Module 15 Week 8 Object Oriented Programming Concepts

In this Module 1.2

So now that we can talk about object oriented programming, let's get into what we can do in this module. We'll give a definition for an object and a class, we'll talk about the concept of encapsulation, we're talk about how to create a class, enforcing protections and access, mechanisms for class accessors and mutators, and we'll talk about the constant modifier. We're also going to talk about how to guarantee that an object is created properly. We'll talk about using operators on classes and we'll talk about classes that contain dynamic memory. We're also going to get into concepts of inheritance and polymorphism. And we'll talk about dynamic binding as well as pure virtual functions.

Object Orientation 1.3

The idea of object orientation came out in the early 1980's because we had an enormous number of functions and we had data not associated with the functions. So programmers would have to pass a large amount of data to a function to perform a specific task on that data and if the data got disconnected from each other, from the different parts of the data, we would have a loss of information. So what the concept came about was to put all of this data into one capsule, encapsulation, and have functions that can work on that data as a single atomic entity. So we can create in code something that exists in real life which contains enormous amount of information, enormous number of pieces of data, and we can combine them and have functions that operate on that data at the same time. We use classes in C++ to protect the information so that it can't be changed from the outside world. So from anything outside the class it wouldn't have unrestricted access to the data inside the class; we only have access from the functions inside the class. It allows us to do multiple atomic operations inside a single function that has to change lots of data.

So it's not just asking a user or a programmer to make one change then a second change, because we know that the programmer invariably will do the first change but forget to do the second change. Now we can create a function that solves the program to make both changes. It enforces the idea that the designer of the class knows best about what to do, rather than having a set of documentation which tells a programmer outside what to do. So the model that I like to employ, the model that I like to use to visual this, is that of a card shuffler inside a Vegas casino. So if you've ever been to Las Vegas you've seen these card shufflers and what they do is they continuously shuffle these cards, the deck of cards. And when new cards come in, they get added to the shuffle so that nobody can count the cards and know what's gonna come out next. But the point is that they're a single contained item, and what happens inside is sor5t of a black box. We don't know what happens inside. But as the dealer puts more cards in and takes more cards out, what is happening inside is controlled by the designers of the card shufflers.

Class vs. Object 1.4

So first it helps to define what the term's class and object are. and for our class we consider that a framework which is used to build multiple objects which are similar. So for example you might think of a blueprint for a house as a class. The idea is it's the framework that we're going to build the object around; it tells us everything about what's going to be inside the class but it doesn't tell us what the values of the data are for example. The object is just an instance of a class; it's a class instantiated. So when we create one of these variables, we have an object and for that. I'll use the example of a house or

a cookie, so going back to our previous examples of the blueprint for the house with a cookie cutter for baking. We can talk about the object being the House or the cookie and of course we don't try and live in the blueprint, we want to live in the house and if we don't want to eat the cookie cutter, because it doesn't taste very good, we'd rather eat the cookie.

Encapsulation 1.5

The idea that we want to employ here is one of encapsulation: we don't want anybody outside the class, outside the class creation, to be able to access data inside the class. So the idea is that code written as part of the class has a greater access than code which is not part of the class. For example main, main shouldn't have access to some of the private information inside the class. And the idea of encapsulation allows us to set levels of protection so that programmers that aren't involved in the creation of the class are not going to have access to certain items. We can give them controlled access to items that we want them to be allowed access to, and then we can strip their access so they can't have access to the items that they shouldn't have access to. So, for example going back to our house and blueprint model. The architect should be able to change anything on the blueprints of the house, the designer of the class should be able to change anything inside the class, but the plumber who's installing the toilet shouldn't be able to change where the location of the toilet is or else the bathroom door might not close, for example. So we want to create a model where we can protect certain parts of the class and allow public access to other parts of the class.

Creating a Class 1.6

Let's get down to the syntax of how we can create a class. First off the class is an abstract data type, so we're creating a new data type for C++ to know about. We can create this class as a data type which is a more complex data type made up of simpler data types. And we're going to get into this a lot because we can make them very, very complex. Of course we'll use the C++ keyword class, which means incidentally that you can never use the word class as an identifier for anything inside your programs. You can't create a variable named class. Here's a quick view of what a date class might look at, and we know that a date is actually just three integers; it's a day, a month, and a year. So here we're creating a new data type for the storage of three integers, atomically, so that they go together. The year and the month and the day are all associated, so that for example, your birthday is actually three different numbers associated with them one another: the day, the month, and the year. And we need to have all three contain together; hence the idea of encapsulation.

Enforcing Protection 1.7

The previous code we looked at was a really simplistic view of a class. And the problem with the simplistic view of a class is that it's storing three integers, and integers, the domain of an integer is negative two point one billion to positive two point one billion. Now I don't know what it means if the day is negative two point one billion or the day is positive three hundred thousand, it really doesn't make sense because in our model in our world, the day can't be a number outside the range of one to thirty one. So rather than allowing uncontrolled or what we call public access previously, we should really protect that data and we should mark the data as being private. Now one thing to note is that if we leave off the notation of private or public, then by default C++ assumes that what we want is private. So we can modify the class now to change that public keyword to be a private keyword. And now nobody outside of the date class, can access day, month, or year. And that means change it print it do anything with it; nobody has access to those three variables.

Accessors and Mutators 1.8

Well now unfortunately since nobody has access to day, month, and year; nobody can store anything into day, month, and year. Nobody can get out the day, month, and year. So what we've done is create a fairly useless class, because day, month, and year are completely inaccessible by anybody. What we really need here are something called accessors and mutators, less formally known as getters and setters. So the accessors are used to get information out of the class and the mutator are used to put information into the class, and that doesn't mean uncontrolled access, what it means is that we can restrict the access so that the values stored in day, month, and year are going to be reasonable.

So we create three functions here that will act as the mutators; so we've got the set day, set month, and set year function, and what I've done is I've created set year as a function inside the class defined inside the class, and set day and set months defined outside of the class. I'm only doing that for demonstration purposes. There's really no difference in the code for creating it inside the class versus creating it outside the class; they both work exactly the same. But what I'm careful to do is not create a lot of code, not to write a lot of code, inside the class because it tends to make things a little bit less easy to read. So in this case, all three functions are members of the class which means that they have access to all of the private information. So this is the idea of encapsulation that members of the class will have access to the private information inside the class. And here what I can see from the set day function, if you take a close look at it is, it restricts the setting of the day variable to make sure that the day does not go outside the range of one to thirty one. Now you'll notice that I have not passed in any other variable besides the new value that I'm trying to set and I have access directly to the day variable inside the class. As a member of the class, I have access to all the data members inside the class and all of functions inside the class and I can do that directly without any extra definitions. So the set day and the set month are just definitions that I've put outside the class and to do so, I need the scope resolution operator. But you can see that the code keeps track of making sure that the day and the month are reasonable values, so that we can't set something that's completely outside what we normally expect.

Accessors and Mutators 1.9

Now that we've got the mutators, we can change the data but that doesn't give us any access to actually view what's in there; so we can store it but we can't change it. If we are going to create a function that is going to access information inside the class but not make any changes to the to the data inside the class, then we really need to mark that function as being a const function. what a const function means is that the object from the start of the function to the end of the function will not change. and it's a protection mechanism to make sure that we don't accidentally make any mistakes by changing variables. So the same as we would use const during our previous programming samples, we now can use const to protect our objects from accidental damage by ourselves, the programmers. Now there is one thing to take into account when we have const functions. If an object is const-ified, so if for example we were to pass an object to a function and we will pass that object by const reference which is very common to do, then the only functions we have access to inside that function is a const function inside the object. So here now if the object is const-ified, the only functions that we can call are const-ified functions inside the object. It's a protection mechanism from C++ to make sure that we can't do more than what the function can do.

Other Useful Functions 1.10

So now that we've got data that we can store inside the function inside the object and we can get data out of the object, then perhaps we should have some other useful functions. It's really common here in the U.S. that we might print a date in the term, in the form of day slash month slash year. So we could

add a function inside our object inside our class where we can display the date in the form of day slash month slash year. So you can take a look at the code for that; it would be a very simple function to write. Again it's const-ified because it's not changing any of the data. We might also write a function to validate, to check to see if the date is valid, and it will return true or false. For example, if somebody puts in February thirty first of two thousand and sixteen, that shouldn't be valid because there really is no February thirty first; so that might return false if we were to have a validate function on that. We could also check to make sure that the leap year is valid and if we need a leap year calculation; we need to check which years are leap years and which years are not leap years that may require another function. And in that case the calculation of a leap year may not be a function that we want to have allowed access to from the outside, which would make that function a private member function. Really what I'm getting at here is the possibilities are endless. What you decide to do with a date class might be very different from what somebody else decides to do with a date class, but you can do literally anything you want with it inside C++.

Creating and Working with an Object 1.11

So we can create objects just like any other data type. it's now a new data type that we have access to and we can create and we can use. Working with an object requires the use of the dot operator, and you may have seen this before in your work with structs but here it's much more common to use it. Remember that we're not going to have any access to the private information, but for example if we want to set the day, the month, and the year, we can have this code to show you how to set a date to for example the sixth, set the month to August, set the year to one nine hundred ninety one and then print out the fact that this is a very important date. And if you're interested, google that and see what date it is.

Constructors 1.12

One of the problems that we're going to run into is that if we just construct an object like we did before and it's a Date object; what are the values of day month in year immediately after the construction of the Date object, they're garbage, they're garbage values. We don't know what their values are and they may not even be valid so it's very possible that we may have a large number for the day, like negative eight hundred thousand and it's now here are stored inside, what we would classify as an acceptable date object, because we created it. So what C++ allows us to do is make sure that we always know that valid values, even if we just created that object, that there are valid values in it. So C++ creates thing called a constructor, and we can create the constructor where it runs code that we want so that we can set the values appropriately for whatever the default date is. So the default constructor is a function which is given a name which is exactly the same and it is case sensitive, it's exactly the same as the name of the class. There's no return type, so it's not void or anything like that, it's just literally no return type, we don't list anything and it doesn't have any parameters. So we can see from the code that what we've done here is create a constructor where we're setting the values of day, month, and year, to the appropriate to what we classify as an appropriate value, without any interaction by the user. All the user has to do is create one of these Date objects, by default, and it will set those values to what we consider the default values.

Constructor Parts 1.13

Here I'd like to demonstrate that we can actually do this two different ways. In the previous slide, you saw the method that we used was creating a value for the day, month, and year, and what I'd like to show you here is that we can use what's called the member initialization list. So the constructor, if we

put a colon at the end of it and then we have a list of the data members, and we don't have to have all of them we can have some inside the member initialization list and some initial-ised inside the code. This allows us to construct the member variables using whatever values we'd like. But I'd also like you to take a look at the other method, which is creating the data inside the code for the class, without the use of a member initialization list. At this point it's really up to you which you like to do. I just want you to see both. Later on there will be a situation where we can only use the member initialization list, but for now it's entirely up to you which you'd like to use.

More Constructors 1.14

So we can also have constructors that take, for example three integers. We might have one that takes a day month in year, sets those values. We can also have a constructor that takes just one of the values and constructs it based on the Unix epoch, in case you're interested in constructing it based on that. But what we can do now is inside main, we can now create a date with those three variables so that our code is a little bit easier. So if you remember back couple slides ago, the previous code, we set the date to August sixth one thousand nine hundred one. We had to create the date first and then set each of the three individual values. Now we can take those four lines of code and compact them down into one/ where we create the date and set the values all at the same time. You can create as many constructors as you like; each has to have a unique set of parameters. And you have to remember that even though we have a default constructor, only one constructor will ever be called. So whichever constructor is chosen to be called, that's the only one that will ever be called; it's not like one is going to call the default constructor and then go further on to do some extra work. Construction only happens once and we only call one constructor.

An Important Pointer 1.15

Now that we've created an object, for example the date object, we have to take a look into the object itself a little bit more. And C++ has a really interesting structure that it creates inside every object and it's called this: this is actually a pointer which points to the object itself. It's sort of like a pointer back into itself. And this pointer points to what we call the calling object, so when you make a function call like a dot set year, the this pointer will be pointing to a inside the set your function. So it's really useful in a lot of cases, for example, if you created a person class and you wanted to have a recognition that the person could be married for example. Then each person would have a spouse pointer, a pointer to another person so speak, and when you get married, of course one person marries the other person but then that second person also marries the first person at the exact same time. So we would have a function call like a dot marrying B, in which case A's spouse pointer should point to B, and B's spouse pointer should point back to A. And the only way that we can get a pointer to a is with this, so the this pointer is needed in a lot of situations where we need to recognize that we have some pointer back to the object that we're working in right now.

Operator Overloading 1.16

C++ allows us to do something that most of the languages have eliminated; they don't like this anymore. Java has gotten rid of it, Python never had it to begin with, actually Java never had to begin with, to be honest. What we can do with C++ is what we call operator overloading, and operator overloading allows us to take two objects, and for example add them together. We can operate on two objects or on an object, using what we would consider a normal operator.

So there's three different types of operators: we have unary operators which only work on one object one operand. A unary operator would be like plus plus, we would say A plus plus; there is no other object that we're working with. A binary operator works on to operate operands, so two objects, A plus B, for example. And the majority of the operators that we're going to work with are in fact binary operators. So we have all the regular math operators, plus, minus, multiplication, division, module, zero. We have the compound operators, which are things like the plus equals, and the minus equals, and so on and so forth. We have the comparison operators, which are the less than, the greater than, the output, the input operators, and then we also have things like the square brackets operator or what we call the array index of operator.

There's also a category of operators called the ternary operators, and in fact C++ only has one of these. It's called the conditional operator and if you haven't come across it, it's horribly confusing and we don't often like to use it; but it is there and some people like it. So you can have the turnaround operator. Really when boils down to it operators are nothing more than functions, and they're just a strange way to use functions. What C++ actually does is it takes that A plus B and it says, I don't know how to do A plus B, but I know how to make function calls. And it converts the function; it converts the operation, into a function call. What it converts the function call to depends on how we as the programmer decide to overload that operator. If we overload the operator as a nonmember, we'll talk about this, the operator the function call would look like operator plus A B. So the A plus B function call would be operator plus A B; it would take two parameters. It's a binary operator, it's overloaded as a nonmember, it takes two parameters. However if it's overloaded as a member, then the this object becomes one of the operands and so we only need one parameter. So, the equivalent function call if we're a member, is A dot operator plus B. Here we only have one parameter because the this object, the calling object, is the second, is actually the first operand. Operator functions can be either member or nonmember and we're going to talk of a lot about how to choose which to use. And some operators change the data inside the objects; some return a new object, so it's important to look at what the operator is supposed to do. A compound operator like the plus equals operator should actually change the calling object. Whereas the plus operator A plus B, doesn't change A or B, it in fact return something entirely different. So you have to take into account what the operator is supposed to do, and then you've got to figure out how to write it.

Operator Restrictions 1.17

C++ has some restrictions; it's not open. There are some operators that can't be overloaded. For example the dot operator; you can't change what it does, it's known as the struct member union select operator and you simply cannot change anything about it. The scope resolution operator, the one that goes between the name of the class and the name of the function when we're defining it outside the class, you can't change that there's nothing you can do with that. There's a couple more that you would probably never try to change; the Star dot which is the de-reference and member select. The size of which is very rarely changed; very rarely even considered, very rarely used. And the ternary operator, you can't change what the ternary operator does. So those are some that are just completely off limits. You can't change anything about them; they are what they are.

There are some that are restricted that you can overload them but you have to be a member in as an to overload this this operator and that's because the operator has so much involved in what it's going to do inside the class, that C++ says you have to be a member of this class in order to overload it. So these are functions that can only be overloaded operators that can only be overloaded as member functions. So the assignment operator and the array index of operator can only be overloaded as members of the

class; you have no choice in that. There's also some that can almost, not be overloaded as a member: the input and the output operators, or technically they're known as the bit shift the left and bit shift right operators, but we all call them the output in the input operators. These operators can be overloaded as a member of the class because if you look at the equivalent function call, if we said something like C out arrow arrow A then the problem is that this would be C out dot operator arrow arrow A. So you wouldn't be making this a member of your class, you would be making this a member of the class which has the object C out, and that's called an O stream. And you're not going to overload override the entire Ostream class; you're going to recreate the entire Ostream class, and create a new C out object just to overload this member.

So while C++ doesn't technically say that you can't overload this members, overload this operator is a member, it's almost universally accepted that you're going to make this a non-member of the class. So those are the only restrictions; any other operator that you can think of, you can't create your own you can't change the precedence of any operator, but any other operator which C++ has you can overload.

Choosing Member vs. Non-Member 1.18

A little while ago I mentioned that you could either have an operator overloading as a member or nonmember. So what are the benefits of having one versus the other; why do we have the option of doing both. Well first off, the key thing to remember of course is that members have access to private. So if we're a member of the class, if we overload this operators a member of the class, then we can directly access the private information. And in a lot of cases we have to overload it as a member because of those restrictions, because we're not going to create functions to allow everybody to access the private information; we might not have an accessor function so we want to have a limited access to the private data, yet we want to have these operators overloaded. In that case the only choice is to make it a member.

We also have the option, and this is a little bit strange, but we also have the option of listing a non-member function, as a friend, if inside the class we write the signature, which is the prototype of the function, we write the function signature and before we write friend. Then we're not saying that that function is a member of the class in fact it's by definition a nonmember but it has access to private information. So friend functions have direct access to the private information, even though they’re non-members. And this is a little way of sidestepping the private restrictions inside C++. It is used in some cases and, is in fact used often in the input and output operator, but apart from those operators it's not used very often because we consider it side stepping around the rules, so we don't want to use it.

Otherwise there's really one fundamental difference between choosing member versus nonmember, apart from the fact that it has access to private. If we're doing for example A plus B; A plus B is going to work in either case whether we choose to overload the plus operator as a member or nonmember function, A plus B will still work. But if we have a constructor which can construct an object out of constant like an integer, then A plus five or A plus an object plus A constant is going to work if there's a constructor, which we can take five and construct one of these objects. So that would working again in either case if we have that constructor; it'll work if it's a member or nonmember.

But here's where life really gets interesting. If we turn that around and we say five plus A, well of course assuming that we have that constructor that can construct one of these objects out of a constant, five plus A will only work if the plus operator is overloaded as a non-member. C++ has this restriction that it says that because we now are going to require that the left hand side is an object, we have to know that

it's not a constant object. And the plus operator overloaded, if it's a member is going to have to have full access to the objects of the object will have to exist before hand It will not be able to be constructed automatically out of a constant.

So here, the only difference between choosing member a nonmember is that a plus five will work in either case but five plus A will only work if we have a nonmember. Generally speaking this is a small price to pay for over for having access directly to the private information. So personally I usually make things members, but there are a lot of purists in C++ who prefer that you create only those items that are members that have to have access to private. And then create nonmember functions that are based on the use of the member functions. So for example, the plus operator you might be able to overload by using the plus equal operator. So you'd overload the plus equal operator as a member but the plus operator as a nonmember, and then use the plus equal to solve the problem. What you choose is entirely up to you; you're the programmer and you decide.

What to Return? 1.19

Now that we understand how we can overload these operators. we have to take a look into what the operator returns, so what's it going to do when it's done the value that's returned really depends on what the operator is going to do. So the plus operator for example is going to return back the sum of the two operands; it doesn't change either operand we said that before, we're not going to change A or B. If we do A plus B, but instead we're going to return back a new value which is the sum of A and B, whatever that means because we haven't defined what data types A and B are. Here we have a little bit of a rule in that we the return data type, really depends on what's being returned of course.

If the item that's being returned was created inside the function, then the return has to be by value. So in the plus operator we add together two operands A and B. In order to do that we have to in the plus function, in the operator plus function, we're going to have to create a temporary object, store the value that's in A and add the value that's in B, and then return that temporary object. So we've created an item; we've created something inside the function and now we want to return it. The only choice we have is to return by value. But if we're returning something that was passed in as a parameter. So for example, we have the plus equal operator; the plus equal operator instead of returning the just the sum of the two items, now we're expecting that we're returning the left hand side item. So A plus equal B would return A, and that's something that existed prior to the function call. We didn't create it; it existed prior to the function call. So the return line is probably going to look something like return star this, there's another use for this.

And in doing so we recognize that we could create another copy of this object, but the time it takes to create the other copy and the duplications of the data uses more memory is a waste of time and memory. So what we'd rather do is return by reference, because now we don't have to do the copy and do use up the extra memory instead we can just return by reference. Returning by reference is really the preferred mechanism. But in order to do that the object that we are returning has to have existed prior otherwise or returning a reference to something that will no longer exist at the end of the function because it was created inside the function. So if we created it, returned by value; if it was passed into us or existed before, return by reference. And that's what to return.

An Odd Case 1.20

One interesting thing about C++ is it has both the pre and the post increment operator; of course we've used these before. But we need a special case to differentiate between the pre and the post increment

operators because they're both unary operators. So since, if they're members it's not going to have any parameters; how do we differentiate between the pre increment and the post increment operators. Well C++ resolves this by passing an int to the post increment operator; the pre increment operator is just going to change the value and then it's going to return a reference to the existing operator, it's going to return star this.

So the format for that looks exactly like you see on the screen: it's just date ampersand operator plus plus with nothing in the parameter list, returned by reference because we are returning star this. But the posting current operators going to create a copy of the object, it's creating something inside the function, and then is going to change the value of the object, not of the copy, and then it's going to return a copy. So we're returning the original value, or a copy of the original value, and we're changing the value at the same time. Since we're returning by value, we have to make sure that that return is by value. So we're returning a copy; we're returning something that we created inside our function so we have to return by value. So you're going to see it as date with no ampersand operator plus plus, and because this is the posting current operator, we need to pass in that integer; so it's date operator plus plus integer. The integer really has no bearing on what happens inside. It's really just a way to differentiate between the pre-increment in the post increment operator and that's the only way that C++ can solve it. So don't look at the value of the integer; the value of the integers completely irrelevant. Just the fact there is an integer there tells us that it's suppose increment operator.

Classes that contain Dynamic Memory 1.21

We've got an interesting problem that we've got to take into account here, and that is that if the class contains any dynamic memory then there is an issue that comes up with what we call shallow copy. So all classes have a leftover of C's struct era, where they'll have an assignment operator and what we call a copy constructor, a constructor that can copy an existing object. That's really a left over C++ because with structs we were always able to say A equals B and all the data members would be copied over. Unfortunately, because we're in C++ and we are using pointers, where pointers are involved those built in operators are going to copy the pointers instead of copying where the pointers of pointing to. And that's really problematic and this is known as what's called a shallow copy.

So I'm going to show you some code here. What we've got is we've got a class called thing, it's very boring, but it contains a pointer to a value. And the thing object constructor is going to create a new integer on the heap which is going to store that value. So here in main I'm creating thing one and thing two, and yes that is a Dr. Seuss reference in case you get it. And what I want you to see is that thing one has a value that points to the value one, and thing two has a value pointer that points integer two on the heap. So on the heap we've got these two integers, one and two, and we've got on the stack we've got two variables, variable one and variable two, each have a value pointer. And the value pointers pointing appropriately, so on the surface this all works perfectly fine and everything's okay. But when we set one equal to two, when we say one equals two, that has the effect of copying the pointers. And now that we've copied the pointers, we have the value pointers both pointing at the same object on the heap, instead of the same object holding the same value. So if we looked at the value in object one we would see two and if we look at the value in object two we would see two but the problem is that those two pointers are actually pointing to the same thing. And what happened to the variable, the integer that object one was pointing at, it's a memory leak. So here we've created a memory leak; just by copying an object. And we can't allow that to happen; so we've got to find a solution to this.

Three Problem, Big 3 Solution 1.22

Now that we've seen the problem, let's take a look at the solution. Well we're going to need to overload; we're going to need to change copy operations. The copy operations need to copy the data and not the pointers, and that's what is really the solution to this problem. Since we're creating memory in the constructor, we're going to need to also destroy that memory at some point. Now we haven't talked about this yet but if you look back at the code when main ends, the data that was created on the heap isn't ever released; we never call delete for that data. So really what we've got are a couple of problems and the solution to that is what we call the Big Three.

And the Big Three is a set of functions that if we need any one of them were going to need them all, and that's almost with one hundred percent certainty. There are very, very rare situations where we might need one copy constructor, for example, we wanted one copy constructor and not need the other two but it's very, very rare. So we call them the big three because if you're going to, if you realize you need any one of them you better create all three of them. The destructor is just like a constructor but the exact opposite; the destructor is called automatically when the object falls out of scope. So at the end of a function call when the object is no longer going to be used, the destructor is called automatically. In the same way that the constructor was called when you created this thing, the destructor is going to be called when it's no longer in use. We also have the copy constructor, which is useful if we want to create an object based on another object; so we might have a copy constructor for the date class that takes in an existing date. And the assignment operators, the third one, it copies one object to another copy and it does that sort of deep copy.

So here we have to be very careful because in a lot of situations that we're going to run across, what we have to do first is clear out the left hand side object. So we might have to empty everything out the left hand side object. And there is one situation that the assignment operator has to take into account and that is self-assignment. If the user is doing something very, very silly and ends up writing something like X equals X, then clearing out the left hand side object will accidentally clear out the right hand side that we're about to make a copy of. And so that we've lost that information that we had intended to make a copy of. To protect against that the code that we like to use is just a simple if statement it's: if this equal equal the address of RHS. And in doing that what we're doing is testing to see if the this pointer is the same as the thing that we're about to copy, the pointer to the thing that we're about to copy, and if those two are the same pointer then really the pointing at the same object which means we really don't need to do any work we can just return star this. So the big three is the solution to the problem of dynamic memory, and I'll give you the code here to take a look at it. You can see the big three: the destructor, the copy constructor and the assignment operator, are really just a couple of pieces of simple code to take care of what has to be done in terms of a deep copy and in terms of destruction of the object.

Inheritance 1.23

Now we have a good basis for how to create classes and when to create classes, there's something else that we can do in C++ which is really great; and that's called inheritance. Inheritance allows us to create really, really complex classes out of simpler ones; not just with composition, not just with adding elements into a class, but what we can really do here is create what we call the is a situation or the is a solution.

I'll give you a couple of very simple examples: a car is a vehicle, or a circle is an object or, for example, a student is a person. What we're saying is that the car class should contain everything that the vehicle

class can contain. So anything you can do with a vehicle, you could do with a car. The person is the same sort of thing, any information we'd like to store about a person, we'd also like to store about a student. So for storing things like name, and height and age and that sort of stuff for a person, we'd like to store that same information as a student but we also like to store additional information in a student that's not in a person.

So what we're creating here is a larger more complex derived class from an existing base class. So using the base class as just a form so that we can add things and not have to reimplement all the work that was done to create the base class. Now the great part about inheritance is that all the items all the functions, all the data, everything that was in the base class is automatically going to be in the derived class. Unfortunately, we don't copy over the constructors, so the constructors for the base class don't come over to the derived class; we have to recreate any constructors that we might want. But we also have the ability to call those base class constructors so it's not as difficult as it might seem. The derived class, of course, can add any new material that it wants to add; anything that we want to put into the derived class that doesn't exist in the base class can obviously be put in.

The derived class can create new versions of existing functions and then it's called overriding. We can override a function that exists in the base class by creating a new function in the derived class which does similar things which has the same parameter list which has the same name. One thing that we're going to come into play with, that's going to come into play here, is that we are not changing any of the accessors; so public is still public and private and is still private. And if you remember I said that inside the only member functions of a class can access private data; does that mean that member functions of a derived class can access private data? The answer is no: member functions of a derived class cannot access private data of a base class. Obviously, member functions of a derived class can access its own private data, but not that of the base class. So we have to take that into account when we look at accessing private information. The derived classes member functions can’t access the base class’ private data.

Pets and Cats 1.24

Here we've got quite a bit of code and I just want to go over it quite simply. We've got a base class called Pet. and the base class pet, what we're doing is we're creating a cat and a pet combination here. So the base class pet has functions like get name and set name and it asked the pet to speak and figure out what the pet speaking does later. But we also have a cat class, which is based on the pet class. If take a look that colon public pet indicates that a cat is a pet, and that's the derivation; that's inheritance right there. The cat doesn't redefine name and pet ID that were created inside the pet class, but it does add a double, which is the whisker length of the cat. The constructor for the cat class tells the pet class to set its pet ID to ten thousand. So here we have an explicit call to the base constructor and if you look, that form of it looks very much like the member initialization list that we saw so many slides ago. Now we call it the base initialization list and it can only be done this way; you can't put anything inside those curly braces in order to initialize the pet. Now keep in mind if you take a look at pet, the cat will not have access to the pet ID because it's a private data member inside the base class. So the only way to set the pet ID is to use that base initialization list. We also have a function speak, which just prints out meow and set length and get length, which are accessors and mutators for the whisker length. And we have another function called set name which coincidentally looks very much like the set name function inside pet. We’re going to look at that in just a minute.

What if we need to override? 1.25

So we want to look at the set name function because the set name function inside cat is going to do something different then just set the name. We have a rule that says if we change the name of our cats we have to cut off their whiskers; I don't know this is a crazy rule, but that's what it is. And so we have to set that whisker length back to zero any time somebody changes the name of a pet. Now how do we do that? Whisker length is a member of the cat class; only cat can change the whisker length. But name is a member of the pet class and only pet can change the name; so how do we do it? Well, we override the set name function so that if anybody has a cat and sets its name the first thing we're going to do is change the whisker line but we also have to go back and actually set the name because at this point we've overridden the function. This is the new version of the function sequence; C++ will not automatically go and call pets set name if what we're working on is a cat object. So what we do is use the scope resolution operator and indicate the name of the base class function, in this case pet, to indicate that what we're calling is the set name function inside the pet class. If we didn't have this we'd have recursion. If we just said set name; we would be the function calling itself would be calling cats set name function which would be a horrible loop that we'd never get out of. So what we do is we indicate that we're calling the pets set name function and then we pass in the name. We effectively have the solution of solving both problems, setting the whisker length and changing the name, in one easy function.

What if derived SHOULD access base's stuff? 1.26

From time to time, there's going to be situations, of course, where we have to have allowance for the derived class to access the base classes' private member variables. And we can't make them private in that case; so there's no friend possibility here. But what sequels post does create is something called protected.

The protected modifier is added, now we now have public, private, and protected. Protected is specifically for this purpose: it allows a derived member function to access a base class's protected information. But it doesn't allow any access outside of those two classes: outside of the base class or the derived class. So if we're inside main protected in private or fact of the same thing but if we're inside a derived class we would not have access to the private information, whereas we would have access to the protected information. It doesn't change anything Internal to the class; so it's only in the case of inheritance that protected becomes useful because that's when the derived class would exist. And it would have access to those items marked as protected inside the base class.

Polymorphism 1.27

This is a topic I actually really like; it's polymorphism. And what we're saying is that since every cat is a pet, we should be able to copy data between cats and pets. Now if we take a look at main memory, what we're going to see is that inside every cat there is a pet object. Not in reality that there is a pet inside, But what we're going to see is that all the stuff that is in pet is also in cat. So if we have a function which expects to take a pet, it could take a cat because the cat class will have all the items that exist inside the pet class. It's going to have more in this case is going to have whisker length but it's going to have everything that was inside the pet. All the functions, all the data, everything that we should be able to do with a pet we should also be able to do with a cat. So since every cat is a pet every cat is going to contain all of the functions inside pet, though not necessarily the same versions and that's where one of the problems really comes along.

Polymorphism in C++ allows us to copy the data over from a cat into a pet object. And that's what I want to want to show you here with a little bit of code. What I've done is create a pet object and a pet pointer and as well as a cat object and a cat pointer and we're going to look at a couple of examples here. If we take a cat and try and store it in a pet, this always works, this is always allowed; it's a solution known as slicing. So taking a cat object and storing it into a pet in this case, P equals C that will always work. But what happens is those objects, those items, of the pet that are in class are copied over. What happens to the whisker like this; it's just completely lost.

So in our example we'll be copying over the name of the cat will be copying over the pet ID of the cat and we'll store that in the pet object but we can't fit the whisker length, so we just don't copy it. It's very much like when you took an integer and you stored a double in that integer; it's just chopped off the decimal point and threw away all the stuff after the decimal point, didn't do any rounding or anything like that. We can also copy information from a pet and sort of upgrade it to that of a cat, but C++ doesn't do this automatically. C++ won't allow this to be done automatically; what we will have to do is overload the assignment operator. So inside the cat's class inside the cat class, if we overload the assignment operator to take a pet object then we'll get this upgrade possibility because the cat class will know what to do, will know what value to set and whisker length. It will copy over or it should copy over those items that are in the pet manually and then you can have whatever value for whisker length you deem appropriate, maybe at zero maybe not, I don't know. But the point is that if we overload the assignment operator for a pet inside the cat class, we're allowed to do cat equals pet.

The pet pointer being assigned to a cat will always be allowed; and this is the real core of polymorphism this where the magic really happens. Because we have a base class pointer, we can make it point to a derived class object because the derived class object will contain all the functions and all the data that are in the base class. So the pointer can point to a derived class object and we haven't lost anything. It's really important to recognize the difference here between slicing P equal C and polymorphous P pointer equals the address of C. In one case we're taking the cat information in the slicing case we're taking the cat information and storing it into an object that actually is a pet; and the other case we're taking a pet pointer and making it point to a cat. So polymorphism is not working with pets, it's working with pet pointers that are pointing to cats. And that's a huge difference because we have to recognize that the pet pointer is not pointing at a pet; it's pointing at a cat or maybe it's pointing at a dog or a fish or a turtle or whatever kind of pet you want. But the point here is that the pointer can point to any data type as long as it's derived data type of the pet class. All the pet functions, all the pet data is still going to be in those derived class objects so it's allowed. Again we've got to keep in mind that the versions of the functions might be different so that's going to become an issue in the next slide. But for right now we can recognize that everything that's in the base class object will also be in the derived class object.

If we do have that pointer, the only things that we can use are those things that exist in the base class. So we can't change the whisker length via the P pointer, we can't change the cat's whisker length, even though it is a cat object; P pointer is a base pointer and the only things were allowed to access are those things that exist in the base. So we can set the name and we can set the pet ID number but we can't set the whiskered length using in the pet pointer, we need a cat pointer to do that. And one thing that's very important to recognize is that we can never take a cat pointer and make it point to a pet object. We're never allowed to do assignment between the data type pet pointer and the data type cat pointer. Even though they're both just pointers and even if the pet pointer is actually pointing at a cat, C++ will not allow us ever to copy the pet pointer into a cat pointer. So what we're doing here with CPTR equals to the address of P. There's no way to make that work in C++.

Virtual Functions 1.28

If we go back to our previous example of having a base class pointer point to a derived class object; C++ makes some interesting assumptions. First off, by default C++ assumes that a base class pointer points to a base class object. And that is not at all true because polymorphism tells us that we can make a base class pointer point to a derived class object. But C++, for good reason, tries to optimize things and say that if we have a base class pointer we can assume that it's pointing to a base class object. Unfortunately, that means that if we call any functions via that base class pointer, the function version that we're going to be calling is that of the base class even if the pointer actually points to the derived class object. So if we, for example in our previous example if you want to take a look at the code going back, if you take a look at the code for the pet object there is a function called speak. And if we call that function on a base pointer, on a pet pointer, we will get no output even if the pointer is pointing in a clear cat object. So by default, we've got this issue where the base class pointer might be pointing to a derived class object and we call the wrong version of the function; that could be catastrophic. Imagine if the destructor for the base class was called but what we wanted to destroy was the derived class. That would be absolutely catastrophic; we'd have a memory leak.

The solution to this is to mark the function as virtual in the base class. If we put virtual in the base class then C++ waits until run time to make the decision on what your version of the function to call. So if we have a base class pointer and were accessing a function which is marked as virtual C++ stops and says: I don't know which version to call, I'll wait until I can actually see what object I'm working on and then I'll call the version appropriate for that object. So rather than make the static binding decision, rather than decide which version of the function we're going to call at compile time, we do what's called late or dynamic binding at run time when we actually see the object that's when C++ will make the decision of which version of the function to call. The version of the function call depends on the type of object not the type of pointer. And that's the way that we can get around the problem of accessing derived classes functions via the base class pointer.

Pure Virtual 1.29

There are some situations as you can see where we might not know what the base class should do, but we know that the function should exist. What kind of noise does a pet make? It's a question we really can't answer; until we know what type of pet this is we can ask the pet to speak although all pets can speak. Alright fish make some sort of “blub blub” noise; I don't know. But the point is that the pet class should have a speak member function we don't know how to make a pet speak. All the pets that we're going to create cats and dogs and fish and turtles, and I don't know what else, are going to have a function called speak. And we want to be able to use the base class pointer to access that speak function. So if the base class should contain a function but it doesn't know what the function should actually do, the function can be marked as pure virtual with the equals zero. So here's the code that we're going to look at.

The idea here is that we have the best pet class and we have the speak function. And speak function is going to be virtual, of course, but we don't know what the speak function should do. So we'll make it pure virtual; we'll set it equal zero. And in doing so, it allows us to use the pet class pointer to call the speak function, in fact inside the pet class itself we could even call the speak function. But we don't know how to actually perform the code for the speak function until it's defined later inside the cat class. So if you see the cat will override the speak function and print out meow in this case, and what we've got is the ability to have a pet pointer pointing to a cat object and call the speak function of that cat object.

Now there are some restrictions though: because the speak function is pure virtual it causes the pet class to become what we call an abstract class.

Abstract classes have a lot of restrictions on them whereas we can create pointers to an abstract class; we can never instantiate an abstract class. We can't create an object of type pet, because if we were allowed to create an object of type pet then we could possibly call the speak function and then we don't know how to do that. So C++ says if you have even one pure virtual function you can't create an object of this class. So what can we do? We can create pointers and we can derive from this class. So that pet becomes useful as a base class but we can never create an object of it; we can create pointers to lots of pets and they can point to objects which override the speaker function and provide the code. But if any of the derived classes contain pure virtual functions, because the either have it overridden or they've created new pure virtual functions, then they too are abstract classes. In this case the cat class is perfectly fine; we can create an object of the cat class and have a pet pointer point to it.

The Same Function Call results in Different Output 1.30

Here what I've done is create a dog class. And the dog class is defined as being having a class as the base class and it also creates a speak function. Dogs have ears but that's not important; we don't care about the ear size. The speak function here will woof instead of meow. And now what I can do in main is create an array of three pet pointers. Now it may be a little bit strange to see the pet star star there, but what I'm saying is that the array is in array of three pet pointers. If I created the array as three pet objects; I would not get any polymorphism. Because polymorphism requires that we're using a pointer to access the object and it requires that the function we're calling is virtual. So here I need three pointers to pets not three pets: I set the first pointer equal to a cat, I set the second pointer equal to a dog, and I set the third pointer equal to a cat. And then I can use a for loop to go through and make all those pets speak. And what I think is great is that if you look at that last line of code, we're making a function call to the speak function via a pet pointer. And we have C++ deciding at run time, which version of the function is going to be called. So it doesn't know which version of the function to call until we actually get to the point in the code, the very nanosecond that C++ is running, that the computer is running that code and it says: oh I'm looking at a cat object so I will call them in the cat's function for speak and I will meow, or I'm looking at a dog object and I'm going to print out woof. So I think that really kind of almost magical, the way that C++ can take the virtual function and a pointer to the base class and decide which object it's actually looking at and call the appropriate function.

What We Discussed 1.31

We went over a lot in this module; and I know it's like drinking from a fire hose right now. But we did talk about the definition of the object in class; we talked about the whole concept of encapsulation. We talked about creation of a class and the enforcement of protections on the class. We talked about accessors and mutators, and the const modifier. We talked about constructors and how to guarantee that they were constructed properly. We talked about using operators on classes and we talked about classes that contain dynamic memory, and then we went into inheritance and polymorphism.

There's a lot having to do with object orientation and the amount that you have to capture for this module is really significant, but once you kind of get your head around the whole idea that we're working with data and the functions to work on that data as a single entity, it really starts to become a little bit obvious. So I hope you are able to follow along with it and if not e-mail us with any questions. Thanks.