8.7 — Destructors

BY ALEX ON SEPTEMBER 6TH, 2007 | LAST MODIFIED BY ALEX ON MAY 20TH, 2018

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:

```
#include <iostream>
#include <cassert>

class IntArray
{
private:
    int *m_array;
    int m_length;

public:
    IntArray(int length) // constructor
    {
        assert(length > 0);

        m_array = new int[length];
        m_length = length;
    }

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

    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 < 10; ++count)
        ar.setValue(count, count+1);

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

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

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

    ~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';
    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

Share this: