Structs and Classes

C++ has two data structures that resemble python classes. They are structs and classes.

They are so similar that, rather than talk about them separately, I will discuss classes and then come back and talk about the trivial differences between classes and structs.

So, on with it.

Classes

Since this is a class for folks who are python savvy, let's compare c++ classes to python classes.

class Keyword

Like in python, in c++, a class is defined using the class keyword:

Python

```
# python  class Person(object):
```

C++

```
class Person {  };
```

Self Reference

In python, each instance method has an explicit reference to itself, commonly spelled out as *self*, which all instance methods are passed explicitly as their first parameter.

C++ has an implicit reference to itself, called *this* which is a pointer to itself (we cover pointers elsewhere). However, you rarely need to use *this*, as C++ is usually smart enough to recognize references to member functions and data.

For completeness sake (hopefully this isn't confusing), consider a greet method defined in a person class in python and c++, which we assume has an instance variable called *myname*:

python

```
...
def greet(self, other):
    print "Hello {}, my name is {}".format(other, self.myname )
```

c++

```
...
void greet(const std::string& other) {
```

```
    cout << "Hello "<< other << ", my name is " << myname << endl;
}
```

Access Specifiers - Private, Public, Protected

One thing you might have fussed with in python a bit is the notion of privacy. You probably have prefixed variables with an underscore as a way of telling the rest of the world that a variable is part of an implementation, and shouldn't be mucked with. You might have even gone so far as to use two underscores in order to invoke name mangling. Well, no offense to python, but its implementation of privacy is second rate and you can get around it. Privacy is kind of like what sexist dinosaurs would have referred to as a "gentlemen's agreement". Phoey!

In a c++ class, all of your variables and functions are defined under an access specifier. And there are three of them:

- public
- private
- protected

Public data and functions may be accessed by anyone. Instance methods, external callers, you name it.

Private data and functions may only be accessed internally by other functions within the class.

Lastly, protected data and functions may only be accessed by internal functions / data or by internal mechanisms of classes which inherit from the class in which the protected data lives.

Oh, and access specifiers may be used more than once. Anything following an access specifier belongs to it. And class data is private by default. So if you see data or methods but no specifier, that data and those methods are private....

Example Time:

```
// person defined with all public methods
class Person {
 public:
    // constructor
    Person(std::string name) : name(name) {};
    std::string name
};

//
class PrivatePerson {
private:
    std::string name;
public:
```

```
    PrivatePerson(std::string name) : name(name) {};
};
//
class ProtectedPerson {
protected:
    std::string name;
public:
    ProtectedPerson(std::string nm) : name(nm) {}
};

class Man : public ProtectedPerson {
    public:
    void greet() { std::cout << "Hi, my name is "<< name << std::endl;}
}
//
// main funciton

int main() {
    // this works
    Person dude("The Dude");
    std::cout << dude.name << std::endl;

    // this doesnt
    PrivatePerson privateDude("The Private Dude");
    std::cout << privateDude.name << std::endl;

    // protected
    Man theDude("The Dude");
    theDude.greet();
    // this doesnt work
    theDude.name = "Shelly";

    return 0;
};
```

Constructor

Python has the initializer, otherwise known to true geeks as the dunderinit (**init**). Technically, this is an initializer, not a constructor, but lets gloss over that.

In C++, you have one or more constructor member functions. I say one or more, because c++ supports overloaded functions in addition to default parameters. So, for any function, you can redefine it multiple times, as long as the parameters differ in each version. This added flexibility is useful, because c++ is strictly typed... so, getting back to the constructor, lets take a look at how it is formed.

Adding to our person class:

```
Person {
    std::string firstname, lastname;
public:
    Person() : firstname(""), lastname("") {}; // default constructor
    Person(const std::string& fname, const std::string& lname) const : firstname(fname), las
}
```

So, lets talk about this. There are a couple of interesting things going on here. First, we have something called the *member initializer list*. it appears after the parens and before the brackets. It starts with a colon. See it?

```
somefunction() : <intializier list> {}
```

The initializer list is a comma separated list of data members which may be initialized form the parameter list of the function, or any constants, using method or uniform initialization notation (assuming you are compiling with c++11 support).

Copy Constructor

One special form of constructor is the copy constructor. The copy constructor is engaged when you initialize a new instance variable from an existing instance variable. For example:

```
Person p("Frank", "Ford");
Person p2 = p1;
```

What is really going on here? A bit of syntactic sugar, to be frank. Here is the translation:

```
Person p("Frank", "Ford");
Person p2(p);
```

(and by the way, this also calls the copy constructor.)

The copy constructor's signature looks like this. (commit it to memory)

```
Person(const Person& p);
```

It takes a const reference to a variable of the same type. Its job is to initialize the new variable with the values of the old variable. If you do not create a copy constructor, one will be provided for you by the compiler. But, and this is a big but, it will often not have the intended behavior. Especially when you start getting into managing memory yourself. At this point, with the person class, there is really no need for a custom copy constructor, but here is how you would implement it:

```
Person(const Person& p) : first_name(p.first_name), last_name(p.last_name) {};
```

Assignment Operator

The assignment operator is the kissing cousin of the Copy Constructor. It's job is to make a copy of the a variable and assign it to an existing variable of the same type. Catch that? its a subtle distinction. In the copy constructor's case, we are initializing a new variable. In the assignment operator's case, we are assigning the value of an existing variable to another existing variable. Because the variable on the left hand side of the equation already exists, in cases where we have pointer variables, with owned memory, we may need to dispose of existing memory before allocating new memory and copying values. This probably makes no sense at this point, because we have not gotten into dynamic memory allocaiton. So just take it on faith that there is a reason for all of this. Anyway, the assignment operator takes the following form:

```
Person& operator=(const Person& p) {
    if (this != &p) {
        //copy
        first_name = p.first_name;
        last_name = p.last_name;
    }
    return *this;
}
```

A couple of interesting notes. We dereference the **this** pointer and return a reference to the value (combination of the signature of the return type(Person&) and the dereferece (*this)).
Anyway, truthfully, I probably should have not plunged ahead into the Assignment Operator and the Copy Constructor; not until we talked about allocating memory. Because the default implementations which the compiler provides are suitable until we get into managing memory. So, while I cannot take back the last two sections, having already gone over them in class, don't fret if they don't quite make sense yet.