Inheritance, Initialization, Constructors & Destructors

Time for a new topic: inheritance and constructors. In some sense, I should skip this topic: it is essentially the same in C++ as it is in Java. But it is important, so let me cover it anyway.

The question is about initialization and inheritance. Let’s return to our familiar scenario with (1) an Animal class, (2) a Mammal class that inherits Animal, and (3) a Quagga class that inherits Mammal. If we create a new instance of Quagga, how do the Animal and Mammal parts of the Quagga get initialized?

Let’s start with Java. In Java, as in C++, inheritance is union. So if the Quagga class inherits the Mammal class, it inherits all the data fields and all the methods of Mammal.

But technically, constructors are not methods. They have no return types or values. More importantly, they cannot be methods because the object instances they are called on haven't been initialized yet (that’s their job). So they are not true methods. As far as I can tell, they don’t have much of an explicit status. So they might not be inherited.

So what happens in Java when Quagga inherits Mammal, and then a new copy of Quagga is created, triggering the Quagga constructor? One of three things:

- The Quagga constructor begins with super(...). This is an explicit call to the parent's constructor, in this case the Mammal constructor. The result of this is that the Mammal component of Quagga is initialized before the rest of the Quagga constructor starts.
- The Quagga constructor starts with a call to another Quagga constructor. In this case, the problem is (temporarily) avoided through recursion. But of course, it only delays the question: how does this other constructor begin?
- The Quagga constructor does not begin with super or another Quagga constructor. In this case, there is an implicit call to super(). In other words, the default Mammal constructor is executed before the Bird constructor.

Why invoke the parent’s constructor before the child’s? The key idea is union. The methods and fields of Animal do not depend on Bird, so they can be safely initialized. But the fields and methods of Bird might depend on the fields or methods of Animal. So it is important for the Animal part to be initialized first.

OK, so what is different in C++? Not that much, although the syntax is a little more direct.

In C++, like Java, a constructor is not technically a method or a function. (Why? Because it doesn’t return a value, and because the instance doesn’t exist yet.) Nonetheless, it acts like a method.

But there is another difference: the initialization list. In C++, constructors have an optional initialization list between the parameters and the function body. (Ordinary methods don’t have this.) The initialization list can be used to initialize data fields before the constructor proper starts to execute.
So far, we have only used the constructor to initialize non-inherited data fields. If we initialize all of them, the body of the constructor can be empty (and the compiler doesn’t have to generate a stack frame or a function call – after all, it’s not a method.)

If you don’t initialize them explicitly in the initializer list, all non-primitive data fields are initialized using their default constructor before the body of the constructor is entered.

But what about the parent class? It’s a part of the class, and it’s not a primitive. So by default, the parent section of the child class is initialized before the body of the child’s constructor is entered. Just like every other field.

If you don’t want to initialize the parent using its default constructor (or if it doesn’t have one), the parent class can be added to the initialization list. For example:

```cpp
Bird::Quagga() : Mammal(3)
{
...
}
```

So C++ and Java are the same in this: in both languages, inheritance is union. And in both languages, if B is a child of A and C is a child of B, then the C constructor begins with a B constructor, which begins with an A constructor....

This is automatic. You can influence which parent constructor is called (assuming your parent has more than one) using the initializer list, but by the time you enter the body of the constructor of the child class, the parent (and all other non-primitive fields) are initialized.

How about destructors? Well, to destruct a Quagga you also have to destruct the Mammal. But you can have a Mammal without a Quagga, but not vice-versa. Therefore the destructors get called in the opposite order: ~Quagga(), then ~Mammal(), then ~Animal().

This is simple, but guaranteed to show up the midterm... (not so subtle hint).

Now, lets talk about virtual pseudo-methods. Can a constructor be virtual? No, that makes no sense. It creates the class instance. It can’t possibly be called on a descendent of the type is was called to initialize (except as part of the inherited initialization chain). Can a destructor be virtual? Sure. Does this make sense? When?

Virtual destructors not only make sense, they are a best practice anytime you are using inheritance. Think about it: your program wants to destruct a Mammal. What if what it really has is a child of Mammal, like Quagga? Don’t you want to destruct the whole instance? Anything else would slide the data and create a memory leak...

So rule of thumb: make your destructors virtual!