12.3 — Virtual destructors, virtual assignment, and overriding virtualization

BY ALEX ON FEBRUARY 1ST, 2008 | LAST MODIFIED BY ALEX ON MAY 3RD, 2017

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()" << std::endl;
    }
};

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

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

    ~Derived() // note: not virtual
    {
        std::cout << "Calling ~Derived()" << std::endl;
        delete[] m_array;
    }
};

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

    return 0;
}
```

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()" << std::endl;
    }
};

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()" << std::endl;
        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.

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:

```
class Base
{
public:
    virtual const char* getName() { return "Base"; }
};

class Derived: public Base
{
public:
    virtual const char* getName() { 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;
    Base &base = derived;
    // Calls Base::GetName() instead of the virtualized Derived::GetName()
    std::cout << base.Base::getName() << std::endl;
}
```

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

12.4 -- Early binding and late binding

Index

12.2a -- The override and final specifiers, and covariant return types

Share this:

Facebook Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

49 comments to 12.3 — Virtual destructors, virtual assignment, and overriding virtualization

Rev

July 1, 2018 at 12:55 am · Reply

So, Mr. Alex, or any other respectable admin, I have an extremely silly question to ask. I define 3 classes like this:

```
#include <iostream>

class A
{
public:
	A(){}
	virtual void print() { std::cout<<"A class"; }
};

class B: public A
{
public:
	B(){}
	virtual void print() { std::cout<<"B class"; }
};

class C: public B
{
public:
	C(){}
	virtual void print() { std::cout<<"C class"; }
};
```

And i create a reference of an A object which refers to a C

```
int main()
{
    C c;
```