37
38     std::cout << "The value of element 5 is: " << ar.getValue(5) << '\n';
39
40     return 0;
41 } // ar is destroyed here, so the ~IntArray() destructor function is called here

```

Tip

If you compile the above example and get the following error:

```

error: 'class IntArray' has pointer data members [-Werror=effc++] |
error:   but does not override 'IntArray(const IntArray&)' [-Werror=effc++] |
error:   or 'operator=(const IntArray&)' [-Werror=effc++] |

```

Then you can either remove the “-Weffc++” flag from your compile settings for this example, or you can add the following two lines to the class:

```

1  IntArray(const IntArray&) = delete;
2  IntArray& operator=(const IntArray&) = delete;

```

We discuss `=delete` for members in lesson [14.14 -- Introduction to the copy constructor](https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/) (<https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/>)²

This program produces the result:

```
The value of element 5 is: 6
```

On the first line of `main()`, we instantiate a new `IntArray` class object called `ar`, and pass in a length of 10. This calls the constructor, which dynamically allocates memory for the array member. We must use dynamic allocation here because we do not know at compile time what the length of the array is (the caller decides that).

At the end of `main()`, `ar` goes out of scope. This causes the `~IntArray()` destructor to be called, which deletes the array that we allocated in the constructor!

A reminder

In lesson [16.2 -- Introduction to `std::vector` and list constructors](https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/)³, we note that parentheses based initialization should be used when initializing an array/container/list class with a length (as opposed to a list of elements). For this reason, we initialize `IntArray` using `IntArray ar (10);`.

Constructor and destructor timing

As mentioned previously, the constructor is called when an object is created, and the destructor is called when an object is destroyed. In the following example, we use `cout` statements inside the constructor and destructor to show this:

```

1  #include <iostream>
2
3  class Simple
4  {
5  private:
6      int m_nID{};
7
8  public:
9      Simple(int nID)
10         : m_nID{ nID }
11     {
12         std::cout << "Constructing Simple " << nID << '\n';
13     }
14
15     ~Simple()
16     {
17         std::cout << "Destructing Simple" << m_nID << '\n';
18     }
19
20     int getID() { return m_nID; }
21 };
22
23 int main()
24 {
25     // Allocate a Simple on the stack
26     Simple simple{ 1 };
27     std::cout << simple.getID() << '\n';
28
29     // Allocate a Simple dynamically
30     Simple* pSimple{ new Simple{ 2 } };
31
32     std::cout << pSimple->getID() << '\n';
33
34     // We allocated pSimple dynamically, so we have to delete it.
35     delete pSimple;
36
37     return 0;
38 } // simple goes out of scope here

```

This program produces the following result:

```

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

```

Note that “Simple 1” is destroyed after “Simple 2” because we deleted pSimple before the end of the function, whereas simple was not destroyed until the end of main().

Global variables are constructed before main() and destroyed after main().

RAII

RAII (Resource Acquisition Is Initialization) is a programming technique whereby resource use is tied to the lifetime of objects with automatic duration (e.g. non-dynamically allocated objects). In C++, RAII is implemented via classes with constructors and destructors. A resource (such as memory, a file or database handle, etc...) is typically acquired in the object’s constructor (though it can be acquired after the object is created if that makes sense). That resource can then be used while the object is alive. The resource is

released in the destructor, when the object is destroyed. The primary advantage of RAI is that it helps prevent resource leaks (e.g. memory not being deallocated) as all resource-holding objects are cleaned up automatically.


The IntArray class at the top of this lesson is an example of a class that implements RAI -- allocation in the constructor, deallocation in the destructor. std::string and std::vector are examples of classes in the standard library that follow RAI -- dynamic memory is acquired on initialization, and cleaned up automatically on destruction.

A warning about the std::exit() function

Note that if you use the std::exit() function, your program will terminate and no destructors will be called. Be wary if you're relying on your destructors to do necessary cleanup work (e.g. write something to a log file or database before exiting).

Summary

As you can see, when constructors and destructors are used together, your classes can initialize and clean up after themselves without the programmer having to do any special work! This reduces the probability of making an error, and makes classes easier to use.



Next lesson


19.4 [Pointers to pointers and dynamic multidimensional arrays](#)

4



[Back to table of contents](#)

5




Previous lesson

19.2 [Dynamically allocating arrays](#)

6

7



B

U


URL



INLINE CODE


C++ CODE BLOCK


HELP!

Leave a comment...

 Name*

 Email* 



Notify me about replies: 

POST COMMENT

🔍 Find a mistake? Leave a comment above!🔗

👤 Avatars from <https://gravatar.com/>¹⁰ are connected to your provided email address.

289 COMMENTS

Newest ▼



Bassem

🕒 April 23, 2025 12:04 pm PDT

small prog to help wrap up all the info


```

1  #include <bits/stdc++.h>
2  using namespace std;
3
4
5  class IntArray{
6  public:
7      //default constructor
8      IntArray() : arr(nullptr), s(0){
9          cout << "Default constructor called" << endl;
10     }
11
12     // constructor with the size
13     IntArray(size_t sz) : s(sz){
14         cout << "size constructor called" << endl;
15         arr = new int[sz]{};
16     }
17
18     // copy constructor
19     IntArray(const IntArray& other) : s(other.s){
20         cout << "copy constructor called" << endl;
21         arr = new int[s];
22         for(size_t i = 0; i < s; ++i){
23             arr[i] = other.arr[i];
24         }
25     }
26
27     // initializer list constructor
28     IntArray(const initializer_list<int>& l) : s(l.size()){
29         cout << "list constructor called" << endl;
30         size_t i = 0;
31         for(const auto e : l){
32             arr[i] = e;
33             ++i;
34         }
35     }
36     // assignment operator
37     const IntArray& operator=(const IntArray& other){
38         cout << "assignment operator called" << endl;
39         if(s != other.s){
40             delete[] arr;
41             s = other.s;
42             arr = new int[s];
43         }
44
45         for(size_t i = 0; i < s; ++i){
46             arr[i] = other.arr[i];
47         }
48         return other;
49     }
50
51     // subscript operator
52     int& operator[](size_t i){
53         return arr[i];
54     }
55
56     void print(){
57         int* b = arr;
58         int* e = arr + s;
59         cout << "Array : ";
60         for(; b != e; ++b){
61             cout << *b << " ";
62         }
63         cout << endl;
64     }
65
66     // destructor
67     ~IntArray(){
68         cout << "destructor called" << endl;
69         delete[] arr;
70     }

```



```

71
72 private:
73     int* arr;
74     size_t s;
75 };
76
77
78 int main(){
79
80     // Array of 4 IntArray
81     IntArray a[4]{};
82     cout << endl;
83     // default constructor will be called
84     // now i have an array of 4 IntArray each is default constructed
85
86     // constructor with size then assignment operator
87     a[0] = IntArray(4);
88     for(size_t i = 0; i < 4; ++i){
89         a[0][i] = static_cast<int>(i + 1);
90     }
91     cout << endl;
92
93     // assinment operator
94     a[1] = a[0];
95     cout << endl;
96
97     // initializer_list constructor then assignment operator
98     a[2] = {2, 3, 4, 5};
99     cout << endl;
100
101     // initializer_list constructor then assignment operator
102     IntArray c{2, 3, 4, 5};
103     a[3] = c;
104     cout << endl;
105
106     for(size_t i = 0; i < 4; ++i){
107         a[i].print();
108     }
109     cout << "\n\n\n";
110
111
112     // i wanna have a dynamically allocated array of IntArray
113     IntArray* p = new IntArray[4];
114     cout << endl;
115     // default constructor will be called
116     // now i have an array of 4 IntArray each is default constructed
117
118     // constructor with size then assignment operator
119     p[0] = IntArray(4);
120     for(size_t i = 0; i < 4; ++i){
121         p[0][i] = static_cast<int>(i + 1);
122     }
123     cout << endl;
124
125     // assinment operator
126     p[1] = p[0];
127     cout << endl;
128
129     // initializer_list constructor then assignment operator
130     p[2] = {2, 3, 4, 5};
131     cout << endl;
132
133     // assignment operator
134     p[3] = c;
135     cout << endl;
136
137     for(size_t i = 0; i < 4; ++i){
138         p[i].print();
139     }
140     // manually deallocating the dynamically allocated array
141     delete[] p;

```

```
141 delete[] p;  
142 // automatically will destruct the first array (RAII)  
143  
144 }
```



Nidhi Gupta

🕒 April 15, 2025 9:40 pm PDT

Destructor is called when an object is destroyed. here is example:

```
~IntArray() // destructor  
{  
    // Dynamically delete the array we allocated earlier  
    delete[] m_array;  
}
```

👍 0 ➡ Reply



JiaChen

🕒 March 8, 2025 6:02 am PST

What is the destructor call order in relation to the return statement?

```
    return 0;  
} // ar is destroyed here, so the ~IntArray() destructor function is called here
```

👍 0 ➡ Reply



JiaChen

🗨 Reply to [JiaChen](#) ¹¹ 🕒 March 8, 2025 6:17 am PST

AI just provided with this code for demonstration. Very clear.

```
1  #include <iostream>  
2  
3  struct Monitor {  
4      int& ref;  
5      Monitor(int& r) : ref(r) {}  
6      ~Monitor() { ref *= 2; }  
7  };  
8  
9  int function() {  
10     int value = 5;  
11     Monitor m(value);  
12     return value; // Returns 5, then destructor modifies value  
13 }  
14  
15 int main() {  
16     std::cout << function() << std::endl; // Outputs 5  
17     return 0;  
18 }
```

👍 1 ➡ Reply



Timon

🕒 March 22, 2024 3:20 am PDT

c-arrays don't need to be indexed with `size_t` right?

```
m_array = new int[static_cast<std::size_t>(length)]{};
```

('line 16' first code bit)

👍 1 ➡ Reply



Alex

Author

➡ Reply to [Timon](#)¹² 🕒 March 22, 2024 4:22 pm PDT

They don't need to be indexed with a `std::size_t`, but as of C++14 they need to be created with one.

👍 3 ➡ Reply



iosefka

🕒 November 13, 2023 10:48 am PST

Just to test it and confirm, I ran the last example with the `delete pSimple;` line commented out and indeed it only said Destructing Simple 1. So does this memory get freed up when the program closes? Or only when Visual Studio closes? Or on reboot? Or do other programs just not know that something is using that memory and other programs can still use it? Sorry if this was already explained and I missed it, but I guess I'm still a bit uncertain about memory leaks.

👍 3 ➡ Reply



Alex

Author

➡ Reply to [iosefka](#)¹³ 🕒 November 14, 2023 9:19 pm PST

Modern OSes will generally clean up any memory leaked by the application when the application shuts down.

👍 4 ➡ Reply



iosefka

➡ Reply to [Alex](#)¹⁴ 🕒 December 4, 2023 9:25 am PST

OK perfect, thanks!

👍 1 ➡ Reply



Helix

🕒 October 12, 2023 11:13 pm PDT

Is default destructor too, like the default constructor, equivalent to `~Destructor(){}` ? If not, how does the default destructor, say for the `class Simple`, look like?

👍 0 ➡ Reply



Alex Author

↻ Reply to [Helix](#)¹⁵ ⌚ October 14, 2023 1:29 pm PDT

Yes, an implicitly generated default destructor has an empty body.

✎ *Last edited 1 year ago by Alex*

👍 0 ➡ Reply

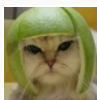


Helix

↻ Reply to [Alex](#)¹⁶ ⌚ October 14, 2023 9:14 pm PDT

You mean destructor? I was asking about destructors.

👍 0 ➡ Reply

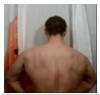


Alex Author

↻ Reply to [Helix](#)¹⁷ ⌚ October 17, 2023 7:18 pm PDT

Yes, sorry, I meant destructor. Prior comment updated.

👍 0 ➡ Reply



Timo

⌚ September 10, 2023 10:39 am PDT

```

1  #include <iostream>
2  #include <cassert>
3  #include <cstdint>
4
5  class IntArray
6  {
7  private:
8      int* m_array{};
9      int m_length{};
10
11 public:
12     IntArray(int length) // constructor
13     {
14         assert(length > 0);
15
16         m_array = new int[static_cast<std::size_t>(length)]{};
17         m_length = length;
18     }
19
20     ~IntArray() // destructor
21     {
22         // Dynamically delete the array we allocated earlier
23         delete[] m_array;
24     }
25
26     void setValue(int index, int value) { m_array[index] = value; }
27     int getValue(int index) { return m_array[index]; }
28
29     int getLength() { return m_length; }
30 };
31
32 int main()
33 {
34     IntArray ar ( 10 ); // allocate 10 integers
35     for (int count{ 0 }; count < ar.getLength(); ++count)
36         ar.setValue(count, count+1);
37
38     std::cout << "The value of element 5 is: " << ar.getValue(5) << '\n';
39
40     return 0;
41 } // ar is destroyed here, so the ~IntArray() destructor function is called here

```

@line34: Why did you use direct list initialization instead of brace here?

EDIT: For those that are searching for the answer. Scroll down while reading XD

 Last edited 1 year ago by Timo

 1  Reply



code creator

🕒 August 16, 2023 9:52 pm PDT

@Alex and everyone else

What if we have a code in which we have in which we have static, const, global, static const constructors and their respective destructors, I asked this from chat GPT and it gave me the following

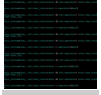
Overall, the order of execution would be:

Constructors for global variables

Const static constructors

Static constructors
Non-static non-const constructors
Const constructors
main() function executes
Destructors for global variables
Non-static non-const destructors
Const destructors
Static destructors
Const static destructors

 0  Reply



learnccp lesson reviewer


 July 17, 2023 11:15 am PDT

Any suggested exercises?

 0  Reply



Suku_go

 July 6, 2023 2:11 am PDT

What does Allocate a Simple dynamically mean in this case ? I can understand dynamic allocate array, what does it mean when we want to dynamically allocate an Instance object ?

```

1  #include <iostream>
2
3  class Simple
4  {
5  private:
6      int m_nID{};
7
8  public:
9      Simple(int nID)
10         : m_nID{ nID }
11     {
12         std::cout << "Constructing Simple " << nID << '\n';
13     }
14
15     ~Simple()
16     {
17         std::cout << "Destructing Simple" << m_nID << '\n';
18     }
19
20     int getID() { return m_nID; }
21 };
22
23 int main()
24 {
25     // Allocate a Simple on the stack
26     Simple simple{ 1 };
27     std::cout << simple.getID() << '\n';
28
29     // Allocate a Simple dynamically
30     Simple* pSimple{ new Simple{ 2 } };
31
32     std::cout << pSimple->getID() << '\n';
33
34     // We allocated pSimple dynamically, so we have to delete it.
35     delete pSimple;
36
37     return 0;
38 } // simple goes out of scope here

```

 Last edited 2 years ago by Suku_go

 0  Reply



Alex Author

 Reply to [Suku_go](#)¹⁸  July 7, 2023 3:21 pm PDT

It means heap memory is allocated for a single instance of a Simple object.

 0  Reply

Links

1. <https://www.learncpp.com/author/Alex/>

2. <https://www.learncpp.com/cpp-tutorial/introduction-to-the-copy-constructor/>
3. <https://www.learncpp.com/cpp-tutorial/introduction-to-stdvector-and-list-constructors/>
4. <https://www.learncpp.com/cpp-tutorial/pointers-to-pointers/>
5. <https://www.learncpp.com/>
6. <https://www.learncpp.com/cpp-tutorial/dynamically-allocating-arrays/>
7. <https://www.learncpp.com/destructors/>
8. <https://www.learncpp.com/cpp-tutorial/introduction-to-constructors/>
9. <https://www.learncpp.com/cpp-tutorial/the-hidden-this-pointer-and-member-function-chaining/>
10. <https://gravatar.com/>
11. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-608356>
12. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-594967>
13. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-589716>
14. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-589802>
15. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-588602>
16. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-588811>
17. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-588831>
18. <https://www.learncpp.com/cpp-tutorial/destructors/#comment-583277>
19. <https://g.ezoic.net/privacy/learncpp.com>