Chapter 9 - Inheritance

Just like Python, C++ supports inheritance. It even supports multiple inheritance (which we will talk about later). The basic syntax for inheritance in c++ is pretty simple.

```
class <classname> : <access specifier> <parent class> {

};
```

The access specificier specifies the most accessible level in the derived class. Normally, you make this *public*. That means any *public* members remain *public*. If you set this to *private*, that means any *public* member data in the parent class becomes *private* in the derived class. And, likewise, the same goes for *protected*.

Construction

When you write your constructor, you have to invoke the parent constructor. This is normally done in the initializer list, as the first entry. For example, say we have a class **Foo** inheriting from **Bar**. Furthermore, say **Foo** has a member variable named **m_parent* and Bar's constructor takes a reference to a string called name. Here is how we would handle this in c++:

```
class Foo : public Bar {
    string m_parent;
public:
    Foo(const string& name, const string& parent) :
    Bar{name},
    m_parent{parent}
    {}
};
```

There is a decent amount to unpack here, so lets go over this:

First, our example Foo class inherits from Bar

```
class Foo : public Bar {
```

Next, we define a constructor which takes two strings

```
Foo(const string& name, const string& parent)
```

Then, we define our initializer list, beginning with the parent class constructor, followed by the initialization of Foo's data:

```
Foo(const string& name, const string& parent) : Bar{name} , m_parent{parent}
```

Notice that we initialize Bar with *name* from the constructor, and m_parent with *parent* also from the constructor.

virtual functions and pure virtual functions and base class pointers

Recall for a moment (and every moment hereafter) that C++ is a statically typed language. That makes defining heterogenious collections a big pain in the butt. One of the ways of handling this is via inheritance. The big idea is that you can define an *interface* in a base class, and implement that interface in a number of inherited classes. Then you can pass derived classes to functions which take pointers to the base class, and as long as said functions restrict themselves to operating on the interface defined in the base class, all is well.

This all probably is a bit abstract, so we are going to provide an example. We are going to create a base class called *Animal*.

```
class Animal {
    std::string m_name;
 public:
    virtual void vocalize() const=0;
    virtual void eat() const=0;
    virtual void run() const=0;
};
```

Hey, what is that *virtual* business? And what is that `=0` thing?

c++11 override , final keywords

```
class Foo : public Bar {

    int somefunc() override;
}
```