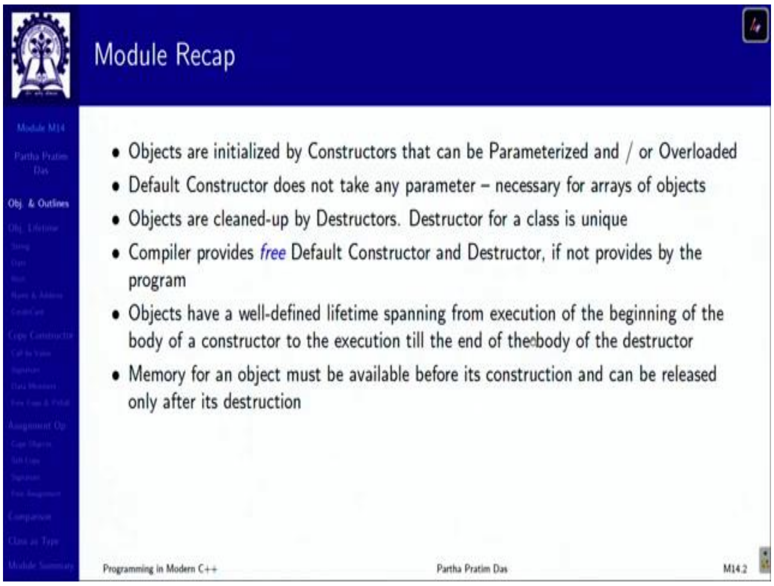
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture – 14
Copy Constructor and Copy Assignment Constructor

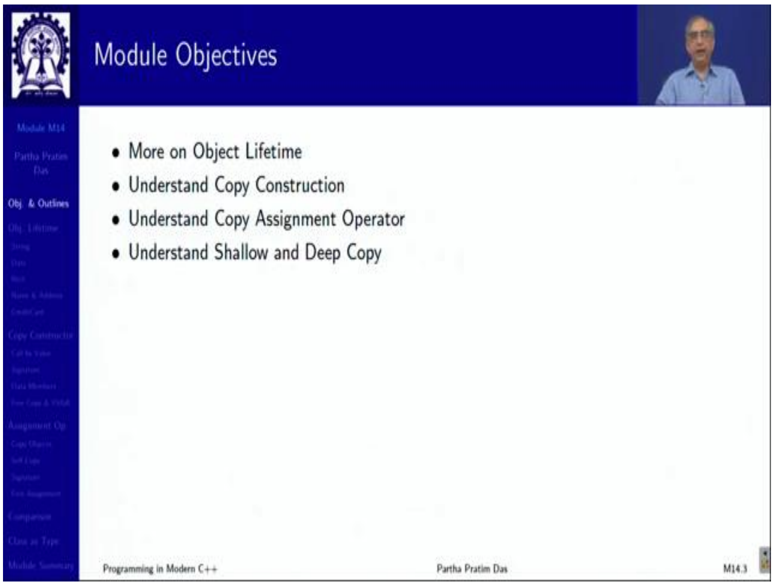
Welcome to programming in modern C++, we are in week three. And we will be discussing module 14.

(Refer Slide Time: 00:32)



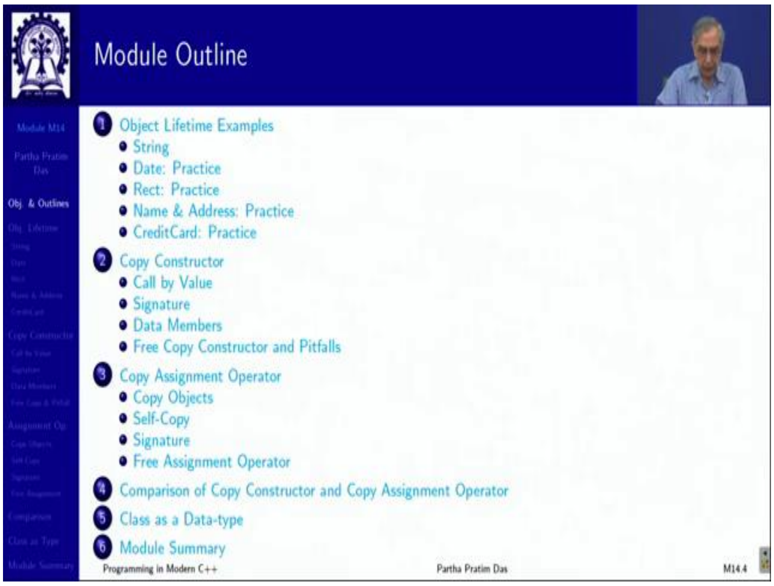
In the last module, we have taken a look into the constructor and destructor that the basic process of creating an object and releasing it with taking care of all dynamic data and so on, to have a very well defined lifecycle. And we have shown examples of the lifetime of variables that are objects, that are automatic, that are static and that are dynamic.

(Refer Slide Time: 01:04)



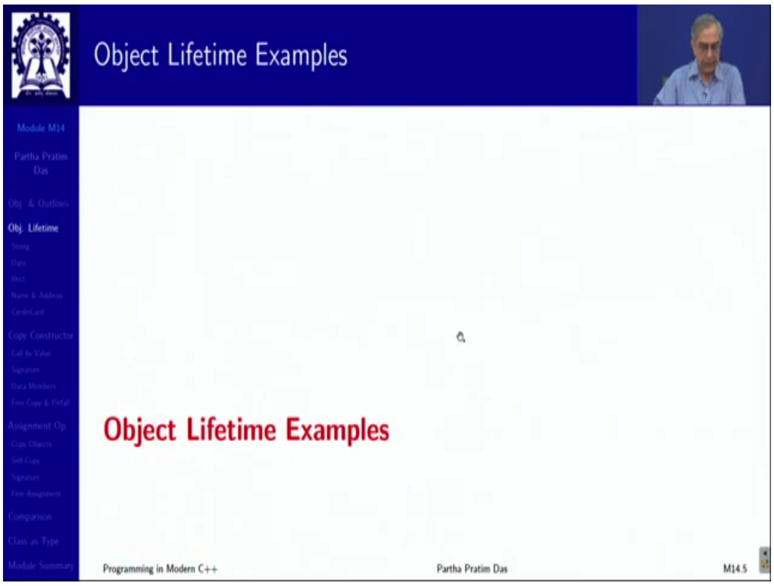
In the, in this module, we will take a couple of more examples of object lifetime, which you should practice. And we will primarily deal with the issue of copy, how do you make copies of objects? And what does that mean, in C++ and why is it important.

(Refer Slide Time: 01:24)



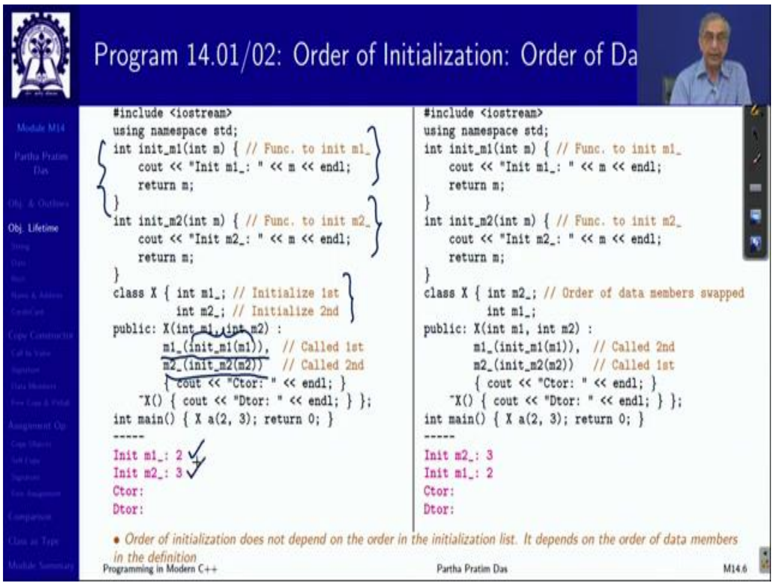
As you will see, in several slides in this presentation, the title is prefixed with practice in blue. And those slides are not what I will discuss in the presentation, but those are included so that you can study them in depth and possibly try them out to have a better understanding of the respective issues.

(Refer Slide Time: 01:49)



Here is the module outline as every time.

(Refer Slide Time: 01:51)



So, in terms of object lifetime, we want to answer a very specific question in this slide as to what is the order in which the different data members of a class is constructed, when the object of that class is constructed? So, you have an object and you have different data members. So, to construct the object, you need to construct each and every data member. Those may also be a user defined types, so they will have data members.

So, the process goes recursively in that manner. But with a simple example, we try to illustrate what is the order in which they are constructed, is it by the ordering which you write them in the initializer list, or is it by the order in which you list them in the class definition. So, to start with, we have a sample class containing two integer data members.

And to be able to understand as to which 1 is actually getting constructed, we have created two insertable dummy functions. So, instead of directly copying the value, we will pass the value that we want to initialize into this into data member to the init function, corresponding init function. So, that I can print the message and know in which order they are happening.

So, these are the two data members and I have that in the initialization list, I put the initialization of m1 first. And I initialize it with the value returned by this function that is, that is, this is just an instrument to tell us that we get to see in in which order they are happening, because the print messages will happen in that way. So, I do that and when I see I see that init m1 has happened first and init m2 is happening second.

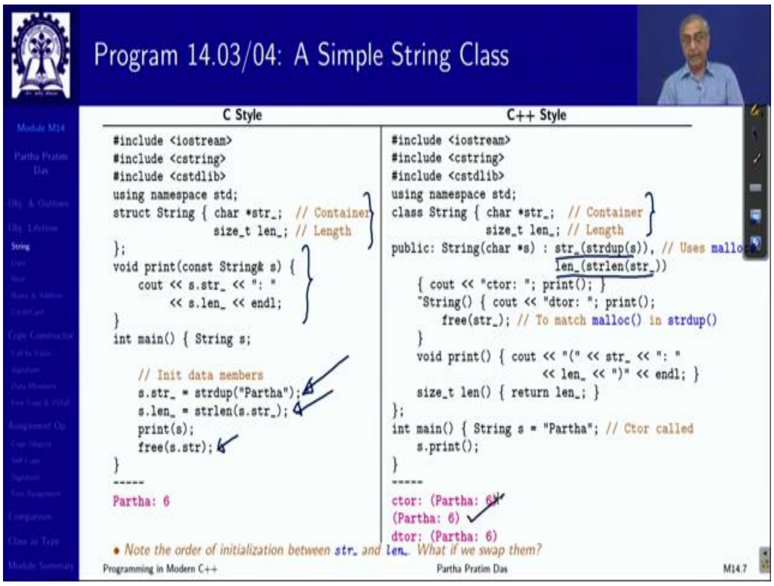
Now, the question is, is it because is init m1 initialization of m1 construction of m1 is happening first is it because m1 is earlier than m2 in the list of data members in the class? Or is it because in the initialization list, it has been initialized first? So, to test that, let us change the let us change the order of data member list in the class.

I now list m2 first and m1 next and I do not change the order they are ordered in the initialization list. Now as I do this, I find that in the invocations are in the same way using the functions. Now I find that init m2 is happening first. So, this clearly tells us the init of m2 is actually listed later in the initializer, but it is happening first because it is the first data member it occurs before m1.

So, this is this simple program tells us that the data members will always be constructed in the order in which they are listed in the class and not in the order in which they are given in the initializer list, which could be anything. And this is very important to understand because a particular class may have multiple different constructors as we have seen.

So, they may list the data members, they may need to put the data members in any order whatsoever they want to. So, if that were detected, that were directing the initialization order, then there would be ambiguity. So, when it is in terms of the order in which they are listed in the class, obviously, there is no ambiguity because there is only one list, irrespective of how many constructors you may have for the class. So, let us see the consequence of this.

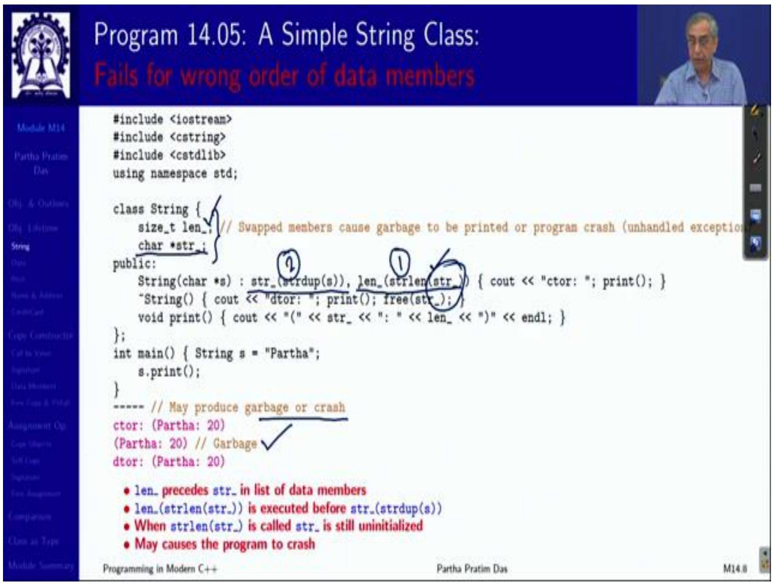
(Refer Slide Time: 05:53)



And for that we pick up a simple string class having a char* pointer for the containing the string and another variable len to keep the length of the string. And this is the order in which they are given. And there is a print here. So, I first in, in the C style I first do a copy of the given string, put the length of that string.

And finally, I free up the space created by strdup. In the C++, actually, I put them in this same order. And I initialize str first and then len, in terms of the initializer list. And what I get to see is, well, it is it is working fine. I can see the call to the constructor and the call to the destructor. But that is not dependent on the order.

(Refer Slide Time: 06:52)



Now, let us question that what happens if I change the order of these data members? If I swap the order of the data member, I first put len, and then put the char* str. If I just change these two, and as you will know that the order of construction will be in the order the data members are listed. And you will see that this will cause a disaster. Because even though here, this is given first, that is you should copy the str, copy the string s into str.

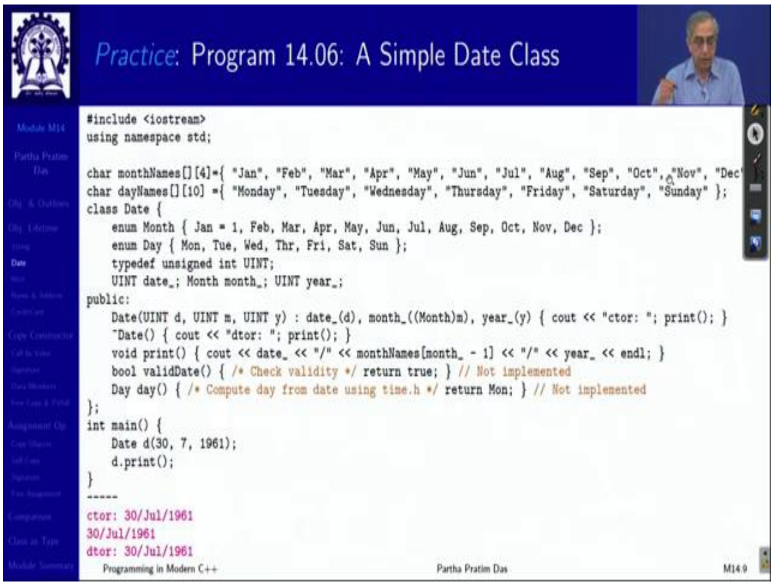
And then you are saying that from the str you will construct, you will compute the length to, this is the order given, but the order in which they will execute is first the len. So, this will be first to execute and this will be second to execute. Now obviously, when you are executing this first, you are passing on str, the second data member, which is still not been constructed is still not been initialized.

So, it is, it is, it is a garbage pointer, basically. So, you are passing a garbage pointer to strlen, so you do not know what it is going to do. So, if you if you run this program, then there are several possibilities. One is what I found on the minGW compiler that I am using on my Windows machine is that it does not crash it constructs, destructs but it arbitrarily takes some values 20.

Well, the actual length of the string, partha is 6. And when I run it on my visual studio, windows, visuals Microsoft Visual Studio, then the program simply crashes saying that there is an unhandled exception, because you have passed the garbage to strlen. And so you will have to be very careful in terms of the order in which you if there is dependency between the values of one initialized data member to the other.

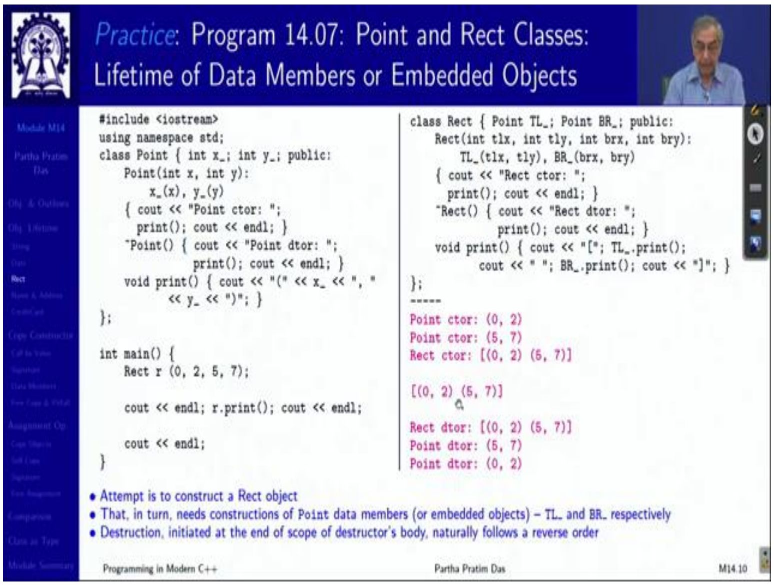
You have to put them in the right order in the listing of the data members. So, that is what so it says that the lifetime of I mean the reason I am discussing it here is it says that the lifetime of str has to start before the lifetime of len is expected to start, because len depends on the str.

(Refer Slide Time: 09:06)



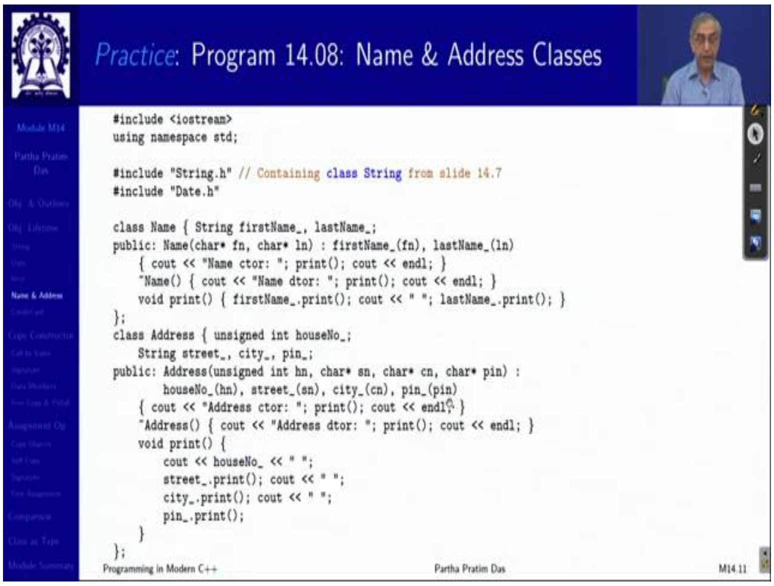
So, now, I have put a number of examples for you to actually trace the constructor destruction and the use of object lifetime. And as you can see here, the title is prefixed with practice. So, we will not get into the details. We will possibly come back using them in, in different contexts later on. But I I insist that you try out these programs, read every line of it and understand it well.

(Refer Slide Time: 09:36)



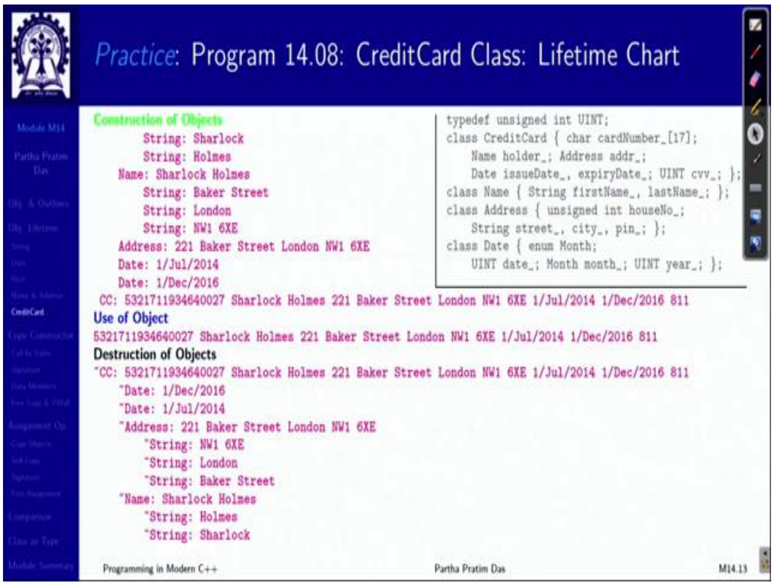
So, there is a, this is the design of a date class. This is our good old friend, point and rec class and I have shown the expected output everywhere so you can check and get convinced that things are happening in the right way.

(Refer Slide Time: 09:48)



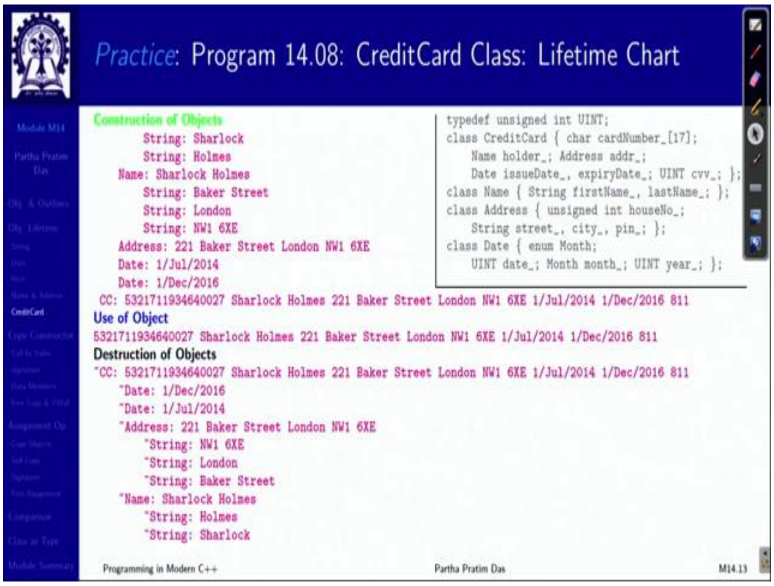
Then there is a name, class and address class.

(Refer Slide Time: 09:52)



These are all for your practice including the credit card class which uses all of these.

(Refer Slide Time: 09:58)

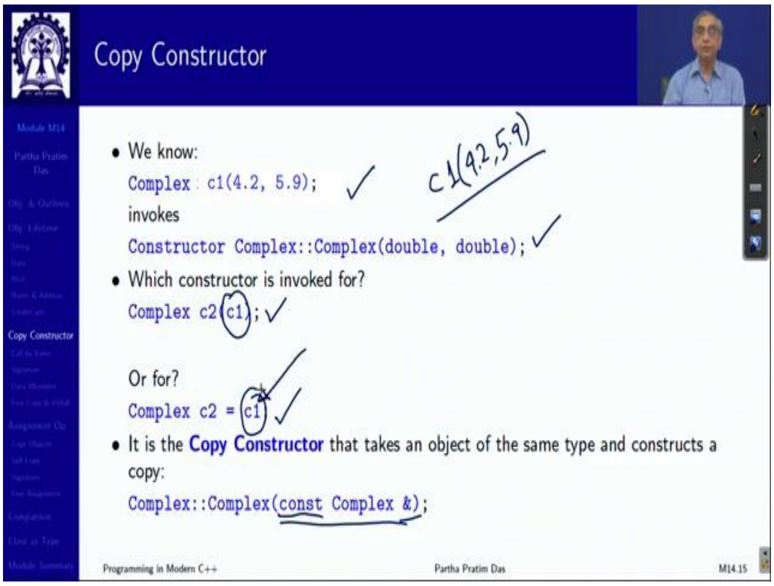


And if you if you do that, if you run that, then this is the kind of the, the magenta is the kind of output that you expect from this, while you have on the right top the design of different classes. So please practice them out to get more confidence.

(Refer Slide Time: 10:15)

Now, let me move on to the copy issues.

(Refer Slide Time: 10:19)

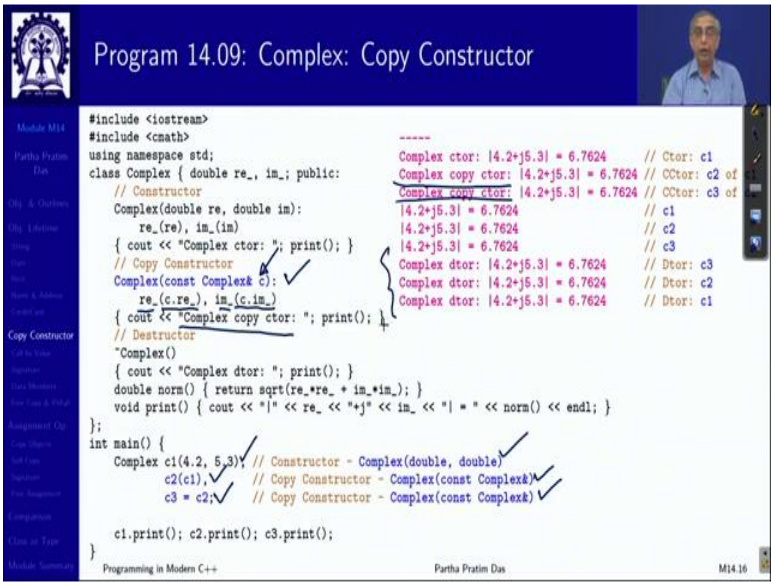


The first is a copy constructor. So, what we do is, this is a normal construction, we all have seen this. So, when we do this, we can we can do it by this or we can actually write it as 4.2 5.9 like this also. So, it constructs it expects a constructor which double double and construct by that. Now, what if you construct, invoke it like this. That c1 is already constructed as a complex object, and you are trying to construct c2, putting c1 in the parameter of this.

Or putting c1 in the initialization value. This, this is the point where copy construction is taking place. Why? Because you already have an object of the complex type here or here. And you want to create another object of the complex type as a copy of this object. So, the copy constructor takes place. So, in the copy constructor, naturally, what you are passing as a parameter is an object of the same type.

So, naturally, it is it has to take a parameter which is complex. We will avoid using the I mean, we would avoid using the call by value as you know, so we will put it as call by reference. And we do not want the source object to be changed by this copy constructor. So, we will put a const. So, that is the that is a basic copy constructor design, that we will have.

(Refer Slide Time: 11:50)

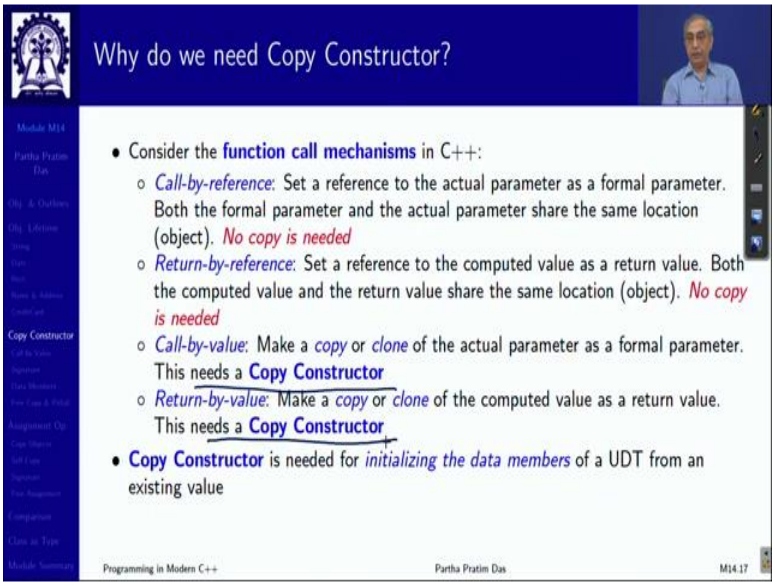


So, here is an example of this, where this is the copy constructor. So, what we do is, I am passing a object c of complex type to this constructor. And I take the real component of c copy to the real component of the object to be constructed similarly for. So, basically, item by item, I am data member by data member I copy. And if those data members are user defined types, then their respective copy constructor in turn will again get invoked.

It is, it is the same process as as the construction where construction in general can take parameters of any type, or may not have a parameter for default. In copy constructor specifically, you take an object of the same type. So, when you write this, you are invoking a constructor, and when you are writing like this or this you are invoking the copy constructor. And I have put messages here specifically.

So, that in the output trace, you can find out which constructor is being called and in which order destructors of course, will happen in the reverse order as we have seen. So, this is a, this is the basic story of copy construction. So, it is like copying variables we have.

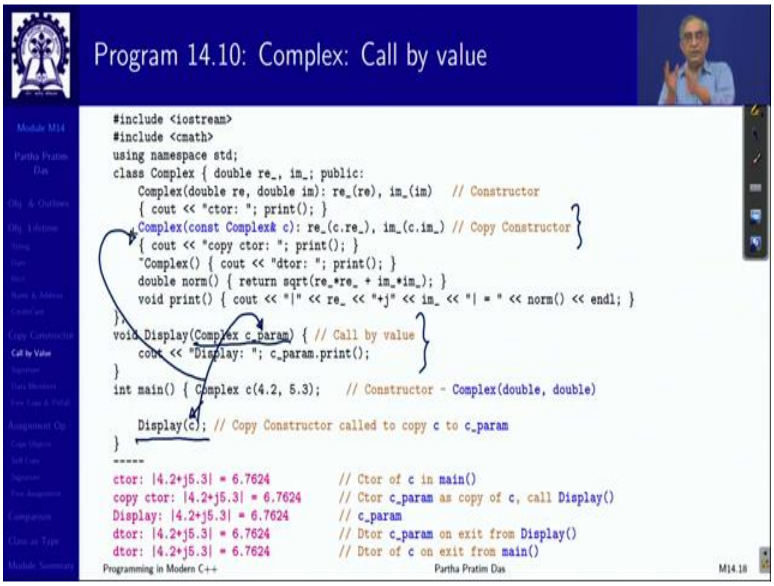
(Refer Slide Time: 13:11)

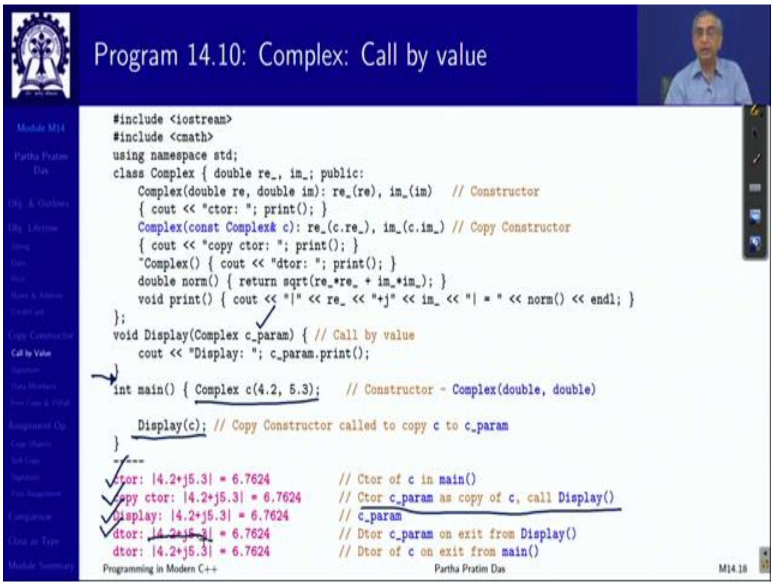


Now, we have given ourselves a mechanism to copy any object that we have created for our user defined types. Now, the question is, when do you need to do this copy construction? The primary reason you need copy construction is to support call by value. As you know, if you do call by reference or return by reference, then only the reference is being passed, no object copies as we have seen already.

But if you do a call by value, then naturally you need to copy the given actual parameter object to the expected formal parameter object. So, you need a copy and that copy is what is done by the copy constructor. Similarly, if you are returning something by value, then you will need the copy constructor. So, copy constructor is needed for initializing the data members of a UDT from an existing value.

(Refer Slide Time: 14:06)





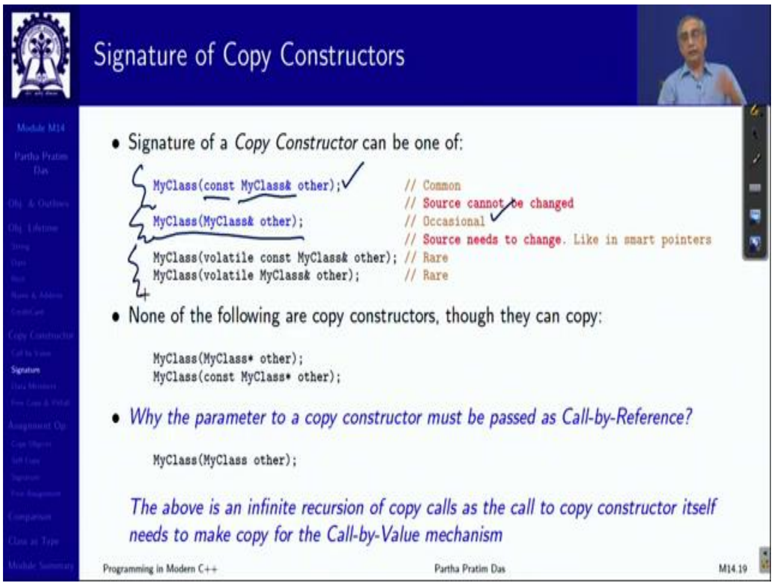
So, this is the reason without the copy constructor, you will not be able to call functions by value. So, here I give an example, this is a copy constructor. And I am trying to write a global function, which is displayed which takes a complex parameter by value. I am not putting the &. So, as I take it by value, and I call it so my actual parameters c needs to be copied to the formal parameters c_param.

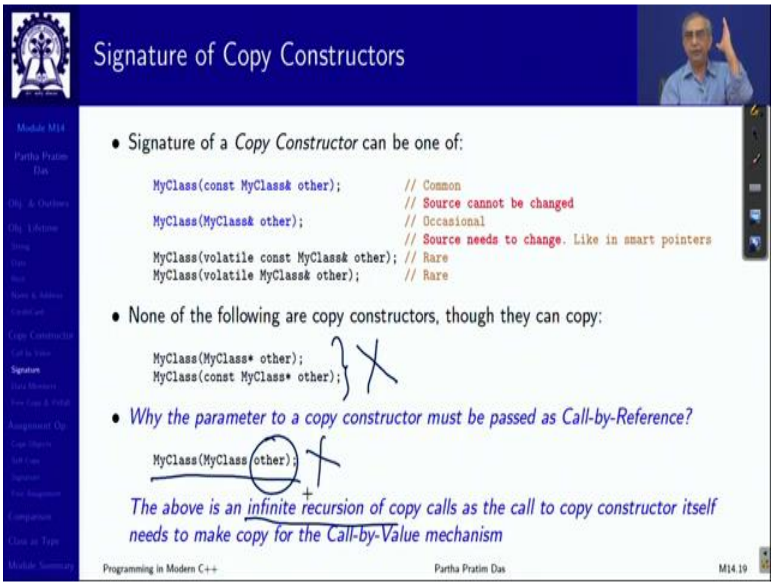
And when I do that, this copy process will invoke the copy constructor because it has to copy the fields of c into the fields of c_param. So, as you can see here is really that initially when you do, initially when you are doing this, you have a constructor call. And then when you you have not made any explicit copy constructor call here, but when you call display try to call display, you have a copy constructor called copying c_param as a copy of c in display.

And then the display is actually called it does the display process. And when you are done, when you are done, you are here you have constructed a an object here. So, the lifetime of that object has started, that formal parameter object has started with the call of display. So, when you get to the end of display, your control is going back, you are out of that scope.

And by that same rule, the destructor will be called for c_param. So, the destructor is called and with the same set of values. And then finally the destructor of c is called at the end of main. So, this is how the call by value is supported.

(Refer Slide Time: 15:56)





You can see the same thing if you do return by value as well. So, what is what should be the signature for the constructor function. So, this is the most common signature that you pass the parameter to the copy constructor by reference by reference, and you make that a constant so that the source cannot be changed. It is also possible that you write a copy constructor where you have not used constant.

In it, you might wonder that while I am copying, why do I need to change the source? See there is a subtle difference between what we say is a copy and what you say is a move. If I want to realize move, which does not exist in C++ 03, then what I want is not only make a copy but actually invalidate the source if required. So, that there is only 1 because if you are just to copy there are two copies of the variable.

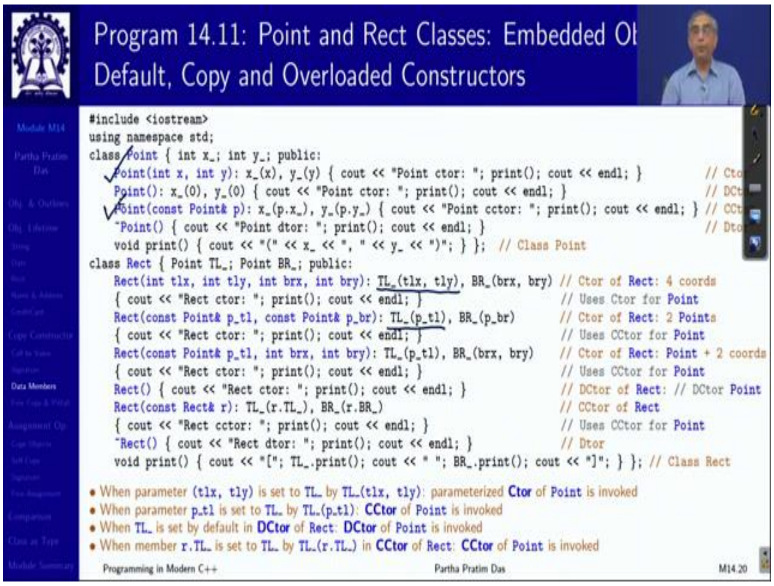
If I just want one, that the source has moved to the destination, then I will need to use this I will talk about this in more details when I talk about smart pointers and how they use this. And we will come back to that in C++ 11 when we talk about the move semantics, which is specifically supported. So, this is the most common 99.9 percent of copy constructors are like this, some are like this and very rarely you may use other qualifiers like volatile and all that.

So, it it may also be noted that if you pass a, if you pass a pointer, like if you try to do a call by address for the copy constructor, you can, you will still be able to make a copy. But that is not a copy constructor, that is if you provide this, the compiler will not use it in the context of call by value or call by reference. So, do not consider that these are functions. These are obviously some overloaded constructors, but they are not copy constructor.

Now the final question is why do I need to pass the parameter to the copy constructor as reference? What if I pass it as a value? Now what will happen? If I pass it as a value, then to be able to call the copy constructor needs to copy this value, which in turn needs the copy constructor itself. So, I am passing it called by value, so call by value needs, the value must be copied that needs the copy constructor must be invoked.

And if I do that for the copy constructor, then the copy constructor itself will have to be invoked. So, it will have to be called. And to call that I need to copy the actual parameter again. So, the copy constructor will be called again. So, I end up having an infinite recursion. So, this is not if you write this, then you will basically have an infinite recursion you will never be able to end. So, the only way to write copy constructed is by reference and preferably by constant reference.

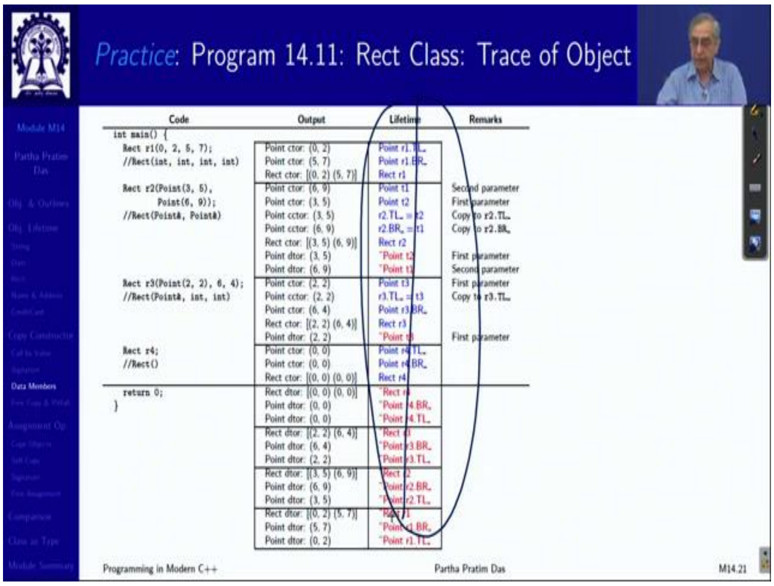
(Refer Slide Time: 19:04)



So, these are again, our point and rect class the different instances. For example, if you see that in the initializer list, you are constructing tl the top left object point by x and y the integer value, so this will invoke the normal constructor. But if you are doing this, that is where your ptl is already a point object and you are trying to create the tl member of the rec class by copying it then you will have the copy constructor invoked.

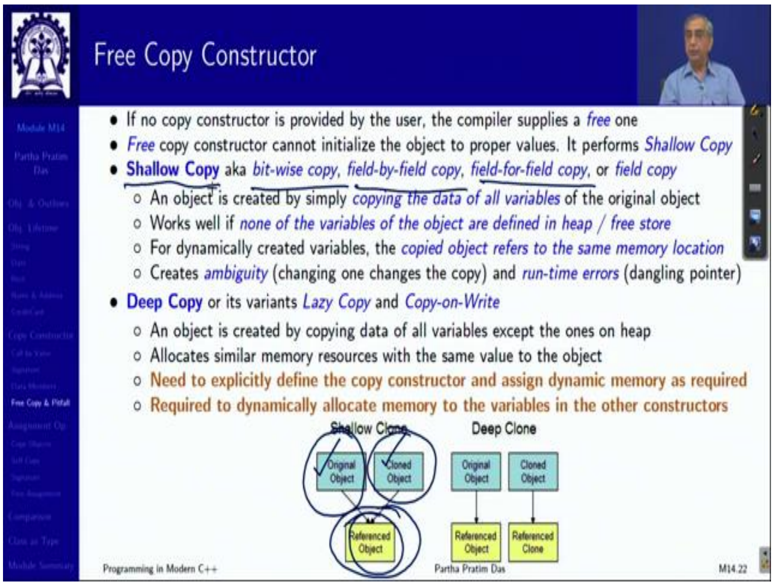
So, you can study it well. The notes are all given explaining what every stage is doing and understand. Where does the default constructor come in? Where does the copy constructor come in? And what does the other overloaded constructors come in?

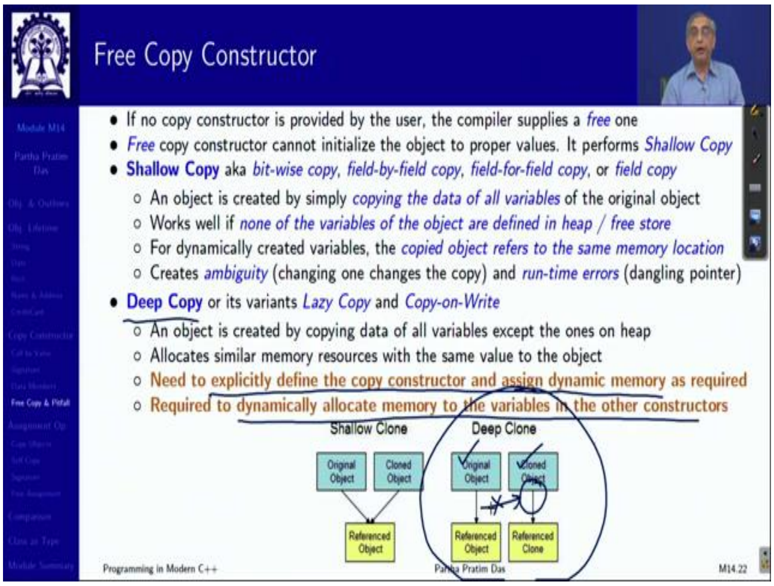
(Refer Slide Time: 19:57)



So, if you run that program and you will see that actually, this is the kind of output that you get. And it will be a good exercise that you do not look into this, right hand side column first, keep it blank on your copy, guess take output, and then try to put the lifetime information as to when things are getting created, when things are getting copied, when things are getting destroyed, when things are getting used and so on, that will give you a good practice. So, this is a, this is yet another practice slide that I have put for you.

(Refer Slide Time: 20:29)





Now, like the constructor, if you do not provide a copy constructor, then the compiler will give you a copy constructor. Now, the copy constructor, the compiler provides, because if I mean, if you have not given a copy constructor, and you are using that object of that class, by in call by value or call by reference, I mean call by value or return by value, then certainly the compiler needs a copy constructor.

So, it will provide you a free one. Now, it does not know what to do, what so what it does, it just copies a bit pattern, whatever the object had it copies a bit pattern. So, this is called the bitwise copy field wise copy, field for field copy, field copy, field copy and so on. So, if you have data members, which are just, built in types, whose actually they do not have any specific thing to construct, their values will come properly.

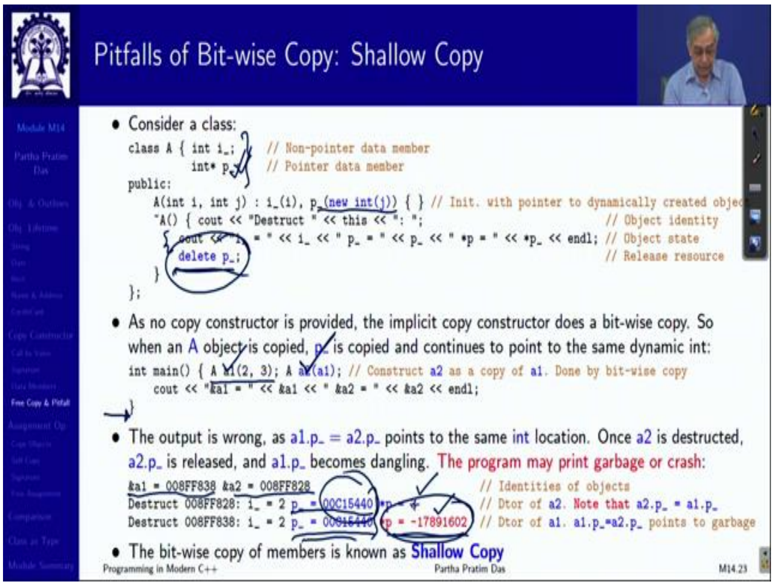
But if you have some reference variable, some pointer variable, which has allocated a location, then what will happen this is your original object. Now, you have if you have copied that pointer value, then in your copied object, the pointer value will be the same. So, you have two copied objects having the pointed values, which are same, so they are basically referring to the same object.

So, this is a very ambiguous copy. And this might lead to unexpected results because, for example, after cloning, if I clone is another process used for referring to the copied object. So, after cloning, if I use that pointer to change this reference object, then the same value will be reflected in the original object. So, this type of copy is called shallow copy, where you are just copying the pointers, because you are doing a bit copy.

That is what the compiler has given you. The other which is which is basically the preferred or the correct way of doing this is to have a deep copy. So, what you do in a deep copy in a deep copy, you for the, every built in type, you copy the value. For every other data members, you call the copy constructor of the corresponding class and for pointers, you actually make a fresh allocation and copy the pointed value as well.

It is not enough to just copy the pointer that is not semantically correct. So, you have to explicitly define the copy constructor assign dynamic memory as required. And put the so we have in terms of a deep clone or deep copy your original object is giving you the clone object. You have a reference object, you are not copying this pointer, you are not copying this pointer rather, you are doing a fresh allocation and copying this object by that object's copy constructor. So, that is a, that is a big pitfall of having a free copy constructor from the compiler or the pitfalls of shallow copy.

(Refer Slide Time: 23:43)



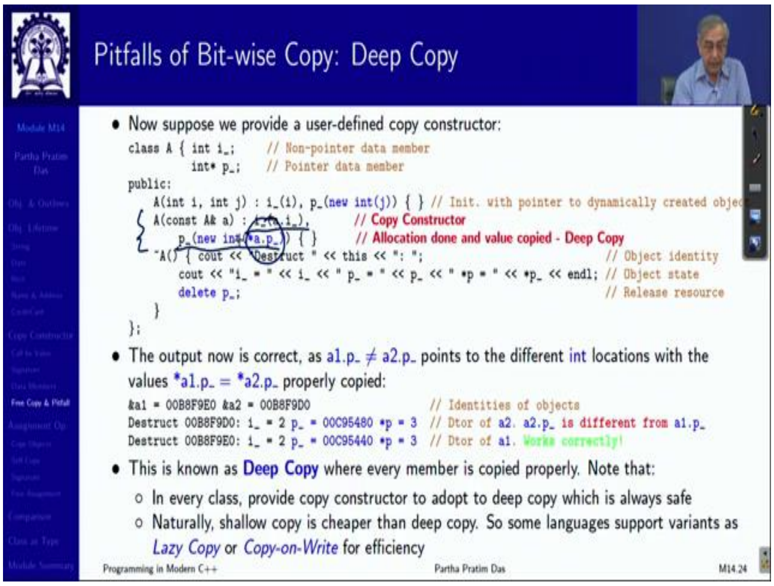
So, to just to illustrate, I have shown here a class with one integer value and one pointer two integer. So, naturally in the constructor, a value is in a dynamic allocation is made and that will be deleted later on. Now, I have not provided any copy constructor. So, what the compiler will do? It will provide a free one which will copy everything. So, what happens I have two objects, one is originally created, the other is a copy of that which has been created by bit copy, shallow copy.

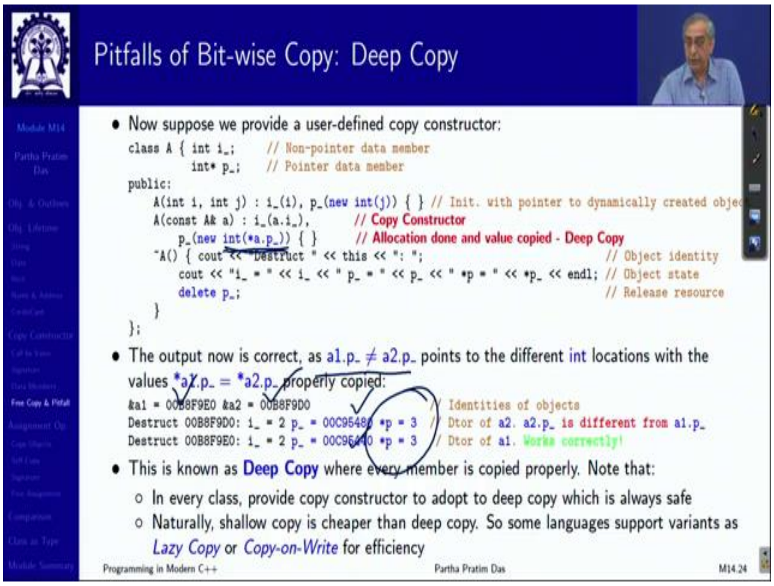
So, when I print the addresses of these two objects, I find that different objects. But when I go deep and print the pointers, this particular p_ pointer that has also got copied. So, they have same values and both of them will try to point to the same location. Now what is happening, what when I do this, then at this point, you can see that my destructor is actually making the print.

So, when I come to the end of main, this object will be the first to get destroyed. The last created first destroyed. So, this object, when this object is destroyed, this p_ pointer is deleted. So, now it becomes a dangling pointer. So, when you go to destroying a1, which is a first object created the second to be destroyed, then you are again trying to do a delete p_ or you are trying to print the value of pointed to by this.

But p_ has already been deleted, because it was pointed to by a2 and a2 has done the destruction. And it is deleted that pointer is a dangling pointer, so, you get a garbage value you might get crash also. So, this is a problem with the shallow copy.

(Refer Slide Time: 25:41)



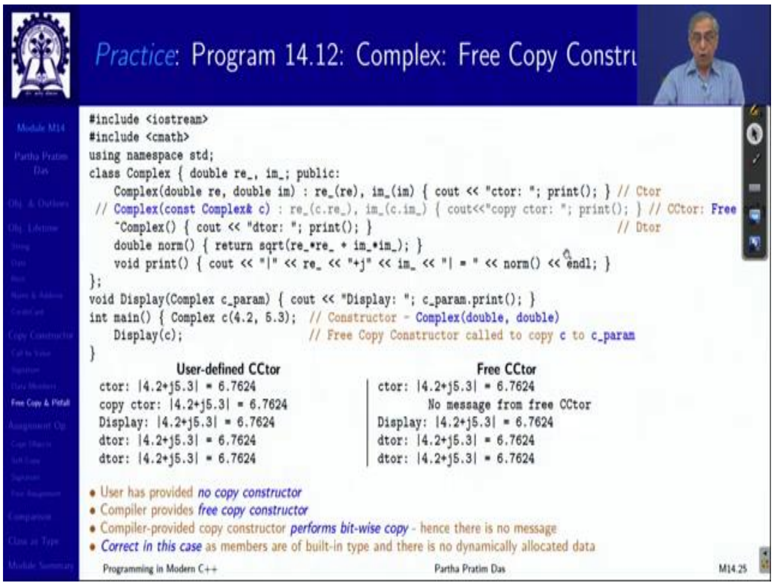


So, do not do that always do deep copy. So, as in the copy constructor, now, you can see that you copy the integer variable because it is not a pointer. For a pointer, you do an allocation and initialize it with the value it was pointing to. Once you do that, and you write the same same code, now, what you have is in the copy it is not a bitwise copy, it is no more that you have these two pointers same.

So, in the two different objects that you have got the two pointers are of different value. And therefore, when a2 is destroyed, and that pointer is released, the pointer in the a1 does not get affected, and both of them have the same value because that is how you have created. So, this is what is deep copy. There are alternate terms also which is used, some use lazy copy, some use copy on right.

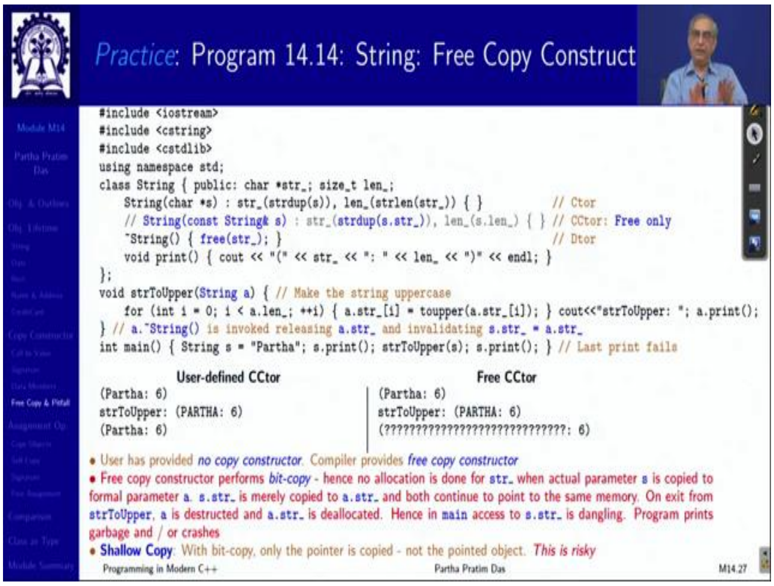
There, they are not exactly same, there are semantic differences and at an appropriate point I will explain that. But the whole idea of deep copy is copy the pointed variable values separately with allocation.

(Refer Slide Time: 26:54)



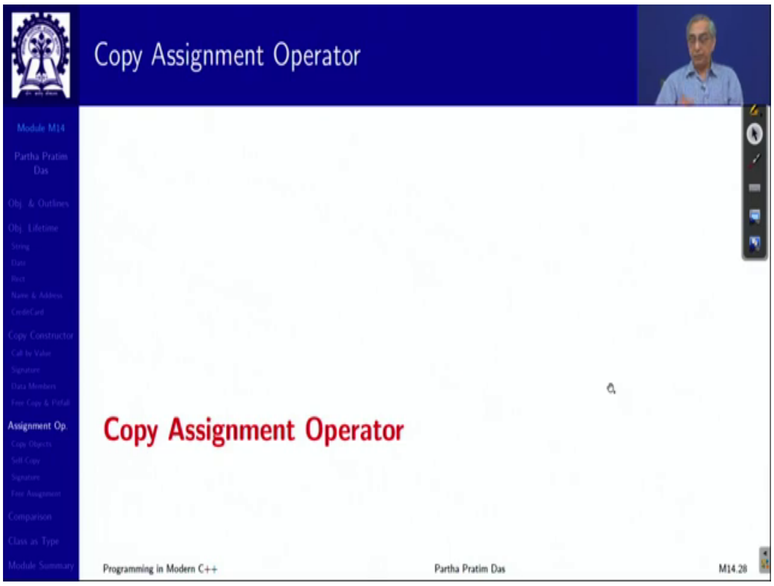
So, again, practice examples given here with a free copy constructor to carry out through them. There is a user defined copy constructor for a string object class, please try that with the free copy constructor.

(Refer Slide Time: 27:02)



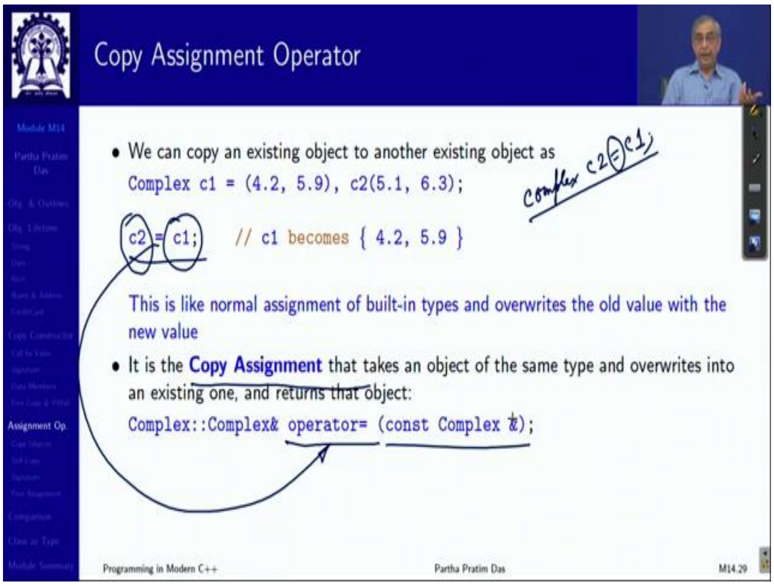
And in the, in the notes, I would always suggest that do not read the notes, first understand the program. If you cannot figure it out, or after you have figured out what has happened and try it build and try to run it. Once you have seen what is happening, then you go to the notes and confirm that your understanding is correct.

(Refer Slide Time: 27:32)



So, this is how the practice ones must be done. Now, let me talk about the copy assignment.

(Refer Slide Time: 27:39)

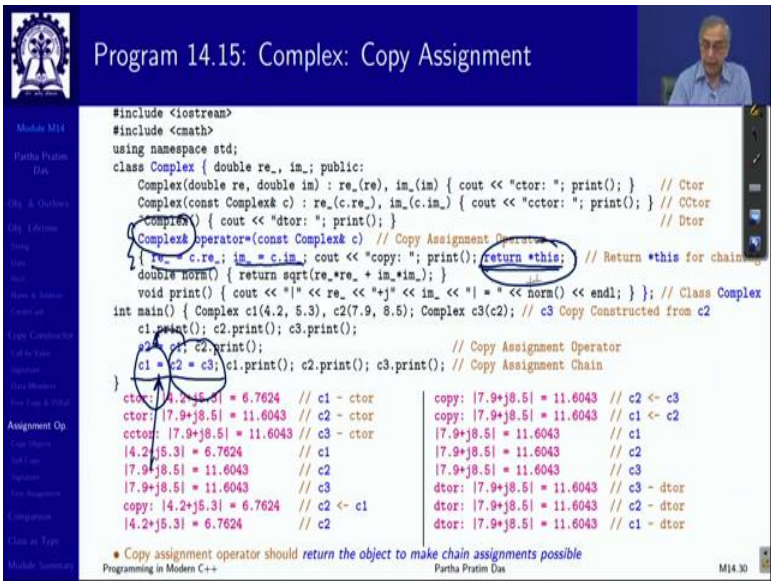


So, other than the construction, which was we were doing, the other that we do regularly is copy 1 value into another. So, when I write it, when I write something like complex c2 then I made a copy construction, because this is an initialization.

But if I simply write a statement c1 c2 both are available, and if we simply write a statement c1 is copied to c2, then I need a copy to be done which means that I would like to erase the values in c2 and overwrite them with the values in c1. So, this is an assignment. So, this is called copy assignment or assignment.

Now, we know operators can be overloaded. So, this operator of assignment is Operator Assignment. Naturally it takes the object of this complex type, the source type, which will come as a, as a constant reference. And what does it return? It has to return an object of the same time because you have copied. So, it also returns by reference the same type of object.

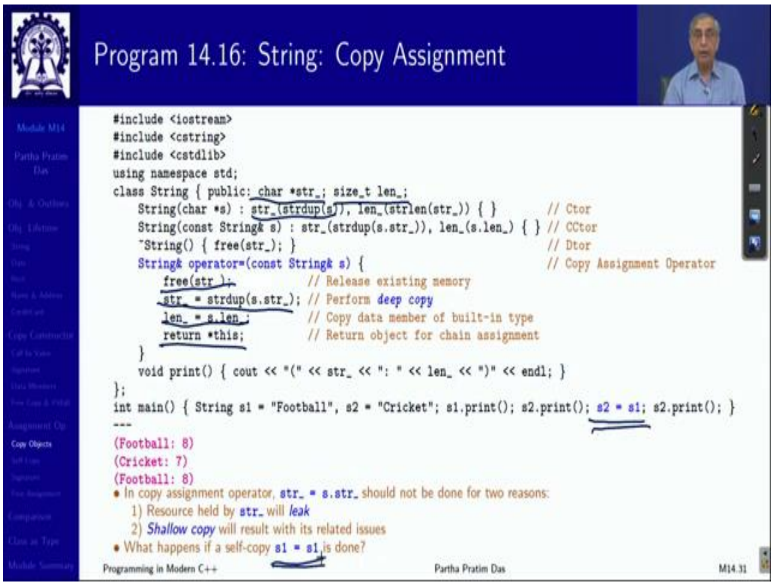
(Refer Slide Time: 29:03)



So, here is a copy assignment here, the copy assignment for this. And it returns, returns this, it is not only enough to copy and make changes. Like you are copying and making changes here, which is fine. But it is also important that it returns the object, of the same type. Why do we need that because we may want to change the assignment. For example, here we have changed the assignment.

So, c3 so this operator is called first because this is right associative. So, c3 is copied to c2 and whatever is copied that object is returned. And that then is copied to c1. If you did not return that object that you have copied into. Then the second one you will not be able to write. So, to support this continuity of semantics of assignment being chainable, you need to return the same object.

(Refer Slide Time: 30:13)



So, this is what is the copy assignment, you can see the copy assignment in string. So, if in the, in the string, you have a pointer and the length, so the string is pointing to an allocated memory where you have doped. So, what you will have to do is when you are copying, you cannot simply copy that pointer from your source to the destination object in the copy assignment.

Because if you do that, then whatever the your destination object was pointing to will leak, because that memory gets lost. So, you have to free that up, then do a fresh strdup, to do a deep copy, as you have understood by now, copy the length and return the same object. So, that is a that is a simple way of doing this. This works pretty fine and if you do this with s1 being assigned to s2, you will have a very nice working.

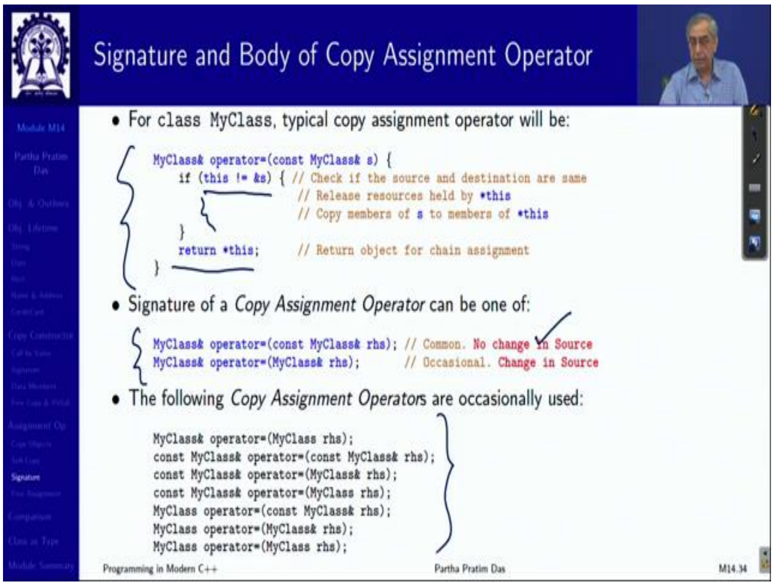
Now, what if you are doing a self-copy? Instead of this, what if you are now you might ask why should I do a self-copy, but it is always possible because, you do not in the whole set of programming, you do not want to create an operator. Because in a built in type of int x, if you assign x to x, then it has it it works perfectly, but here it should also work perfectly, but does it?

Suppose you have done this, this is the only change that I have made. Now, what will happen? You have released the memory once you have released the memory here, here you actually wanted to release the memory here in the destination object but is same as the source object. So, it has got released in that as well. So, when you are doing this strdup for performing grip copy, it is a garbage value.

So it will create a either a garbage value or a program crash. So, self-copy is a problem in this strategy. And so it has to be detected and somehow taken care of. So it is very easy to do that. Because if you are doing a self-copy, then all that you need to detect is the object that you are getting as a source and the current object on which the copy assignment operator has been invoked.

That is that this pointer, these are the same object. So, what is identity the address, so this is the destination object. And &s is a source object. If they are identical, then they are the same object and therefore, there is nothing to be done for this copy. So, if this is equal to &s, you just return this, that is all. Otherwise, you do this, where you know you are, they are different and it is it is going to be always correct.

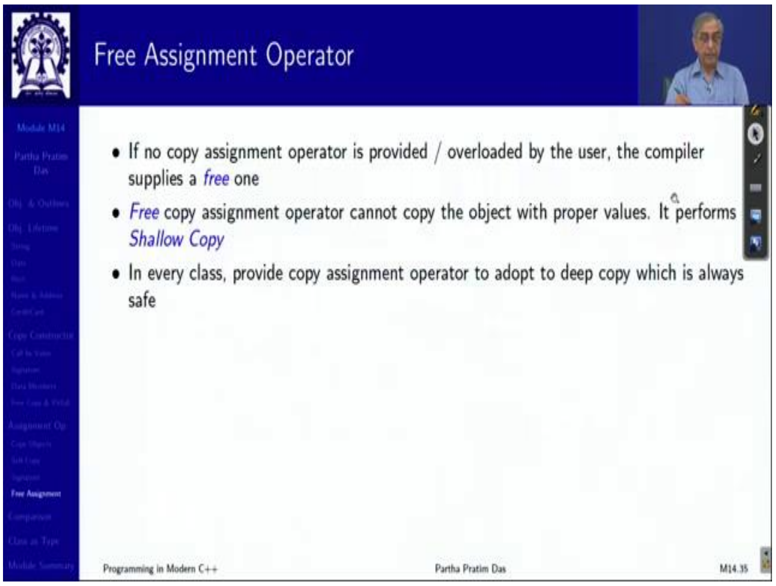
(Refer Slide Time: 33:15)



And this is the self-copy needs to be taken care of. And otherwise, so, this is what turns out to be what is the basic signature of the copy assignment operator, where you check for self-copy, do whatever is required for the deep copy, and then you return this.

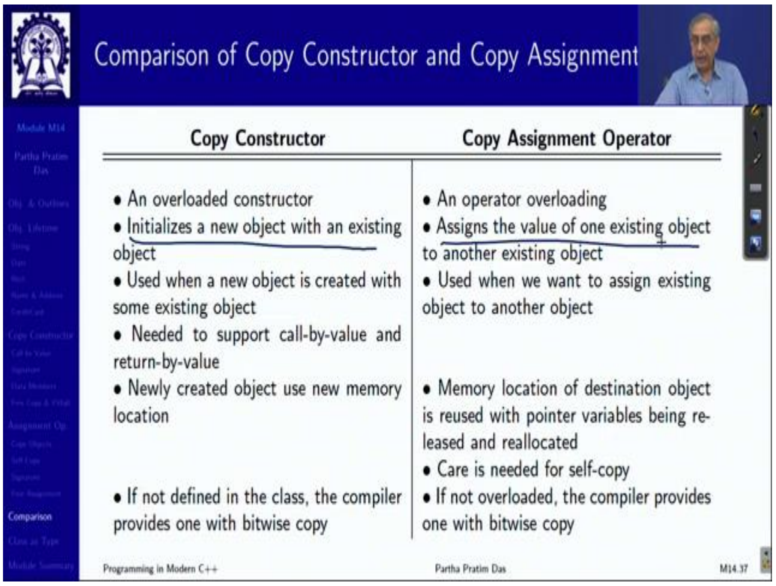
You can copy assignment operator could also be of this type, like the copy constructor that is in the first 1 which is common, if the source is not changed. In the second one, the source will get changed. So, I might want to move this, there could be several other signatures, but they are rarely used. So, do not try to use them.

(Refer Slide Time: 33:54)



Copy assignment operator could also be, will also be provided free by the compiler and therefore it will come with all the issues of shallow copy that we have talked off.

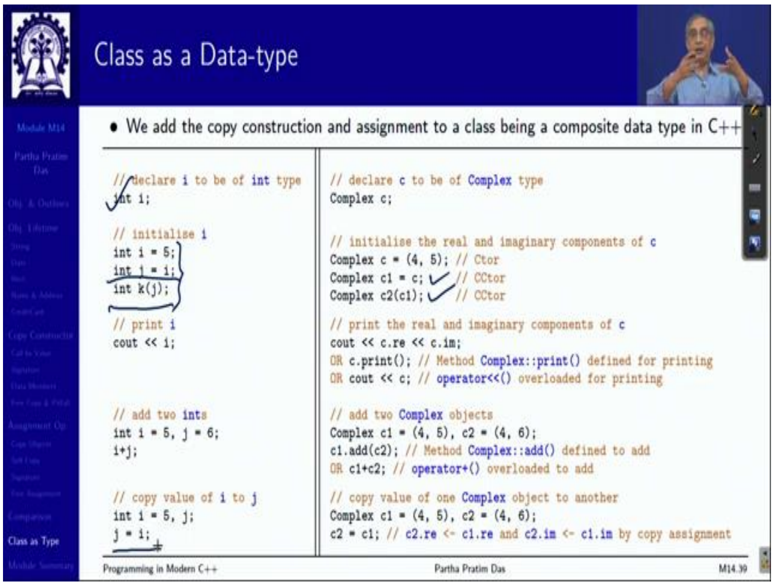
(Refer Slide Time: 34:06)



Now, before I conclude the just to compare what is the difference both our copying constructor and assignment operators. Now constructer is overloaded, the operator is also overloaded. The basic differences is when you do copy construction, the object does not exist. So, that object has to be created as a copy.

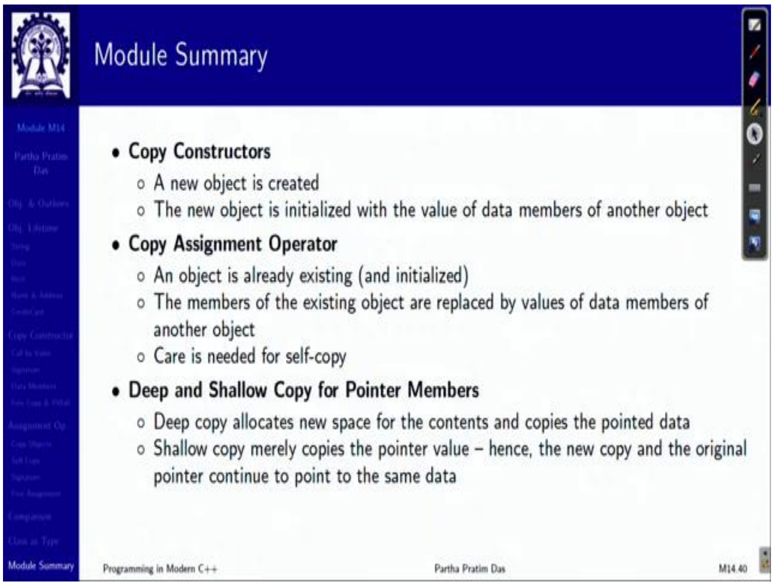
Whereas when you are doing copy assignment, there you already have two objects and you are changing the destination object according to the source object. So, that is a fundamental difference, rest of it is whatever we have discussed in the slides so far.

(Refer Slide Time: 34:50)



And finally, you have the class as a, as a type. So, this we had seen earlier. Now with this and this coming in, we have the copy construction which is also common, also available for the built in type. So, you can see that your assignments and like in assignments here you can see that we are actually extending the class and making it more like a perfect type.

(Refer Slide Time: 35:22)



So, with so with this I conclude on the on the module with different semantics of copy construction assignment, and deep and shallow copy. Do practice this very, very thoroughly because this will be critically important in all kinds of things, programs that we write in future. Thank you very much for your attention and we will meet in the next module.