8.7 — Destructors

BY ALEX ON SEPTEMBER 6TH, 2007 | LAST MODIFIED BY ALEX ON JANUARY 23RD, 2020

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.

Note that rule 2 implies that only one destructor may exist per class, as there is no way to overload destructors since they can not be differentiated from each other based on arguments.

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
4  class IntArray
5  {
6  private:
7      int *m_array;
8      int m_length;
9
10 public:
11     IntArray(int length) // constructor
12     {
13         assert(length > 0);
14
15         m_array = new int[length]{};
16         m_length = length;
17     }
18
19     ~IntArray() // destructor
20     {
21         // Dynamically delete the array we allocated earlier
22         delete[] m_array;
23     }
24
```

```
    void setValue(int index, int value) { m_array[index] = value; }
    int getValue(int index) { return m_array[index]; }

    int getLength() { return m_length; }
};

int main()
{
    IntArray ar(10); // allocate 10 integers
    for (int count{ 0 }; count < ar.getLength(); ++count)
        ar.setValue(count, count+1);

    std::cout << "The value of element 5 is: " << ar.getValue(5) << '\n';

    return 0;
} // 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:

```
    IntArray(const IntArray&) = delete;
    IntArray& operator=(const IntArray&) = delete;
```

We'll discuss what these do in chapter 9.

This program produces the result:

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

On the first line, 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!

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:

```
class Simple
{
private:
    int m_nID;

public:
    Simple(int nID)
```

```
        : m_nID{ nID }
    {
        std::cout << "Constructing Simple " << nID << '\n';
    }

    ~Simple()
    {
        std::cout << "Destructing Simple" << m_nID << '\n';
    }

    int getID() { return m_nID; }
};

int main()
{
    // Allocate a Simple on the stack
    Simple simple{ 1 };
    std::cout << simple.getID() << '\n';

    // Allocate a Simple dynamically
    Simple *pSimple{ new Simple{ 2 } };

    std::cout << pSimple->getID() << '\n';

    // We allocated pSimple dynamically, so we have to delete it.
    delete pSimple;

    return 0;
} // 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 RAII is that it helps prevent resource leaks (e.g. memory not being deallocated) as all resource-holding objects are cleaned up automatically.

Under the RAII paradigm, objects holding resources should not be dynamically allocated. This is because destructors are only called when an object is destroyed. For objects allocated on the stack, this happens automatically when the object goes out of scope, so there’s no need to worry about a resource eventually getting cleaned up. However, for dynamically allocated objects, the user is responsible for deletion -- if the user forgets to

do that, then the destructor will not be called, and the memory for both the class object and the resource being managed will be leaked!

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

Rule: If your class dynamically allocates memory, use the RAII paradigm, and don't allocate objects of your class dynamically

A warning about the exit() function

Note that if you use the 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.

 **8.8 -- The hidden "this" pointer**

 **Index**

 **8.6 -- Overlapping and delegating constructors**

C++ TUTORIAL | PRINT THIS POST

181 comments to 8.7 — Destructors

« Older Comments 1 2 3

hellmet
October 27, 2019 at 10:38 am · Reply

A small suggestion, I think it would be great to mention that explicitly not calling delete on line 30 will not free up the resources in Simple(2). The pointer goes out of scope and deleted, but the data is still on the heap. I understand this was already covered in the 'Stack and Heap' and later chapters, but perhaps this detail might slip newbies (like me). Or add it as a quiz question!

nascardriver
October 28, 2019 at 3:43 am · Reply

Thanks for the suggestion!
I added a comment stating that we have to delete it. I don't think a re-explanation of what happens when you don't delete it is necessary.

Anthony
October 9, 2019 at 6:49 am · Reply

I'm having a problem with some C++ code, which necessarily uses on the C sockets API, so my first question is whether it's ok to use

```
1 new
```

and

```
1 delete
```

with the buffers and structures associated with sockets in general. I'm pretty sure the answer is yes to this one.

Secondly, I believe the problem is to do with the way destruction is taking place. Let's say that

```
1 Class A
```

contains (as a member variable on the stack) a pointer to

```
1 Class B
```

. What happens to this pointer when Class A goes out of scope? Does Class A destruct (along with the pointer), leaving Class B intact?

Thanks :)

nascardriver
October 9, 2019 at 6:59 am · Reply

> it's ok to use [new]
Yes

> it's ok to use [delete]
To delete objects that were created with `new`, yes. You just can't `delete` objects that were created with `malloc`, but you shouldn't have to anyway.

> Secondly [...]
If `A` is only storing a pointer to `B`, then nothing will happen to the `B` object `A` was pointing to (Unless `A`'s user-defined destructor does something to `B`).
If `A` created the `B`, then `A` should `delete` the `B`.

If `A` only stores the pointer, but the `B` is owned by something else (ie. something else created it), then you don't have to do anything, the `B` will stay intact.

Anthony
October 9, 2019 at 7:18 am · Reply

Thank you.

Anthony
October 10, 2019 at 4:08 am · Reply

Quick follow-up as I'm still trying to see where I've gone wrong:

If I have a pointer pointing at some dynamically allocated memory, and then I copy the base address to another pointer (with a different name, obviously, but of the same type), can I use this pointer to deallocate the memory?

nascardriver
October 10, 2019 at 6:14 am · Reply

Yes. Keep in mind that you're deleting the pointed-to memory, not the pointer. Only use `delete` on one of the pointers.

```
1 int* a{ new int{} };
2 int* b{ a };
3
4 delete b;
5 delete a; // Undefined behavior, that memory has already been deleted.
6 // Delete either @a or @b, not both.
```

Anthony
October 10, 2019 at 8:37 am · Reply

Thanks. I've squashed the bug. That one aged me. Mind, they all do.

It brings up something else. I was dealing with a @struct msghdr from the C sockets API. It has a member @msg_name of type @void*, which contains a socket address (@sockaddr_in, @sockaddr_in6, etc) depending on how you cast it. I've written an init function which uses @calloc() to fill out all the pointer members of the @msghdr struct and will do a @new version of this shortly. If I do this to allocate:

```
1 struct msghdr msg;
2 msg.msg_name = (struct sockaddr_storage*)calloc(1, sizeof(struct sockc
```

Would I do the following to deallocate - i.e. I have to cast?

```
1 free((struct sockaddr_storage*)(msgh_.msg_name));
```

More generally, when using the C sockets API in C++, is it just a matter of writing C++ wrappers for the parts of it one needs, because that's what I'm doing?

nascardriver
October 11, 2019 at 12:41 am · Reply

> uses @calloc() to fill out all the pointer members
IIRC you don't have to allocate them, instead point them to regular variables. Check the documentation or tutorials to make sure you're doing it right. I could very well be wrong, this isn't something I've done often.
Any reason why you're allocating an array with 1 element? That seems odd to me.

> Would I do the following to deallocate - i.e. I have to cast?
No, you don't have to cast.

> is it just a matter of writing C++ wrappers
It depends on how close to the api you have to be. Most likely, you don't care which sockets api your program is using, so you'd write an abstraction that makes using the sockets very easy, ie. the user (of your code) won't know which sockets api you're using and they don't have to know how it works.
In the unlikely case that you need to use the socket's function all over the place, you can wrap the individual functions (maybe small groups of functions) so that you don't have to worry about memory management and such and wrap everything in a class so that you don't have to pass the socket's handle around.

C++23 might add networking capabilities, but that's a long way to go.
There are many networking libraries for C++. Unless you're doing networking for fun, those libraries should be used (I can't recommend any, I always did networking for fun :)).

Anthony
October 20, 2019 at 7:15 am · Reply

I had to take a few days to reflect on your advice. The trickiest task has been to decide whether to allocate 'fairly complicated' socket-related structures like @sockaddr_storage, @addrinfo and @msghdr on the stack or the heap, and how to do so when it comes to the heap. I see that they're not like C++ classes because they don't have overloaded assignment operators, so creating copies (and it turns out I really do need to do so) isn't as straightforward as it could be.

I'm pretty sure I have it down, but there is one thing I don't know how to do, which is to do with void pointers. So, if I take the @msghdr structure as an example (https://linux.die.net/man/2/recvmsg). It has a member called @void *msg_control, but in practice, it points to an array of @struct cmsghdr's. Accordingly, I'm trying to allocate memory and perform the copy like this:

```
1  void copy_msghdr(msghdr* dest, msghdr *src) {
2      ...
3      dest->msg_control = (struct cmsghdr*)malloc(src->msg_con
4      dest->msg_controllen = src->msg_controllen;
5      for (int i{}; i < src->msg_controllen; ++i) {
6          dest->msg_control[i].cmsg_data = (char*)malloc(src->
7          memcpy(dest->msg_control[i].cmsg_data, src->msg_cont
8          dest->msg_control[i].cmsg_len = src->msg_control[i].
9          ...
10     }
11     ...
12 }
```

However, the compiler complains that I'm performing arithmetic on a void pointer, which of course I am through my use of array index operators. My question is, how do I use casting to solve this problem?

nascardriver
October 20, 2019 at 8:04 am · Reply

You don't need to copy the data at all, unless you want to store a copy of course, but I don't see why you would do that.

`msg_control` is a `void*`. As your compiler says, you can't do arithmetic with it (Array indexing is arithmetic).
If you're sure that `msg_control` is a `cmsghdr` array, you can cast it.
You're using C-style casts. They allow unsafe casts. Use C++ casts.

```
1 auto msg_source{ static_cast<cmsghdr*>(src->msg_control[i
2 auto msg_destination{ static_cast<cmsghdr*>(src->msg_cont
3
4 // You can use the new variables as arrays.
5
6 // See doc of msg_controllen.
7 std::copy_n(msg_source, src->msg_controllen, msg_destinat
```

Please see "Ancillary data should only be accessed by the macros defined in cmsg(3)." on the page you linked. Your code might be unsafe.
The page also says that `msg_controllen` is the length of the buffer. You're using it as the array's length (ie. number of elements). It might just be worded poorly in the doc.
My code assumes that your code is correct. I didn't check to documentation.

Anthony
October 20, 2019 at 4:39 pm

Yes, I'm not certain either about the poorly worded documentation. I think it doesn't really make sense for `msg_controllen` to be the entire length of all the ancilliary data, because the length of each `cmsghdr` buffer is given individually by the `msg_len` member. But then.. I'm not sure. Stack Overflow beckons.

Yes, I'm using the macros to access the ancilliary data. But now you make me think. If I make a byte for byte copy, shouldn't they still work? Still, I shall try, aware that it may all be for nothing.

The reason I want to store a copy is that I have a Server class that can make many multiple connections with clients, and I want to keep track of not only the socket file descriptor for each of them, but the remote sockaddr and the last msghdr struct received from or sent to the client, too. Maybe a poor design decision, but I'm learning a lot.

Thank you for your help, as always.

nascardriver
October 21, 2019 at 3:23 am · Reply

> I'm not certain either about the poorly worded documentation
Look at the value of `msg_controllen` when you receive a message. That should tell you what it is.

> shouldn't they [macros] still work?
It might even work entirely without them, but if the documentation tells you to use those macros, then you use those macros. It might work without them today or in the current scenario, but the developers of the library are free the make your macro-less code break in an update. This is just like undefined behavior. It might work, it might not, better do it the right way.

> I want to store a copy
I suggest you to get rid of the C data as soon as possible unless you process it right away. If you use the C data, you'll have to worry about memory, which can be difficult as you've already noticed.
You can use valgrind's memcheck to see if you're leaking memory, which can happen quite easily.

```
1 valgrind --tool-memcheck ./your_binary
```

Anthony
October 22, 2019 at 12:46 pm

I have tried a stack-allocated 'msghdr' struct too, as a member variable. I'm wondering about it, though. If I have a struct like this, i.e. one that contains member pointers, I can of course allocate memory for these on the stack as well. However, I then have a bunch of member variables, when I really only wanted one (the original 'msghdr' struct), and things get rather messy. How would I do this and keep everything tidy?

nascardriver
October 23, 2019 at 2:44 am · Reply

I'm replying here, because there are too many messages.
Don't mix regular members and pointer members, that will make your code ugly and harder to use. Since you want to use the original struct and that struct has pointer members, do everything with pointers.

Parsa
October 3, 2019 at 7:29 am · Reply

A pointer to a class will point to the first initialized member variable? Or how does it work.

nascardriver
October 3, 2019 at 7:38 am · Reply

It points to where the instance starts in memory. There may or may not be a member at the address, classes can store more than members.

Parsa
October 3, 2019 at 11:21 am · Reply

What do you mean by instance?

Parsa
October 3, 2019 at 5:34 pm · Reply

So it does point to anything in specific, just the object?

nascardriver
October 4, 2019 at 1:33 am · Reply

An instance is a variable of the class type.

```
1 // parsa is an instance of @C
2 C parsa{};
```

Every instance takes up some memory. A pointer to the instance points to the first byte that the instance occupies. For now, if all the class' members have the same visibility (eg. all public), a pointer to the instance points to the same address as a pointer to the first member of the class. There are more restrictions for this to be true, but you don't know about them yet.

Parsa
October 4, 2019 at 2:22 am · Reply

Alright, thanks.

« Older Comments 1 2 3