Introduction To Classes

Prose lifted from an old handout written by Andy Maag.

The basic unit of abstraction in C++ is the **class**. A class is used to encapsulate some piece of user-defined data and the operations that access and manipulate that data. A class is the blueprint from which an **object** is created — put another way, an object is an instance of a class. Later in the quarter, we'll see how objects and classes become the building blocks of object-oriented programming.

In C, many of you may have seen and used abstract data types. A class is a powerful way to express abstract data types which goes beyond the facilities provided in C. C uses structures to define abstract data types and standard functions to access and manipulate these data types. You should find that working with abstract data types in C++ is more reliable and concise than it would be in C.

Readings from Eckel:

Chapters 4, 5, and 6, skipping any discussion of `friend`s.

Readings from Deitel[2]:

Chapters 6 and 7, skipping any discussion of `friend`s.

Defining a Class

Let's look at an example of a very simple class to represent a person and his popularity. We define the class by using the `class` keyword, and supply both data and function **members** for the class together. A member is just a fancy name for one data or function element of a class, and a **method** is an even fancier name for a function member.

```
class Person {

   public:
      void init(const char* newName, double newPop);
      const char* getName(void) const;
      double getPopularity(void) const;
      void setPopularity(double popularity);

   private:
      const char* name;
      double popularity;
};
```

There are a few things to notice about the definition of a class. First, the name of the class is defined before the contents of the class. This is useful for defining classes with self-referential data members, as you'd have with a linked list or binary tree.

Next, you'll notice the keywords `public` and `private`— these are called *access specifiers*. Members declared `public` can be accessed from outside of the class, while members marked as private can only be accessed from within that class's member functions. As a general design rule, data members should **always** be declared `private`. Some of your methods will be `public`, some will be `private` — you should only provide the minimal set of `public` methods that make your class useful. There is a fine line to walk between having too many function members and too few. The best way to find out what is best is to experiment until you get a feel of the right balance.

The function members in the `Person` example are an initialization method, **accessor** methods that return the value of a corresponding data member, and a **mutator** method that sets the value of a corresponding data member. Note that function overloading can be used for class methods — the accessor and mutator methods use this. However, a data member can not have the same name as a function member within the same class.

Classes may contain other types of member functions, such as functions to clean up an object just before it is deleted, or a single method that accesses or changes multiple data members.

Implementing Methods

After you define a class, you need to provide the implementation for each method in the class. Because two different classes may each have a member function with the same name, you need to specify the class name when defining it. For instance, the `init` method of the `Person` class might be defined like this:

```
void Person::init(const char* newName, double newPop)
{
   char *buffer = new char[strlen(newName) + 1];
   strcpy(buffer, newName);
   name = buffer;
   popularity = newPop;
}
```

and the `getPopularity` method might look like:

```
double Person::getPopularity(void) const
{
   return popularity;
}
```

These methods are defined similar to the way simple functions are defined in C++, but they are prefixed with the name of the class and `::`.

Instanstiate and Using Objects

Now that we've defined the `Person` class and implemented its methods, we can now create instances of it. You can declare a class either in `static`, local, or dynamic memory. A `static` variable representing John Lennon could be declared as:

```
static Person john;
```

When a variable is declared like this, it has no default value. You would have to initialize the `static` variable somewhere else in your program to John's name and a suitably high popularity factor for the slain leader of the Beatles. If you declared and initialized the following variable within a function, you would get a local variable representing George Harrison who was John Lennon's less popular band-mate:

```
Person george;
george.init("George Harrison", 5.0);
```

To dynamically allocate a `Person` variable, you can use the `new` operator. For example, to allocate a variable representing your CS193D instructor the night before an assignment is due, you could do the following:

```
Person* jerry = new Person;
jerry->init("Jerry Cain", 0.0);
```

Note that you access a function member in the same fashion that you access a data member. If the variable is declared directly, or you have a reference to it, you use the `.` operator to access a member. If you have a pointer to the variable, you should use the `->` operator to access any members.

A member function does not have to access other member functions in this way. Instead, it can use standard function call syntax.

Accessors and Mutators

Not all members need to have accessors and mutators, or at least ones that are declared `public`. Internal state for a class which is not intended to be visible outside of the class should not have `public` accessors or mutators, or else the abstraction boundary provided by classes would be broken. Also, depending on what the class represents, some members should be read-only, such as the name member in the `Person` example. In this example, while the popularity of a person may change at any time, a person's name should not change after it is initialized. To make a read-only attribute, either do not declare a mutator method for it, or make the mutator private. Naming conventions for accessors and mutators differ between programmers. Most like to adopt a convention where an accessor will be named with the prefix "`get`", and a mutator will be named with the prefix "`set`". The same example using this style would have an accessor named `getPopularity`, and a mutator named `setPopularity`. For the assignments,

you're welcome to adopt either of these conventions, or any other reasonable convention as long as you're consistent in using it.

Constant Methods

There is yet another strong use for the `const` keyword in C++, and you should use it consistently throughout your C++ classes. You can label a method as being `const`, which means that it will not change any of the data members within the object. All accessors should be declared as constant, as well as any other function that shouldn't change the object. If a method is declared constant, it can only call other `const` methods, and only use data members as constant variables—that is, data members can never be l-values in a `const` method implementation.

Logically constant means that the semantics of the method does not require any changes to the data members of the class. **Physically constant** means that no changes are made to any data members. Using the `const` keyword requires that the method be physically constant. In some cases, a method may be logically constant, but not physically constant.

Static Data Members

Suppose we wanted to keep track of the number of existing `Person` objects. We can do this by declaring a `static` variable within the class. For instance, we could add the following declaration to the `Person` class:

```
class Person
{
      ...

   private:
      static int numPeople;
      ...
};
```

The `numPeople` member should certainly be marked as `private`, since the responsibility of maintaining the count should lie on the implementation end of things. You also need to declare and initialize the variable, which is usually done in the class implementation like this:

```
int Person::numPeople = 0;
```

When declaring the variable you don't need to specify `static`, but you do need to include the class specifier. If you want an accessor or a mutator for a `static` data member, you can add a `static` member function to do this. If you want to declare a `class` constant, it's a bit easier to use a `class` enumerated type or a global constant rather than use a constant static data member.

Static Methods

Suppose we have a class-specific variable, as in the `numPeople` example above. Whenever we create a `Person` object we'll increment the `numPeople` variable, and when we destroy a `Person` object, we'll decrement it. To do this, we'll need an accessor and a mutator for `numPeople`. We can do this by adding the following function members to the `Person` class:

```
class Person
{
   public:
      static int getNumPeople(void);
      static void incrementNumPeople(void);
      static void decrememtNumPeople(void);
      ...
};
```

You also need to provide the implementation for these functions, which is usually done as part of the class implementation. Static methods cannot access any non-static data or function members, because they are not invoked on a specific object, and they cannot be declared `const`.

```
int Person::getNumPeople(void)
{
   return numPeople;
}

void Person::incrementNumPeople(void)
{
   numPeople++;
}
```

After doing this, you can call these two functions. In order to call a `static` function, you qualify it with the class name, because two different classes might have the same `static` function. However, you don't need to specify which object to call the method with because you're not accessing any data. For example to read the number of persons, you could do the following:

```
int classSize = Person::getNumPeople();
```

`this`

Each non-static member function has an implicit parameter named `this`. The `this` parameter is a pointer to the object to which the message (the invocation of a method) is being sent . Using the `Person` class example, when we invoke the `init` method on a variable as in:

```
jerry->init("Jerry Cain", 0.0);
```

Presented here is the implementation of the `init` method, but this time, we use the this pointer help resolve ambiguity between local and member variables.

```
void Person::init(const char* name, double popularity)
{
   this->name = new char[strlen(name) + 1];
   strcpy(this->name, name);
   this->popularity = popularity;
}
```

`class` and `struct`

Because of backward compatibility with C, structures can be defined the same as they were in C, even though the class supersedes the functionality of a structure. However, in C++, a structure is really nothing but a class with all public members. This means that in C++, a structure can have methods, static data members, or other features that classes have except for access specifiers. Occasionally you'll see structures used as classes, often as nodes within a linked list. (Check out the `SymbolTable` implementation of Assignment 1 to catch a glimpse!)

For those of you reading the Eckel text, you'll notice that he segways into the object by first talking about structs and then coming around to classes. It's an odd way to present the transition, but fortunately he explains everything very nicely, so you'll have to forgive him.