12.3 — Virtual destructors, virtual assignment, and overriding virtualization

BY ALEX ON FEBRUARY 1ST, 2008 | LAST MODIFIED BY ALEX ON MARCH 6TH, 2016

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:

```
class Base
{
public:
    ~Base()
    {
        cout << "Calling ~Base()" << endl;
    }
};

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

public:
    Derived(int nLength)
    {
        m_pnArray = new int[nLength];
    }

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

int main()
{
    Derived *pDerived = new Derived(5);
    Base *pBase = pDerived;
    delete pBase;

    return 0;
}
```

Because pBase is a Base pointer, when pBase 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). We do this by making Base's destructor virtual:

```
class Base
{
public:
    virtual ~Base()
    {
        cout << "Calling ~Base()" << endl;
    }
};

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

public:
    Derived(int nLength)
    {
        m_pnArray = new int[nLength];
    }

    virtual ~Derived()
    {
        cout << "Calling ~Derived()" << endl;
        delete[] m_pnArray;
    }
};

int main()
{
    Derived *pDerived = new Derived(5);
    Base *pBase = pDerived;
    delete pBase;

    return 0;
}
```

Now this program produces the following result:

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

Again, whenever you are dealing with inheritance, you should make your 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.

Overriding virtualization

Very rarely you may want to override 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:

```
int main()
{
    Derived cDerived;
    Base &rBase = cDerived;
    // Calls Base::GetName() instead of the virtualized Derived::GetName()
    cout << rBase.Base::GetName() << endl;
}
```

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

The downside of virtual functions

Since most of the time you'll want your functions to be virtual, why not just make all functions virtual? The answer is because it's inefficient -- resolving a virtual function call takes longer than a resolving a regular one. Furthermore, the compiler also has to allocate an extra pointer for each class object that has one or more virtual functions. We'll talk about this more in the next couple of lessons.

12.4 -- Early binding and late binding

Index

12.2 -- Virtual functions

Share this:

Email Facebook 2 Twitter Google Pinterest

C++ TUTORIAL | PRINT THIS POST

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

sandhya
July 11, 2008 at 12:40 am · Reply