15.4 — Introduction to destructors

learncpp.com/cpp-tutorial/introduction-to-destructors/

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:

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

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

        void addData(std::string_view data)
        {
                m_data.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 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.

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 UserSettings 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_data{};

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_data.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), 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.