Programming Notes 13: Classes

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

1. Object Oriented Programming (OOP)

We're now going to start learning about the most popular programming paradigm, Object Oriented Programming, or OOP. OOP is not necessarily the best programming method in general (despite what some may tell you), but it is often very useful and is certainly the most popular. So, the basic element of OOP, is, unsurprisingly, the object. Essentially, an object is simply something that can hold both data and logic to manipulate that data. Hence, the programming style is centered on creating these objects and the relationships between them. The basic concept is really just that simple, although, of course, there are many, many different ways in which you can manipulate objects in C++, so there's a lot to learn.

C++ is an object oriented language, and it's concept of an object is implemented using classes. A class is like a structure in that it's an abstract datatype that you define, which we learned about previously, except that it can utilize several more features that the structure can't. We'll get into some of the more complicated aspects next semester, so for now, just know that a class can use these three new concepts: private and public data, member functions, and self-contained memory. *Technically*, structures can also have member functions, but it is congenital to use a class if you want your type to contain more than just data.

2. Declaring a Class

To declare an object in C++, simply use the "class" keyword. The basic structure is the same as with structures: after "class," add the type name of your class, brackets, and end it with a semicolon. Also like structures, you put your data within your class's scope. However, with classes you can also define member functions of the class here. Declaring a member function is just like declaring any other function: start with the return type, type the function name, and add your parameters. These functions, like the data, will be a part of your class, and hence will operate on the data contained within it. Example:

```
class card {
  char* rank;
  char* suit;
  int value;

  //These are member functions, and will operate on the classes' data.
  void print();
  char* getSuit();
};
```

3. Private vs. Public

If you try creating a class like the one above, you'll run into an error if you ever try to use the class, and it will say that something or other is trying to access a "private" member of the class. This is because the default access modifier of data within a class is private. This means that no part of your program other than member functions of the class itself can access that data—your main function, for example, cannot ever touch any of that data. However, this also means that if everything is private, nothing can use your class, which you don't want. Instead, you can specify that some member functions and/or data is "public," meaning that anyone can access it—both your main function and the member functions, for example. However, you're usually going to want your actual data members of the class to be private, because only your class should be able to operate on them. If any part of your program can have access to the data contained in your class, they can change it, and your class won't know what happened! Hence, you want all use of data to go through your class, meaning you write member functions that will operate on the data instead of other parts of the program. Finally, to specify that a section of your class is public, and a section is private, use the keywords "public" and "private" followed by a colon to mark that section of your class as specified. Generally, your public member functions should go before the private ones. Example:

```
class card {
public:
  //These member functions will be able to be accessed from anywhere
  void print();
  char* getSuit();

private:
  //These data members will only be able to be changed by the other member functions.
  char* rank;
  char* suit;
  int value;
};
```

4. Member Functions

So, I've been talking about member functions for the last few sections, but I haven't exactly explained how they are used. First, remember that they belong to the class, just like the data belongs to the class. Hence, to access or call a member function of a class, you can also use the dot or arrow operator, depending on your situation. For example, if we have an instance of the "card" class from above…

```
card c1;
c1.print();   //Will call the print() function of the "card" class.
```

Also like structures, each of your instances of your class will have its own set of data, so calling the member functions that act on that data will produce different results for each instance.

Now, while you can technically implement your functions directly where they are prototyped within your class, it is conventional to define them outside your class—in fact usually in a different file.

However, we'll get to multiple files later. To implement your class members outside of your class, it's just like implementing any other function, except you type the class type name and the scope resolution operator (two colons or "::") before the name of the function. You then implement your function as usual, except that because this function is a member of a class, you know it will have to be attached to an instance of a class, and will also have access to the data of that object. To access the data (because it's a member function), you can simple type the variable names declared in your class definition. These will pertain only to the "calling object," which essentially means from the instance of the object from which you use the dot/arrow operator to call the function. For example:

This implements the card's print member function. It automatically has the data members available to it.

```
void card::print() {
  cout << rank << " " << suit << " " << value << endl;
}
```

So, when you call the print function from an instance of a card (as like below), this function will be called, and it will use the data from that particular instance of card.

```
card c2;
c2.print();
```

5. Getters and Setters

Member functions are usually used to manipulate the data contained by the class, but because your data will be private, you will often need a quick way to read from or write to the data. This is where getter and setter member functions come into play—they do what they say on the box, read from and write to your data members. While it is indeed worse than absolutely useless to have a getter and setter for a value if (somewhat counterintuitively) literally all they do is return the value or set the value, but when you start to build more complicated objects than what are basically data containers, they will necessitate more complicated methods of assigning and retrieving data. This is the real value of getters and setters—they encapsulate the accessing and assigning logic and make it very easy for the rest of your program to use your class.

You can see that while the getter shown here is pretty useless, the setter will automatically work with the dynamically allocated string, so your main function or wherever else does not have to worry about it.

```
char* card::getRank() {
  return rank;
}

void card::setRank(const char* r) {
  delete[] rank;
  rank = new char[strlen(r) + 1];
  strcpy(rank,r);
}
```

6. Constructors

There is a special type of member function that all classes have, whether you implement it or not. This is the constructor. The constructor, as the name implies, is called once and only once, when an instance of you object comes into being, technically known as instantiated. There are several types of constructors, but they all serve the same purpose: to initialize the data of your class in an intelligent manner so the rest of your class as a starting point. For now, this will most likely mean setting values to default values and sometimes allocating dynamic memory. Another important aspect of constructors is that you never directly call them (i.e. you never do something like class.constructor()). Instead, it is automatically called for you when a variable of your classes' type is instantiated.

For example, your constructor will be called in both of these situations.

```
myclass var1;
myclass* var2 = new myclass;
```

To define a constructor, you must add it as a member function, just like any other. However, there are a few specifics here. First, all constructors must have the same name as the class. This means that, for example, if you have a class named "card," all of your constructors will literally be named "card." Second, constructors have no return type—not even "void." You simply leave out the return type. Third, your constructor must be public, or else nothing in your program will be able to create an instance of your class—the constructor is called when you instantiate an object, and if it's private, you can't call it, and hence can't create the object. Implementing a constructor is just like any other member function, except that you know when your constructor is called, it will be as if your data members were just declared—for example, they will have garbage values. You must keep this in mind—the point of the constructor is to initialize them.

Here's a class with a constructor.

```
class card {
public:
  card();
  //other functions
private:
  int value;
}

//Implement the constructor
card::card() {
  value = 0;
}
```

There are three major types of constructors, and we'll cover the first two here. The first type is the simplest—it has no parameters and will simply set your data to default values. Because it has no parameters, this is called a default constructor. An example of a default constructor is displayed above. Your default constructor will be called if you create an object without specifying anything, for example

```
card card1;   //Will call the default constructor.
```

Next, there is the parameterized constructor. As the name suggests, this constructor takes parameters that signify what the data members should be initialized to. The implementation is exactly the same as the default constructor, except that it takes parameters. Finally, remember when we went over overloaded functions, quite a while ago? Well, you can overload the constructor, meaning you can have different constructors for different situations.

For example, this class as a default and a parameterized constructor

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

//Default constructor
card::card() {
  value = 0;
}

//Parameterized constructor
card::card(int v) {
  value = v;
}
```

You might be wondering how your program knows what constructor to call if you have more than one. Well, we already mentioned how to use the default constructor, but to specify that you want to use your parameterized constructor, simply add your parameters when instantiating the class.

For example,

```
card card1;   //Will use your default constructor

card card2(10);   //Will use your parameterized constructor

card* card3 = new card(10);   //Will also use your parameterized constructor
```

7. Destructors

The counterpart to constructors is, of course, the destructor. The destructor is like the constructor in that it is only called once in the life of an object, except this time it's when the object is destroyed. Hence, the point of the destructor is to clean up your classes' data, which for now will mean cleaning up any dynamic memory allocated by your class. The object is either destroyed when it goes out of scope (if it's a statically declared variable), or it is deleted (if it's a dynamically allocated variable). Like the constructor, the destructor has a specific name, which is a tilde ("~") followed by the name of the class. Finally, although you can have multiple constructors, you can only have one destructor.

Here is an example of a class that will allocate memory on construction, and deallocate it on destruction. The dynamic memory is only seen and managed by the class itself, so the rest of your program does not have to worry about memory leaks or the like.

```
class card {
public:
  card();
  ~card();
private:
  char* name;
}

//Constructor
card::card() {
  name = new char[10];
}

//Destructor
card::~card() {
  delete[] name;
}
```

So if you were to declare an instance of your class here

```
{
 card card1;   //The default constructor will be called here on instantiation
}   //And the destructor will be called here, when the object goes out of scope
```

The constructor would allocate the memory and the destructor would delete it, without you doing anything in your main function.