Programming Notes 12: Inheritance and Polymorphism

As usual, more examples of the notes are in the example program. The example is very important this week! It gives a full overview of templating a data structure.

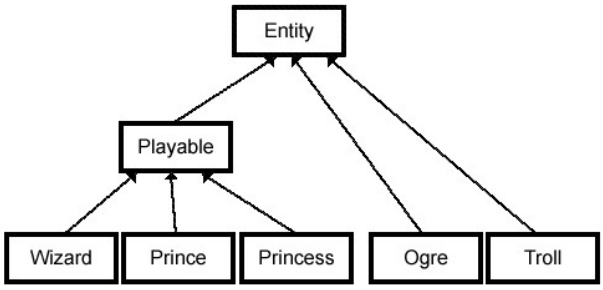
1. Inheritance

Inheritance and polymorphism are two defining characteristics of object-oriented programming—with only what we've learned so far, OOP isn't a whole lot different than procedural programming. These concepts are so defining, in fact, that if someone says they use OOP, it very often means they use inheritance and polymorphism (although the design pattern of composition is also rather popular).

So, what is inheritance? Generally, it's a way to organize classes into a hierarchy where data and logic can be shared and overridden. Specifically, one creates what's called a "base class," which contains the most general logic and data, creates "derived classes" which extend and modify the functionality of the base class. Some examples of this are (see: the example program) having an "animal" base class that is extended by specific classes like "dog" and "cat." Derived classes are said to "inherit" from their base class, and the base class is called the "parent" or just the base class of its derived classes.

In C++, any class can be inherited from—derived classes can themselves be used as base classes—multiple derived classes can inherit from the same base class, and a derived class can inherit from multiple base classes.

It's common practice to diagram inheritance trees (as they're called) like we diagramed linked trees, although here, the links represent inheritance rather than pointers. For example, one could structure the entities in a game as such:



Inheritance can be useful in many situations—entire languages, such as Java and C#, are based off of it—but I'm not trying to say OOP is the best programming paradigm to use in any situation. Ideally, you should assess what you're trying to do and choose whichever method will be the most effective. Some people *really* like OOP, some abhor it. Personally, I'd stay at a distance from complicated inheritance and polymorphism trees and logic, as I find it more effective, maintainable, and performant to use other design patterns.

2. Public, Private, and Protected

We've already learned about the "public" and "private" keywords to specify data availability within classes. To rehash, "public" allows anything and anyone to access the method or data, and "private" allows only other member functions of that class to access the method or data.

However, there is a third keyword we haven't talked about: "protected." Essentially, "protected" signifies that the class itself can access the data (like "private"), but *derived* classes may also access the data. For example, if your "animal" base class contained a "protected" name data member, all derived classes (e.g. "dog") can

access the name member as if it is a member. However, "private" data in a base class will not be accessible through a base class—still only the base class' member functions may access it. This means that if a class has only private data, there's likely not much point in inheriting from it, as the derived class won't receive any of the data.

Why ever use "private" instead of "protected," you might ask, which is a good question. Well, the most common use of "private" is to make sure that the class is not inheritable (technically, it still is, but it will not be useful to do so). Additionally, one can make sure that important data—say an encryption key—will never be messed with if someone else (or their self) extends that class and adds functionality.

3. Using Inheritance

Declaring an inherited class is quite simple in C++: simply begin like any other class with the keyword "class" and your identifier, but then add a colon, an access modifier, and your base class. For example:

```
class dog : public animal {};
class monster : protected entity {};
```

Here, the access modifier has a different meaning; it represents how the base members are interpreted *within* the derived class. Inheriting as "public" will change nothing about the base class's members—public members will remain public and protected will be protected. Inheriting as "protected" will simply make the public members protected within the derived class, and the protected members will still remain protected. Finally, inheriting as "private" will make both public and protected members of the base class private within the derived class.

After declaring your class as inherited, the rest of the syntax is exactly the same as we've learned, except that you can now use the "protected" keyword. Implementing the class is also the same, except that you can use the inherited data members and member functions. Once you've created your derived class, you can create it, use it, and destroy it like any other class. Example:

```
class animal
{
public:
        animal();
        ~animal();
       char name[10];
}

class dog : public animal
{
public:
        dog();
        ~dog();
        int size;
};

dog derived;

derived.size = 10;
derived.setName("fido");
```

4. Polymorphism

Using purely inheritance allows you to extend functionality and create sub-types, but you're still limited to using only your new derived class, and you can't change the base functionality of the base class. Polymorphism allows you to generalize derived classes by their base class, enabling you do things such as store them under one type, generalize functionality, and customize logic. Essentially, your bass class specifies an interface that all subclasses are guaranteed to support.

For example, if you created some polymorphic animals (again, see example), your base "animal" class specifies what functions will be available in all subclasses, so you can have your "dog" override the "speak" method to print "bark!," while your "cat" can override the "speak" method to say "meow!"

Now, you can store all your "cats" and "dogs" within the same, single-typed data structure, under their shared base class "animal." In C++, polymorphism only works using pointers/dynamic memory. This means that if you, for example, create an array that holds "animals," all your "cats" and "dogs" will be converted to simple "animals" when added to the array, but if you create an array that holds pointers to "animals," your "cats" and "dogs" remain "cats" and "dogs" when you add their addresses to the array.

See example program.

5. Virtual Functions

In C++, you must specifically define what member functions of a class are polymorphic and hence overrideable. This specifies that the function will be at least available in all derived classes, but it can be changed, overridden to do different things depending on the derived class. If the function exists in the derived class, even when accessing, for example, a "dog" as a general "animal," the more specific version ("bark!") will always be called. If the derived class doesn't implement the function, the base class's more general version will be called. The keyword "virtual" is used for this, and any virtual function will make the class polymorphic.

You are almost always going to want *at least* your base class destructor to be virtual, as this tells the program to first call the destructor of the derived class, then the destructor of the base class when an object of the derived class is destroyed. If you make another member function virtual, it works similarly to the destructor: if it is redefined within your base class, the more specific function will be called, even if you're looking at, for example, your "dog" as simply an "animal." However, the more general base class function will not be called afterwards.

All of this is rather theoretical; I really encourage you to check out the example program. Example:

```
class animal
{
public:
        animal();
        virtual ~animal();
        virtual speak();
}
```

```
class dog : public animal
{
public:
        dog();
        ~dog();
        speak();
}
```

```
dog* doge = new dog();
animal* generalAnimal = doge;

generalAnimal->speak();          // Will call speak() from dog
delete generalAnimal;            // Will call ~dog() then ~animal()
```

6. Using Polymorphic Classes

Finally, using your polymorphic, inherited classes is no different than using normal classes – you can still use them on their own. However, with polymorphic classes, you can create data structures of general (base) data types and actually store the specific (derived) data types within them, and have the functionality be preserved. Remember, you must use pointers for the polymorphic references to work.

See the example program, where we create an array (vector) of animals and compare it to an array of animal*s in terms of polymorphic properties.