Chapter 11: Classes

Notes

- ❑ When you call functions in a constructor or destructor, their virtual versions are *not* used. This means if you declare a CleanUp virtual function in a base class, have the base class destructor call it, and then over-ride it in all derived classes, only the base class version will ever be called. [see C++ Standard 12.7.3]

Blobs with Class

It always annoys me how books on C++ introduce classes with silly titles like "a touch of class". Ironically, I was bitten by the same bug so I had to do my own variant on it. ☺

A structure variable is a blob of data made up of multiple parts, member variables. C++ also contains a more advanced version of a structure known as a class. A variable created from a class is known as an instance of a class, class instance, or object. Sometimes people go so far as to heave out "an object instance of class X" where 'X' is the name of some class. Regardless, most people just call instances of classes, objects[1].

Structures themselves are blue prints for creating a variable based on the structure. The same is true with classes. A class is a blue print for creating an object. The object is an instance of the class. A 1986 Honda Accord LX is an instance of the 1986 Honda Accord LX plan. You can have multiple cars of this type, but you only have that one type, that one plan.

Blue prints in the form of classes are more complex than that of structures. Not only does it contain member variables, but *member functions*. Members of classes, be them variables or functions, are also assigned *access modifiers*. There is also the possibility of *static members* which are to classes as global variables are to your entire program; i.e. it is a piece of data that is shared between all instances of the class. Inheritance is also a possibility in classes, which is where a class inherits members of a base class. And all of this confusion I am going to unravel for you.

Simple Declaration and Usage

Coding a class is designing a blue print for the objects that will be instances of your class and looks much like the declaration for a structure. An instance, really, is just something that has been created from some sort of plan. All of your simple variables and structures up to this point are instances of their type:

[1] Structure variables can also be known as objects.

```
int x;
```

In the above code, ‘x’ is an instance of the ‘int’ type. The term “instance” is most often used in conjunction with objects of classes, and I think you’ll see it used rarely outside of that context.

A class declaration closely resembles a structure declaration except for the usage of the keyword ‘class’ rather than ‘struct’. Below is an example class declaration:

```
class apple
{
public:
    int weight;
    int worms;
    int redness;
};
```

The member variables are declared as normal variables are and as member variables are inside structures. The other change is the addition of ‘public:’ which sits atop all of the member variables. This is known as an *access modifier*. Every member following an access modifier label has that access type. Thus all of the member variables in the above code are “public”; i.e. publicly accessible. Until this is further explored and explained, all of the members will be public.

You can run through all of your old source code, replace ‘struct’ with ‘class’, and add ‘public:’ at the top of your declaration, and everything will still compile and work as you expected. So to create an ‘apple’ object or instance of ‘apple’:

```
apple a;
a.weight = 10;
a.worms = 1;
a.redness = 2;
```

Creating and using an object is identical to that of a structure variable. Pointers will work the same way where the member-through-pointer (`->`) operator is used:

```
apple *p = &a;
p->weight = 11;
p->worms = 0;
p->redness = 2;
```

Using a class is identical to that of using a structure like you’ve seen. All the same rules apply.

Member Functions

Unlike you've done with structures, classes can contain member functions as well as member variables. Prototyping a member function is identical to that of a normal function except that it occurs *within* the class declaration[2]:

```
class apple
{
public:
    void eat(void);
    int weight;
    int worms;
    int redness;
};
```

The prototype will "absorb" the access modifier labeled above it. Thus in the above class declaration, the '`eat()`' function is "public". There are two ways, or rather two places, to define a member function. It can be either *inline*, which is inside the class declaration, or *outline*, which is outside the class declaration.

To define a member function as outline, you write its definition outside the class declaration. In this method you must specify the class that the member function is a part of. This is done by preceding the function name with the name of the class followed by two colons ('`::`'):

```
void apple::eat(void)
{
    cout << "Munch munch munch ... mmm" << endl;
}
```

Writing in the class name says that the function 'eat()' is a member function of 'apple' and that's where its prototype exists.

Inline member functions are defined inside the body of the class declaration. When defining an inline member function you do not need to prototype it and unlike outline functions, you don't have to specify the class:

```
class apple
{
public:
    void eat(void)
    {
        cout << "Munch munch munch ... mmm" << endl;
    }
    int weight;
    int worms;
    int redness;
};
```

[2] The term 'prototype' is also used to mean 'declaration' for classes (*class prototype*), but I will avoid that for clarity.

There are some severe differences between inline and outline functions, but none of them are syntactical. These differences affect performance more than anything, but there are also some functional aspects. I will cover these at a later point.

Calling member functions is the same as with accessing a member variable. You specify the object (instance of the class), follow with a period, and end with the name of the function:

```
a.eat();
```

It is infeasible to call a member function without specifying the *object*. Do not try to use the member function without an object or by using the class name:

```
eat(); // <-- won't work
apple.eat(); // <-- won't work
```

The reason a function is created as a member is to have implicit access to the object's member variables and functions. That is to say, a member function can use the member variables of an object as if they had been declared locally:

```
void apple::show_redness(void)
{
    cout << "The redness of this apple is " << redness << endl;
}
```

Member variables have "member scope" which is implicit visibility to member functions and explicit visibility outside of that. For a programmer to use the variables of a class, or a structure in fact, they must specify the object as well as the variable:

```
a.redness = 7;
```

The member variable has "member scope" because it is not accessible without specifying the object that it is a part of. Member functions do not need to specify the object to access a member variable because they are already within the same visibility, or scope.

Just as all 'apple' objects have 'redness' member variables, they all have 'eat()' member functions as well. When you call 'eat()' on any instance of 'apple', that function automatically has access to the member variables of that instance. Within the 'eat()' function, you can use the 'redness', 'weight', and 'worms' variables as if they had been declared locally:

```
void apple::reset_vars(void)
{
    weight = 0;
    worms = 0;
    redness = 0;
}
```

The purpose and process of member functions are tied to the class, and instances of that class, they are a part of. There are *few* (though some) reasons that a function should be a member if it has no association with the other members of the class.

Function members of a class are implicitly available, just as variable members, to the definition of member functions. Member functions of an object can be called without specifying the object:

```
void apple::munch()
{
    eat();
}
```

To recap, the code within the body of a member function has implicit access to all variables and functions that are members of the current object. When you create an object and call a member function; that member function has implicit access to the members within that selfsame object.

Constructors and Destructors

Because member variables cannot be initialized when they are declared, as normal variables, classes provide the concept of *constructors*. A constructor is a function that is called after an object has been created. It is primarily used for the initialization of member variables.

There are many kinds of constructors, each for a different situation. When an object is created for some reason, one of its constructors is called. A constructor is simply a member function, *with no return value*, that is automatically called. They can be defined inline or outline, like other member functions, but again they cannot have a return value. All constructors of a class have the same name as the class, but varying parameters.

The most common is the *default* constructor, which is called when you create an object like a normal variable without any parameters, as I have done many times above in my examples of classes. The default constructor has no parameters. Therefore, the default constructor of the class 'apple' would be *prototyped* as:

```
apple();
```

An outline definition of this function to initialize all member variables to zero would be:

```
apple::apple()
{
    cout << "Default constructing an apple" << endl;
    weight = 0;
    worms = 0;
    redness = 0;
}
```

Notice that, because it is a member function, it is within "member scope" and has implicit access to all member variables. In my prototype and definition above I completely omit a parameter list, but it is perfectly acceptable (and equivalent) to put 'void' there. You should not, however, declare a constructor as *returning* 'void' because it may upset some compilers. A constructor simply has *no* return value, not even 'void' (if that's possible … seems silly to me ☺).

This default constructor would be automatically called when creating an object instance of 'apple' as we have done:

```
apple a;
```

Copy constructors are a special breed of constructor that is called when an object is created from an already existing object. This happens when you pass an object by value to a function or when you create an object and initialize it to another pre-existing object:

```
apple a;
apple b = a;
```

In the above code the default constructor would be called for object 'a', and the copy constructor for object 'b'. A copy constructor is defined as taking a reference parameter to an object of the current class. This stands to reason since the object must be initialized to another object of the same type or class. A prototype for 'apple' would look like this:

```
apple(apple &);
```

A definition for this copy constructor might be defined as:

```
apple::apple(apple &src)
{
    cout << "Copy constructing an apple" << endl;
    weight = src.weight;
    worms = src.worms;
    redness = src.redness;
}
```

The reference is to the object that the current object is being initialized too. Far above our 'apple' object 'b' would be initialized to 'a', so 'a' would be 'src' and 'b' would be implicit. As I mentioned, the other way for a copy constructor to be called is when you pass an object by value to a function:

```
void eat_apple(apple ap)
{
    ap.eat();
}
```

If you were to create an instance of 'apple' and pass it to the above function, the copy constructor would be called. Passing by value means to make a copy of the value you are passing in and using it within the function. So, the parameter 'ap' will be created when the function is called and initialized to the object that is passed in. After the function ends, 'ap' will go out of scope and be destroyed:

```
class a;
eat_apple(a);
```

The output of the above with our current default and copy constructors defined would be:

```
Default constructing an apple
Copy constructing an apple
```

Destructors are the opposite of constructors. They are called when the object is deleted, but before the memory for it is actually removed. This gives the object the chance to do clean up work. This point is practically moot at this point in your C++ programming, but it'll be useful later on. A destructor is a member function that has no parameters and no return value. Its name is the name of the class preceeded by a tilde ('~'). The destructor prototype for 'apple' would be:

```
~apple();
```

We could define it for informational purposes as so:

```
apple::~apple()
{
    cout << "Destroying apple object" << endl;
}
```

For objects created locally, they will be destroyed when they go out of scope. So if you created an 'apple' object in 'main()', it would be destroyed at the end of that function. When the object is about to be destroyed, its destructor is called automatically.

Constructors and destructors cannot be called explicitly; they exist solely for automatic initialization and cleanup of objects.

Author's Preference: If you want to be able to call a constructor explicitly, just define another member function that does all the work. Call that member function from within your constructor and you can also call it explicitly whenever you want to.

Custom Constructors

It is possible to create a constructor that takes any amount and type of parameters. As long as this constructor doesn't amount to a single reference parameter of an object of the same class, this would be a custom or explicit constructor. A constructor of this type is only automatically called if the specified parameters are passed in when an object is created.

For example, the ‘apple’ class might contain a constructor that allows the programmer to specify a weight to be stored in the ‘weight’ member variable. This might be declared as:

```
apple(int);
```

We could define it to initialize the ‘weight’ member of our class like so:

```
apple(int new_weight)
{
    cout << "Weight constructing apple to " << new_weight
         << endl;
    weight = new_weight;
    worms = 0;
    redness = 0;
}
```

In order for this constructor to be called, you would have to specify an ‘int’ value within a parameter list (i.e. enclosed in parenthesis) when creating an ‘apple’ object. This is done by placing the parameter list after the name of the new object:

```
apple a(10);
```

The output of this, bearing our custom “weight” constructor, would be:

```
Weight constructing apple to 10
```

Notice how specifying the constructor and its parameters is remarkably similar to calling a function. A constructor is a function deep down so it shouldn’t be too surprising. In fact, you can specify the default constructor by providing an empty parameter list:

```
apple a();
apple b;
```

Both of the above creation statements are identical. See, even though a class acts like it is a normal variable type, it is still a complex one underneath. Creating ‘b’ makes the ‘apple’ class look more like a built-in type which some programmers strive for.

Custom constructors can contain any number of parameters as well as parameter defaults. There is really no more examples I can give without being redundant; try creating some constructors for you to experiment.

Initializing Member Variables

Member variables can be initialized in a special way within the constructor, which is another thing that sets constructors apart from normal functions. To initialize member variables without assignment within the constructor, follow the parameter list in the definition with a single colon. After that list out each member variable you want to initialize separated by a comma. Following the name of each variable you should have an opening and closing parenthesis containing the initializer value:

```
apple::apple() : weight (0), worms (0), redness (0)
{
    cout << "Default constructing apple" << endl;
}
```

This allows you to initialize member variables before they can ever be used, much like how local variables can be initialized when they are created. In essence, you are doing the same thing here. The object is created and before anything else, the member variables are initialized.

Inheritance

Members of a class can be *inherited* from a base class. In the hierarchy of inheritance, a class that inherits from another class is a child or *derived* class. The class that a child class inherits *from* is a parent or *base* class. A child class can use the members it declares as well as the members declared in its parent.

It's a little like your genes. Your dad has black hair and you inherit it from him. You have your own attributes as well as those that you *inherited* from your parents.

The first step in utilizing inheritance is creating a base class. After that, declare the child class and after the name of the class put a colon followed by the name of the parent class:

```
class child : parent
```

For example:

```
class fruit
{
public:
    int weight;
};

class apple : fruit
{
public:
    int worms;
    int redness;
};
```

The 'apple' class in the example above has *three* data members. The first two are 'worms' and 'redness' which it declares explicitly, and the other is 'weight' which it inherits from 'fruit'. An object created from 'apple' can use all three of these data members. An object created from 'fruit' can only use 'weight' because that's all that's been declared there and it doesn't inherit from anything else:

```
apple a;
a.weight = 10;
a.worms = 0;

fruit f;
f.weight = 10;
```

Data isn't the only thing that can be inherited, functions can as well. Remember that a member function can only access members declared in the current class as well as any derived classes. A member function will not be able to access members in a derived class, because it can't know what classes derive from the current one.

Make sure you keep the classes you declare separated in your mind. A base class will know nothing about the class that inherits from it. Consider the 'fruit' class above; it doesn't know about 'apple', nor does it have any access to anything declared in it. All 'fruit' knows about is 'fruit' and any class that it derives from, which isn't anything in this case.

Base classes can be declared to take away redundant members. Declaring the class 'orange' and 'apple' with identical members would be a case of this redundancy. However, you can declare 'fruit' and place all of the common members there and have 'orange' and 'apple' inherit from it:

```
class fruit
{
public:
    int weight;
};

class apple : fruit
{
public:
    int worms;
    int redness;
};

class orange : fruit
{
public:
    int orangeness;
};
```

Blarg.

Overloading

Members in a child class with the same name as another member, even a derived one, are said to be *overloaded*. Only functions can be overloaded because they can be identified uniquely not just by their name, but by the parameters they accept. Overloaded functions must have different parameters than the functions they are overloading except when overloading a derived function. The return value does not matter in the case of overloading, because it does not have to be used so it is useless in determining which function is being called.

For example, if you created two functions in ‘apple’ called ‘eat()’ and both took no parameters, how would you know which one was being called? Even if the two functions returned different value types, how would you know? The return value of functions does not have to be utilized, so it is an unreliable way to distinguish two functions. Parameters, on the other hand are very unique.

If one of our ‘eat()’ functions accepted and ‘int’ value and the other had ‘void’ for parameters, it would be a simple matter to distinguish the two. When calling the function ‘eat()’, the caller would specify an ‘int’ parameter to use the first version or no parameters to use the second:

```
a.eat(1); // call apple::eat(int)
a.eat(); // call apple::eat(void)
```

There can be any number of overloaded functions as long as their parameters warrant sufficient uniqueness. This is a fairly gray area so treat overloading with care. Certain variables like ‘int’ and ‘long’ are so similar that the compiler will likely not let you overload a function based on them as differentiating parameters. In that case the compiler will warn you with the word “ambiguous”. This fancy word simply means “too similar to the point of being indistinguishable”. If you had two ‘eat()’ functions and one took an ‘int’ while the other took a ‘long’, which one would be used when you called ‘eat(1);’?

Author’s Preference: Do not try overloading based on different integer types (‘int’ versus ‘long’, etc.), except for with ‘char’. Number literals are never interpreted by the compiler as ‘char’ types and character literals are never interpreted as any other integer type. So they are a safe distinction. Also, do not try to overload based on ‘double’ versus ‘float’ as it will likely cause ambiguity warnings.

When a base class provides a function, you can overload it in a derived class with the same parameters. By this you are, in effect, “replacing” the base class function with one of your own. A function on the immediate class is always used over a parent class’s function. So, even if the function prototype is identical, it will not be ambiguous. The immediate class’s function will be used.

Consider the ‘eat()’ function on ‘fruit’ if it contained methods to simply “consume”. An overloaded function on ‘apple’ might want to do more than this, like “crunch” for example. When you create an instance of ‘apple’ and call ‘eat()’, it will execute the function to “crunch” which is more specific than simply to “consume”.

If your class has a default constructor, as does the class it inherits from, then it is overloading the default constructor. Unless you specify to call the default constructor of the parent class, it will never be called at all. The danger is that the base class may do initialization of variables and because it is never called, the initialization never happens.

Initiating the base class's constructor is the same as initializing a member variable in the constructor. The name of the base class should follow the constructor name after a colon or comma in the case of being part of a list of initializations. The parenthesis may contain no parameters to call the default constructor or whatever parameters to specify another custom constructor. Just as you would call a specific constructor by designating certain parameters when creating an object, you would do the same when calling a base class constructor.

So, if the 'apple' class constructor wanted to call the 'fruit' default constructor it might do this:

```
apple::apple() : fruit()
{
    // constructor stuff.
}
```

But we mustn't forget the initialization of the members of 'apple' class. The call to the base constructor can be part of this list:

```
apple::apple() : redness(0), worms(0), fruit()
{
    // constructor stuff
}
```

If a custom 'apple' constructor was called to specify the 'weight' member, that would need to be propagated up to the 'fruit' class; assuming that it also has a custom constructor to accept a 'weight' value:

```
apple::apple(int new_weight) : redness(0), worms(0),
 fruit(new_weight)
{
    // constructor stuff
}
```

It is possible to overload functions outside of classes. These exist in a global scope, rather than a class scope, but the same rules still apply. To overload a global function it must differ by the amount and types of parameters, not the return value.

Class Namespaces

Declaring a class also creates a new namespace (see Chapter 4: Flow Control) under the same name. Global and local namespaces have no names, ironically, but member namespaces do. The name of the member namespace is that of the class the member is a part of. Thus, the 'redness' member of 'apple' is within the 'apple' namespace.

If a class contains a member with the same identifier as a global one, you cannot use the global one without specifying the global namespace using the scope operator. The name of the global namespace is blank, so the expression is only the double-colon (scope operator) followed by the identifier:

```
int redness = 5;
void apple::print_redness(void)
{
    cout << "member redness = " << apple::redness << endl;
    cout << "global redness = " << ::redness << endl;
}
```

When inside a member function, the *implicit* namespace is that of the class itself. By implicit I mean that when you specify an identifier, the one in the class is used first before the global one. So, in the example above, I could have removed 'apple::' from output of the member 'redness' and it would still function the same.

The global namespace is cluttered quickly because of its uniqueness and using the scope operator to specify it is quite common when inside member functions. Class namespace is not usually specified within the same class because it is implicit. It is necessary though if you want to use a member of the current class within a member function that contains a local with the same name. Consider:

```
void apple::print_redness(void)
{
    int redness = 0;
    cout << "redness = " << redness << endl;
}
```

Will the above code print the value of the member variable 'redness'? No, the output will always be:

```
redness = 0
```

This is because of the local variable 'redness'. Local variables take precedence over *all* namespaces unless otherwise specified. Thus, to force the output of the member variable's value over the local you would have to specify the class namespace:

```
void apple::print_redness(void)
{
    int redness = 0;
    cout << "redness = " apple::redness << endl;
}
```

When a class inherits from another class, all of the identifiers from the parent class are also present in the child class. If the 'apple' class inherits the 'weight' variable from the 'fruit' base class, then you can specify the 'weight' variable inside an 'apple' member function by either of the following:

```
apple::weight
```

```
fruit::weight
```

It is known that since ‘apple’ inherits from ‘fruit’ that both of these refer to the ‘weight’ member that was inherited. However, if ‘apple’ defined its own ‘weight’ variable then they would refer to two separate things. Specifying ‘apple::weight’ would refer to the variable declared explicitly inside the ‘apple’ class; whereas ‘fruit::weight’ would refer to the variable declared inside ‘fruit’ that apple inherited from.

Not specifying a namespace for a variable that exists in both the current class and the base class will cause you to access the variable in the current class. When you declare a variable for the current class you are basically putting it above all others. The only way to access the other variables of the same name, be them global or base class members, is to explicitly specify the namespace they belong to.

You can specify any namespace with the scope operator, but it will *only* work for the ones you have access to. If you specify a member function under a class namespace that is not affiliated with the current one in any way, it will fail[3].

Although I have used variables in all of my examples, all of this holds true for functions as well. The one exception is local functions which are not allowed. If you overload a base class function, you can access with the base class namespace using the scope operator. One reason for doing this is if the base class provides basic functionality that your overloaded function extends upon. You could call the base class function at the end of your overloaded function.

Static Members

Members that exist for all instances of a specific class are known as *static members*. Static member variables act like global variables, but are limited to the scope of the class they are declared in. They are much like the static variables of functions. They exist for the duration of the program rather than being tied to the lifetime of a specific object. Member functions can also be static, not just member variables. The unique thing about static members is that they can be called by specifying the class, not necessarily an instance of that class.

To declare any member, function or variable, as static, simply precede it by the keyword ‘static’ inside the class declaration:

```
class apple
{
public:
    static void print_count(void);
    static int count;
}
```

[3] Unless the member function of the other class is static.

Nothing more needs to be done to the functions, but static member *variables* must now be initialized. The initialization marks the variables *definition*. Whereas normal member variables are defined when an object of the class is created, static member variables must be explicitly initialized somewhere. This initialization must be somewhere in the global scope, i.e. outside of any function.

The initialization of a static member variable looks like a normal variable declaration except that the name of the variable is preceded by the class namespace and scope operator:

```
int apple::count = 0;
```

A static member variable *must* be initialized and not simply declared outside of the class. This gives the variable an initial value. Basically a static member *is* a global that exists within a specific class namespace. So, to create it you need to declare it inside the class it will be a part of an initialize it in the global namespace, but *specifying* the class namespace the identifier is in.

As previously said, a static member variable, like any static member, is accessible without specifying an object. Instead, you must specify the class namespace that the static member is a part of using the scope operator. So, inside 'main()' if we were to print the value of 'count' (inside 'apple') using 'cout', it might look like so:

```
cout << apple::count << endl;
```

We don't need an instance because 'count' is really just a global variable that is under the 'apple' namespace umbrella. It is not tied to any particular object. Accessing 'count' through an object of 'apple' will yield the same result as accessing it with the class namespace and scope operator:

```
apple a;
a.count == apple::count;
```

Because a static member belongs to no particular instance, *static member functions* can ***not*** use non-static member variables, sometimes known as *instance members*. These functions are the same as global ones except within the class scope. They have no relation to the non-static member variables *or* functions within other than by being under the same namespace umbrella.

Static member functions can, however, access static member variables of the same class (or base classes) implicitly without needing the scope operator. This is because the default scope inside a static member function is that of the current class:

```
void apple::print_count(void)
{
    cout << "There are " << count << " apples" << endl;
}
```

Author's Suggestion: Using all static members you could create a class that simply contains helper functions and variables. These helper functions are usable anywhere and without an instance of the class, but they do not clutter the global namespace.

Access Modifiers

Labels within the class declaration, known as *access modifiers*, determine how members beneath them can be accessed. There are three access levels that can be specified: public, protected, and private. This first access level you've already used up to this point.

Public members can be accessed by anything. That is, they can be accessed from outside the scope of the class. Protected and private members, on the other hand, can *only* be accessed by functions acting within the class scope. Private members take it a step further and only members of the *current* class have access. Private members are not usable by derived classes. Privates are basically extreme versions of protected variables.

The default access modifier for all class members is private. That is, if you do not specify 'public:' or 'protected:' (or 'private:'), the default access level is private:

```
class apple
{
    int this_is_a_private_member;
public:
    int this_is_public;
};
```

Access is specified at the class level; i.e. per class *not* per object. That is, an object of 'apple' has access to the protected and private members of any other object of apple. This goes for static member functions as well.

For example, the copy constructor of 'apple' is passed a reference to another object of 'apple'. Inside this copy constructor you can access the current object's protected and private members, as well as the protected and private members of the reference passed in:

```
class apple
{
    int privatemember;
    apple(apple &src)
    {
        privatemember = src.privatemember;
    }
};
```

And even static member functions can access these v.i.m.'s (very important members ☺).

Friends

Classes and functions can be declared as *friends* to a particular class. Simply put, a *friend* has access to protected and private members. Friend declarations can be made anywhere inside the class declaration. The syntax is as follows:

```
friend <identifier>;
```

So if you wanted to make the function 'Eat()' and the class 'orange' friends of 'apple', it would look like so:

```
class apple
{
    friend Eat;
    friend orange;

    // other stuff in 'apple'
};
```

A friend still has to access the members in the same way, the difference is that it is not restricted to public members. So if you created an object of 'apple' and 'weight' was a private variable, you wouldn't be able to access it unless you were inside a friend function or a member function of a friend class.

Friends get the same access as the class they are friends with. That is, if a function is a friend to a derived class then it has access to the base class members in the same way the derived class does. Private members of the base class cannot be accessed by the derived class *or* by the friend.

This

There is a special variable called 'this' which exists within the body of every *non-static* (i.e. instance) member function. This variable, no pun intended, is a pointer of the current class to the current object. You can use it as you would any other pointer, but you *cannot* assign it any new value. It is a read-only variable.

The '*this pointer*', as it is called, is used to refer to the current object for whatever nefarious purposes you can conjure up. I promise that as you develop more advanced programs in C++ you will find a purpose for using the 'this pointer'. One reason might be if you had a static member function that accepted a pointer of the current class as a parameter. You could call that function and specify the current object using the 'this pointer'.

Since it is a pointer to the current object, you can use it to access members the same as you would members of a structure:

```
void apple::print_redness(void)
{
    cout << "redness = " this->redness << endl;
}
```

Remember that it exists implicitly inside every non-static member function.

Structures are Classes

Structures are actually identical to classes with one exception: members in structures default to public rather than private. Thus you could replace every ‘class’ keyword in my and your programs with ‘struct’ and nothing will change.

Author’s Preference: For simple compound data types that only have member variables and no member functions, I use ‘struct’. For all data types that use anything other than plain vanilla member variables, I use ‘class’. This trend is fairly common among other programmers and purists will yank out your hair if you start using ‘struct’ instead of ‘class’ for anything else. ☺