Review

Overview

This is a summary of classes. It should cover most of what you need to know and more to start in CMSC 214.

Terminology

In this course, I will sometimes use the following terminology. *Instance variables* are the same as *data members*. *Instance functions* are the same as *member functions*. *Class variables* are the same as *static data members*. *Class functions* are the same as *static member functions*. I may also use the term *methods* instead of *functions*.

If you have a variable x of type `Foo`, then x is said to be an *instance* of `Foo`. Each instance has its own copy of instance variables. When instance methods are called, they must be called on instances (i.e., variables). In C++, instance methods always have the object itself (referred to as $*$ `this`) is an implicit argument. Instance methods typically access or modify instance variables.

Class variables (static data members) are basically global variables for a class. You don't need an instance to access or modify them. This is how a class variable looks in C++:

```
class Foo {
  static int x;  // Static data member (class variable)
};

int Foo::x = 3;  // Must be initialized/declared outside the class, usuall
                 // in the header file just after the class definition.
```

Here's a brief example illustrating the differences between class variables and instance variables.

```
class Foo {
  static int s;  // Static data member (class variable)
  int i;
public:
  Foo() { s = 1, i = 1 }
  void add() { s++; i++; }  // these are inlined functions
  void print() { cout << "s = " << s << ", i = " << i << endl; } // inline
```

```
};

int Foo::s = 3;  // Must be initialized/declared outside the class, usuall
                 // in the header file just after the class definition.
```

In `main()`, if you have:

```
int main()
{
   Foo c, d;
   c.print();  // prints s = 1, i = 1
   d.print();  // prints s = 1, i = 1
   c.add(); c.print(); // prints s = 2, i = 2
   d.add(); d.print(); // prints s = 3, i = 2
   c.print(); // prints s = 3, i = 2
}
```

There are two copies of the instance variable i (one for c and one for d). However, there is only one copy of the class variable s. Thus, when both c and d add to s, they add to a single memory location.

You do not need an instance to access class variables (especially, if it's public). For example, if s were public, you could modify it by saying `Foo::s = 4;`.

Class methods are methods that do not use the instance as part of the arguments. They can only access class variables, not instance variables (while instance functions can access both). Effectively, class methods are practically stand-alone functions, just like in C except you get some kind of namespace protection because you must refer to the class method using the scope resolution operator. For example, if a class had a public class method called `run()`, then you can call it with `Foo::run()`.

We won't use class variables or methods much in this course. Nevertheless, any good C++ programmer knows what it is.

Interface vs. Implementation

One of the goals of learning object-oriented programming is to understand how it allows you to control the interface and implementation.

Interface What the user sees and uses. This is an *abstraction*. User sees what designer provides. Often, interface is easy-to-understand, yet powerful.

Implementation The mechanism which makes the interface work. Often complicated. Typically, hidden from the user who only sees the interface.

The implementation is often hidden. Two reasons: first, the complexity of the implementation is hidden so user doesn't have to deal with it. Second, by hiding the implementation, the implementation can change. E.g., cars have become more fuel efficient (implementation) even though the interface (how one uses the car) is the same.

In C++ classes, the interface are the public instance functions (more specifically, they are the prototypes). This is what the user sees and uses. Except who is the user? The user is the person who uses the class to write their own code (which is often you, but certainly could be someone else).

The implementation are the instance variables, private instance functions and the code used within the public instance functions. The user using the classes never sees that. Thus, they are insulated from the complexity of the implementation. They see an *abstraction*.

As you write your own classes (at least, those that you get to decide on the implementation), you should think about the interface. Too often, programmers think about the implementation and the problem to be solved. Object oriented programming is thinking about the object as a kind of tool and deciding how to make that tool useful and versatile. Only once the tool (i.e., the class) has been designed and debugged do you use the tool to solve the problem.

Thus, object design is basically tool design. Just like someone building a house might be better served if they thought about the right tools needed for the job (possibly developing the tools themselves), the same applies to writing code. Design good classes by thinking about the interface first, and then use the objects to solve the problem.

Implicit Instance Functions

Classes with No Explicit Instance Functions

Consider the following class. It has no explicit instance functions (meaning, none are explicitly written in the class definition).

```
class Foo
{
  int x; // private
  Bar y; // private---Bar is another class
};
```

Since there are no explicit instance functions, does this mean you can't do anything with this class? No. There are implicit instance functions (meaning member functions you get for free without writing it). When no constructors are specified, you get: the (implicit) default constructor, the (implicit) copy constructor, the (implicit) overloaded assignment operator, and the (implicit) destructor.

Here's a summary of the behavior of these implicit functions.

	Basic Types/Pointers	Objects
Default Cons	Nothing	Calls default cons
Copy cons	Normal assignment	Calls copy cons
op=	Normal assignment	Calls op= (not copy)
Destructor	Nothing	Calls destructor

A basic type refers to built-in types like int, float, char that are not objects. Pointers are basic types as well (i.e., they are not objects). Objects (or arrays of objects) on the other hand call constructors, destructors, etc. even if those are implicit as well. For example, in the implicit overloaded assignment operator, the overloaded assignment operator is called instance variable. Obviously, if you define it explicitly, you can change the implicit behavior.

Classes with Non-default Constructor

Suppose you now define one constructor, but not the default (empty) constructor. Here's an example:

```
class Moo
{
  int x; // private
  Bar y; // private---Bar is another class
public:

  Moo( int val ) { x = val }; // again, inline
};
```

Once you define a non-default constructor, then the implicit default constructor is gone. You have to explicitly write one if you want to use it. However, you still get the copy constructor, assignment operator and destructor.

When Constructors, Operator= and Destructors Get Called

The default constructor is called when declaring a variable (or an array) without using initializers. For example,

```
  Moo x;  // default cons called
  Moo arr[ 10 ];  // default called 10 times
```

The copy constructor is called in many places, which is why it's easy to get it wrong. First, it can be called when a variable is initialized to an object of the same type.

```
Moo x;        // default cons called
Moo y = x;    // copy cons (NOT operator=) INITIALIZER
Moo z( y );   // copy cons
z = x;        // operator= (NOT copy)
```

It's also called when an object is passed by value or returned by value. However, the copy copy constructor is not called when an object is passed by reference, nor when it is passed by an array (which is a pointer), or pointer, nor returned by reference or by pointer. Obviously, it can be called explicitly by using the `new` operator with an object of the same type as the argument.

`operator=` is called when there is an assignment statement. The only confusion is when students think an initializer is an assignment statement. It's not. It's usually not too hard to figure out when this operator is called.

Like copy constructors, destructors are also called all over the place, and it's difficult to remember when it happens. The obvious time it occurs is when `delete` is called. It's called when you leave the scope of some block of a variable or array. It's called when you leave a function call and there was an object that was passed by value as a parameter. It's also called if an object is returned by value, but only after the statement which caused the instance function to be called (and received the object afterwards) is completed. So, when the destructor is called can be difficult to trace.

Default Parameter Values

Parameters can have default values. If there are n parameters, then the first k parameters are non-default, while the remaining $n - k$ parameters are default where k is between 0 and $n - 1$ inclusive. The default values must always appear at the end of the parameter list (otherwise, ambiguity can result). This creates k different functions with $n - i$ parameters for i from 0 to i.

Initializer Lists

Constructors (and only constructors) can have initializer lists. For example:

```
Moo( int x, int y ) : val( x ), num( y )
{
}
```

Without initializer lists, all the instance variables are initialized either with garbage values (for basic types and pointers) or by using the default constructor. With an initializer list, the appropriate constructor is called depending on what "arguments" are given to it. For example, if `val` were an int, then `val` would be assigned x by normal initialization. If it were an object, then `val` would be initialized by finding a constructor which can take a single int as argument.

Initializer lists are typically a little more efficient because you avoid the step of the default constructor being called. In general, they are not required unless you have instance variables which are references (this is rare, but possible) in which case you MUST initialize them using initializer lists.

Again, only constructors can use initializer lists.

const

`const` can be a rather difficult to fully understand since it can appear in so many places. I will deal with const in the places that are most important.

```
int foo( const int &x, const Bar y, const Bar *z ) const
```

Suppose `foo` is an instance function. x is a const reference, which means you can't modify x (the compiler will complain. y is a const object. Similarly, you can't modify y. z is a const pointer. You can't modify the object pointed to by z, but you can change what it points to. There is a way to describe not moving the pointer (which means the pointer itself is const, not the object it points to), but allowing the object to be modified. Finally, the const that appears at the very end means that the instance variables can't be modified.

It is the responsibility of the compiler to make sure that variables are not modified if they have the const keyword. Variables can be modified in several ways. Either you directly modify it (such as $x++$), or you call an instance function on that variable which is not a const function, or you pass is as an argument where the parameter is not const. As you may noticed, the compiler takes a rather conservative view about when an object can be modified, especially when function calls are applied to the variable.

(If there are any errors, please let Charles know, so he can fix them).