25.4 — Virtual destructors, virtual assignment, and overriding virtualization

ALEX SEPTEMBER 11, 2023

Virtual destructors

Although C++ provides a default destructor for your classes if you do not provide one yourself, it is sometimes the case that you will want to provide your own destructor (particularly if the class needs to deallocate memory). You should **always** make your destructors virtual if you're dealing with inheritance. Consider the following example:

```
#include <iostream>
class Base
{
public:
    ~Base() // note: not virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    ~Derived() // note: not virtual (your compiler may warn you about this)
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Note: If you compile the above example, your compiler may warn you about the non-virtual destructor (which is intentional for this example). You may need to disable the compiler flag that treats warnings as errors to proceed.

Because base is a Base pointer, when base is deleted, the program looks to see if the Base destructor is virtual. It's not, so it assumes it only needs to call the Base destructor. We can see this in the fact that the above example prints:

```
Calling ~Base()
```

However, we really want the delete function to call Derived's destructor (which will call Base's destructor in turn), otherwise m_array will not be deleted. We do this by making Base's destructor virtual:

```
#include <iostream>
class Base
{
public:
    virtual ~Base() // note: virtual
    {
        std::cout << "Calling ~Base()\n";
    }
};

class Derived: public Base
{
private:
    int* m_array {};

public:
    Derived(int length)
      : m_array{ new int[length] }
    {
    }

    virtual ~Derived() // note: virtual
    {
        std::cout << "Calling ~Derived()\n";
        delete[] m_array;
    }
};

int main()
{
    Derived* derived { new Derived(5) };
    Base* base { derived };

    delete base;

    return 0;
}
```

Now this program produces the following result:

```
Calling ~Derived()
Calling ~Base()
```

Rule

Whenever you are dealing with inheritance, you should make any explicit destructors virtual.

As with normal virtual member functions, if a base class function is virtual, all derived overrides will be considered virtual regardless of whether they are specified as such. It is not necessary to create an empty derived class destructor just to mark it as virtual.

Note that if you want your base class to have a virtual destructor that is otherwise empty, you can define your destructor this way:

```
    virtual ~Base() = default; // generate a virtual default destructor
```

Virtual assignment

It is possible to make the assignment operator virtual. However, unlike the destructor case where virtualization is always a good idea, virtualizing the assignment operator really opens up a bag full of worms and gets into some advanced topics outside of the scope of this tutorial. Consequently, we are going to recommend you leave your assignments non-virtual for now, in the interest of simplicity.

Ignoring virtualization

Very rarely you may want to ignore the virtualization of a function. For example, consider the following code:

```
#include <string_view>
class Base
{
public:
    virtual ~Base() = default;
    virtual std::string_view getName() const { return "Base"; }
};

class Derived: public Base
{
public:
    virtual std::string_view getName() const { return "Derived"; }
};
```

There may be cases where you want a Base pointer to a Derived object to call Base::getName() instead of Derived::getName(). To do so, simply use the scope resolution operator:

```
#include <iostream>
int main()
{
    Derived derived {};
    const Base& base { derived };

    // Calls Base::getName() instead of the virtualized Derived::getName()
    std::cout << base.Base::getName() << '\n';

    return 0;
}
```

You probably won't use this very often, but it's good to know it's at least possible.

Should we make all destructors virtual?

This is a common question asked by new programmers. As noted in the top example, if the base class destructor isn't marked as virtual, then the program is at risk for leaking memory if a programmer later deletes a base class pointer that is pointing to a derived object. One way to avoid this is to mark all your destructors as virtual. But should you?

It's easy to say yes, so that way you can later use any class as a base class -- but there's a performance penalty for doing so (a virtual pointer added to every instance of your class). So you have to balance that cost, as well as your intent.

Conventional wisdom (as initially put forth by Herb Sutter, a highly regarded C++ guru) has suggested avoiding the non-virtual destructor memory leak situation as follows, "A base class destructor should be either public and virtual, or protected and nonvirtual." A class with a protected destructor can't be deleted via a pointer, thus preventing the accidental deleting of a derived class through a base pointer when the base class has a non-virtual destructor. Unfortunately, this also means the base class can't be deleted through a base class pointer, which essentially means the class can't be dynamically allocated or deleted except by a derived class. This also precludes using smart pointers (such as std::unique_ptr and std::shared_ptr) for such classes, which limits the usefulness of that rule. It also means the base class can't be allocated on the stack. That's a pretty heavy set of penalties.

Now that the final specifier has been introduced into the language, our recommendations are as follows:

- If you intend your class to be inherited from, make sure your destructor is virtual.
- If you do not intend your class to be inherited from, mark your class as final. This will prevent other classes from inheriting from it in the first place, without imposing any other use restrictions on the class itself.

Next lesson
25.5 Early binding and late binding

Back to table of contents

Previous lesson
25.3 The override and final specifiers, and covariant return types