


24.3 — Order of construction of derived classes

 learncpp.com/cpp-tutorial/order-of-construction-of-derived-classes/

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_id {}

    Base(int id=0)
        : m_id { id }
    {
    }

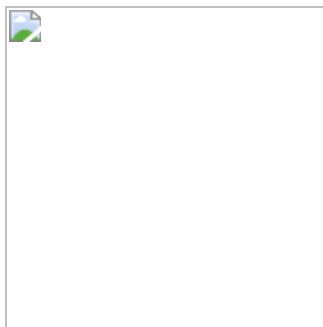
    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost {};

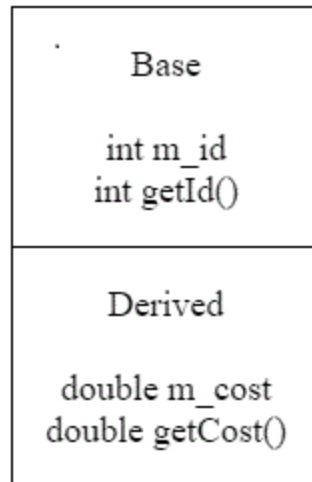
    Derived(double cost=0.0)
        : m_cost { cost }
    {
    }

    double getCost() const { return m_cost; }
};
```

In this example, class `Derived` is derived from class `Base`.



Because Derived inherits functions and variables from Base, you may assume that the members of Base are copied into Derived. However, this is not true. Instead, we can consider 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 base;

    return 0;
}
```

Base is a non-derived class because it does not inherit from any other classes. 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 derived;

    return 0;
}
```

If you were to try this yourself, you wouldn't notice any difference from the previous example where we instantiate non-derived class Base. But behind the scenes, things happen slightly differently. As mentioned above, Derived is really two parts: a Base part, and a Derived part. When C++ constructs derived objects, it does so in phases. First, the most-base class (at the top of the inheritance tree) is constructed first. Then each child class is constructed in order, until the most-child class (at the bottom of the inheritance tree) is constructed last.

So when we instantiate an instance of Derived, first the Base portion of Derived is constructed (using the Base default constructor). Once the Base portion is finished, the Derived portion is constructed (using the Derived default constructor). At this point, there are no more derived classes, so we are done.

This process is actually easy to illustrate.

```
#include <iostream>

class Base
{
public:
    int m_id {};

    Base(int id=0)
        : m_id { id }
    {
        std::cout << "Base\n";
    }

    int getId() const { return m_id; }
};

class Derived: public Base
{
public:
    double m_cost {};

    Derived(double cost=0.0)
        : m_cost { cost }
    {
        std::cout << "Derived\n";
    }

    double getCost() const { return m_cost; }
};

int main()
{
    std::cout << "Instantiating Base\n";
    Base base;

    std::cout << "Instantiating Derived\n";
    Derived derived;

    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:

```

#include <iostream>

class A
{
public:
    A()
    {
        std::cout << "A\n";
    }
};

class B: public A
{
public:
    B()
    {
        std::cout << "B\n";
    }
};

class C: public B
{
public:
    C()
    {
        std::cout << "C\n";
    }
};

class D: public C
{
public:
    D()
    {
        std::cout << "D\n";
    }
};

```

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()
{
    std::cout << "Constructing A: \n";
    A a;

    std::cout << "Constructing B: \n";
    B b;

    std::cout << "Constructing C: \n";
    C c;

    std::cout << "Constructing D: \n";
    D d;
}

```

This code prints the following:

```

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

```

Conclusion

C++ constructs derived classes in phases, starting with the most-base class (at the top of the inheritance tree) and finishing with the most-child class (at the bottom of the inheritance tree). As each class is constructed, the appropriate constructor from that class is called to initialize that part of the class.

You will note that our example classes in this section have all used base class default constructors (for simplicity). In the next lesson, we will take a closer look at the role of constructors in the process of constructing derived classes (including how to explicitly choose which base class constructor you want your derived class to use).