11.3 — Order of construction of derived classes

BY ALEX ON JANUARY 7TH, 2008 | LAST MODIFIED BY ALEX ON JANUARY 7TH, 2008

In the **previous lesson on basic inheritance in C++**, you learned that classes can inherit members and functions from other classes. In this lesson, we're going to take a closer look at the order of construction that happens when a derived class is instantiated.

First, let's introduce some new classes that will help us illustrate some important points.

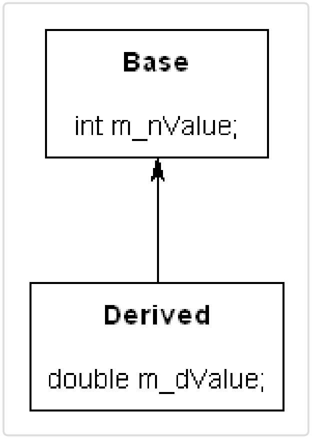
```
class Base
{
public:
    int m_nValue;

    Base(int nValue=0)
        : m_nValue(nValue)
    {
    }
};

class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0)
        : m_dValue(dValue)
    {
    }
};
```

In this example, Derived is derived from Base. Because Derived inherits functions and variables from Base, it is convenient to think of Derived as a two part class: one part Derived, and one part Base.



You've already seen plenty examples of what happens when we instantiate a normal (non-derived) class:

```
int main()
{
    Base cBase;

    return 0;
}
```

Base is a non-derived class because it does not inherit from anybody. C++ allocates memory for Base, then calls Base's default constructor to do the initialization.

Now let's take a look at what happens when we instantiate a derived class:

```
int main()
{
    Derived cDerived;

    return 0;
}
```

If you were to try this yourself, you wouldn't notice any difference from the previous example where we instantiate the non-derived class. But behind the scenes, things are slightly different. As mentioned, Derived is really two parts: a Base part, and a Derived part. When C++ constructs derived objects, it does so in pieces, starting with the base portion of the class. Once that is complete, it then walks through the inheritance tree and constructs each derived portion of the class.

So what actually happens in this example is that the Base portion of Derived is constructed first. Once the Base portion is finished, the Derived portion is constructed. At this point, there are no more derived classes, so we are done.

This process is actually easy to illustrate.

```
#include <iostream>
using namespace std;

class Base
{
public:
    int m_nValue;

    Base(int nValue=0)
        : m_nValue(nValue)
    {
        cout << "Base" << endl;
    }
};

class Derived: public Base
{
public:
    double m_dValue;

    Derived(double dValue=0.0)
        : m_dValue(dValue)
    {
        cout << "Derived" << endl;
    }
};

int main()
{
    cout << "Instantiating Base" << endl;
    Base cBase;

    cout << "Instantiating Derived" << endl;
    Derived cDerived;

    return 0;
}
```

This program produces the following result:

```
Instantiating Base
Base
Instantiating Derived
Base
Derived
```

As you can see, when we constructed Derived, the Base portion of Derived got constructed first. This makes sense: logically, a child can not exist without a parent. It's also the safe way to do things: the child class often uses variables and functions from the parent, but the parent class knows nothing about the child. Instantiating the parent class first ensures those variables are already initialized by the time the derived class is created and ready to use them.

Order of construction for inheritance chains

It is sometimes the case that classes are derived from other classes, which are themselves derived from other classes. For example:

```
class A
{
public:
    A()
    {
        cout << "A" << endl;
    }
};

class B: public A
{
public:
    B()
    {
        cout << "B" << endl;
    }
};

class C: public B
{
public:
    C()
    {
        cout << "C" << endl;
    }
};

class D: public C
{
public:
    D()
    {
        cout << "D" << endl;
    }
};
```

Remember that C++ always constructs the "first" or "most base" class first. It then walks through the inheritance tree in order and constructs each successive derived class.

Here's a short program that illustrates the order of creation all along the inheritance chain.

```
int main()
{
    cout << "Constructing A: " << endl;
    A cA;

    cout << "Constructing B: " << endl;
    B cB;

    cout << "Constructing C: " << endl;
    C cC;

    cout << "Constructing D: " << endl;
    D cD;
}
```

This code prints the following:

```
Constructing A:
A
Constructing B:
A
B
Constructing C:
A
B
C
Constructing D:
A
B
C
D
```

Conclusion

You will note that our example classes in this section have all used default constructors. In the next lesson, we will take a closer look at the special role of constructors in the process of constructing derived classes.

11.4 -- Constructors and initialization of derived classes

Index

11.2 -- Basic inheritance in C++

Share this:

Email Facebook 2 Twitter Google Pinterest