Programming Notes 14: Classes Con't

As usual, more examples of the notes are in the example program.

1. Copy constructors

The first topic we're covering today is the 3rd and final type of constructor. We've talked about default and parameterized constructors, but this is neither. As the name implies, the copy constructor will only take another object of the same class type to copy data from. There are four instances where a copy constructor is called, but you only have to worry about three of them: using another instance of your class as an initialization parameter, when you pass an object to a function by value, and when you return an object from a function by value. Example:

```
myClass c;
myClass x(c); //Calls the copy constructor, using 'c' as the parameter

void func(myClass p) { //Because 'p' is passed by value, the copy constructor of myClass will be
                       called to copy your actual parameter into 'p'
  return p; //The copy constructor will be called again here, as the compiler creates a copy of 'p' to
              actually return to the calling fuction.
}
```

Because the copy constructor is called when you pass an object by value, and the copy constructor takes a parameter of your class type, if you pass the parameter by value, you'll create an infinite loop of calling the copy constructor. Hence, you must pass the parameter by reference, and because your copy constructor shouldn't change anything in the parameter object, a constant reference. Remember that the copy constructor is still a constructor, so you will want to initialize all of your data members, do memory allocation, and all that, as well as copying the values from your "source" object. Example:

```
class card {
public:
  card::card(const card& source);
private:
  int value;
}

card::card(const card& source) {
  value = source.value;
}
```

2. Constants

You've probably seen the keyword "const," written before a variable or parameter definition. As one would expect, it stands for 'constant,' and marks some value or some code in your program that cannot be changed. As you may have seen, constants are commonly used to define global values, but there are actually four cases in which it can be useful to mark code as constant.

3. Constant variables

The most obvious use of the constant keyword is to define variables whose values cannot change. While it is often not incredibly useful to have constant values within your code, they can still be used to make sure a value that should not change doesn't change (you will get an error), and can be used to define "global constants," which are global values that can be used by your entire program, but cannot be changed. These global constants are usually named in all caps, and are acceptable, while global variables are not. Additionally, you can make any variable constant, no matter the type. Hence, you can have a constant integer, character, student, or card, or whatever you want. Example:

```
//Global constant defining the size of a deck
const int NUM_CARDS = 52;

int main() {
  //Creates a welcome text string that cannot be changed. If you try to change it later in your
  program, you will get an error.
  const string welcome = "Welcome to this program";
}
```

4. Constant parameters

The other case in which you have seen the constant keyword is in the definition of a function's parameters. This use case is very similar to creating constant variables, as it tells your function that the value of the parameter cannot be changed within the function. This means that if you passed a constant integer, your function would not be able to change the value of the integer. Using constant parameters is more useful if you take into account passing parameters by reference, and passing pointers to functions. If you pass a constant reference to a function (the keyword "const" as well as the ampersand), you will have the speed benefit of passing by reference, but because your function will not be able to change the value of the parameter, you don't have to worry about your function messing with the parameters you pass in. Hence, you should always do this when passing a class (not a pointer to a class) to a function. Second, when you pass a constant pointer to a function, it means that the function cannot change where the pointer *points*, but it can still change the data that the parameter points to. Hence, you must be wary of this. Example:

```
//This function takes a constant reference to a string class. It will not be able to change the value of
the string.
void func(const string& str);

//This function takes a constant character pointer. This means the function will not change where
the pointer points, but it can still change the data. Additionally, you must use "const" to pass a string
literal (i.e. "hello") without generating a warning.
void func2(const char* str);
```

5. Constant returns

There are two more places in which you can use constants, which you probably haven't encountered before. The first is a constant return type. You will probably never use constant returns, as they are essentially obsolete and were never incredibly useful anyhow. Basically, marking a returned value as constant prevents the function from being called on a temporary object, for example the result of an addition before it is captured elsewhere. It doesn't do anything like cause the value to become constant in the calling function, so you will most likely never see it do anything at all. If you're not exactly sure how this works, that's fine, we'll talk about this more later. Example:

```
//This function returns a constant card
const card deck::shuffle() {
  return x;
}
```

6. Constant member functions

Finally, we have the most important use of constant, and why this lesson relates to classes. Essentially, member functions of classes can be marked as constant, and it means that they will never change any of the data members of the class. For example, a print function should simply print out all the values of an object, and not change any of them. Hence, the print function should be constant. So, why is this useful? Two reasons: first, it is useful simply as a documentation feature, as others will know the function will not change any data. Second, if you create a constant variable of a class type, you'll notice you get an error if you try to call any of the member functions. You can only call constant member functions from a constant instance of your object, because you know that they will not change the data of the object. Finally, the syntax for this is simply to add the "const" keyword after your member function prototype, but before the semicolon. Example:

```
class card {
public:
  void setRank(string r);          //Not a constant member
  void print() const;              //Constant member
private:
  string rank;
};

//So later in your program, if you declare a constant card
const card c;
//You can NOT call the set function
c.setRank("king"); //NO

//But you can still call the print function
c.print(); //Yes
```

7. Friendship

Moving on from the uses of constants, we have our second mini-lesson, friendship. Friendship is extremely simple: it allows non-member functions and classes to have access to the private data members of a specific class type. You have to specifically specify what functions or classes can access the private data in your class definition. To do this, simply add the keyword "friend" before either the prototype of the friend function, or the name of the friend class. This allows the function or class to access private data members of objects of the specified type, but note that they are not member functions of that class. This means that, for example, if a friend function is passed a class object of your type, it can access the private members of that parameter. Additionally, because friend functions are not member functions, they are not called using an instance of the object and the dot operator, but instead are called just like any other normal function you've used in the past. Granted, if your function does not take a parameter of your type, it's not very useful for it to be a friend. Example:

```
class card {
public:
  //functions
private:
  char* rank;

  friend void print(const card& c); //Allows the print function to access the rank data member
  friend class deck; ///Allows the deck class to access the rank data member
};

void print(const char& c) {
  cout << "rank: " << c.rank; //This function can access the private rank member of the card
}

int main() {
  card c;
  print(c); //Calling the friend function is no different than calling any other funciton
}
```

Finally, an interesting feature of friendship is that it gives you the option to only allow a specific class the ability to create objects of another class. Remember when I said your constructors should always be public? Well, if you make all your constructors private, and give another class friendship, that class can create instances of your private class, as it can access private members, but no one else can. Cool, huh?