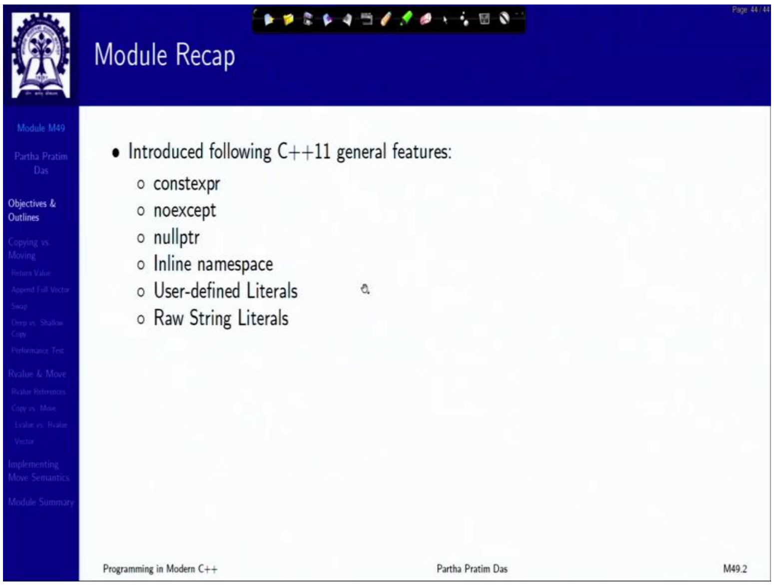
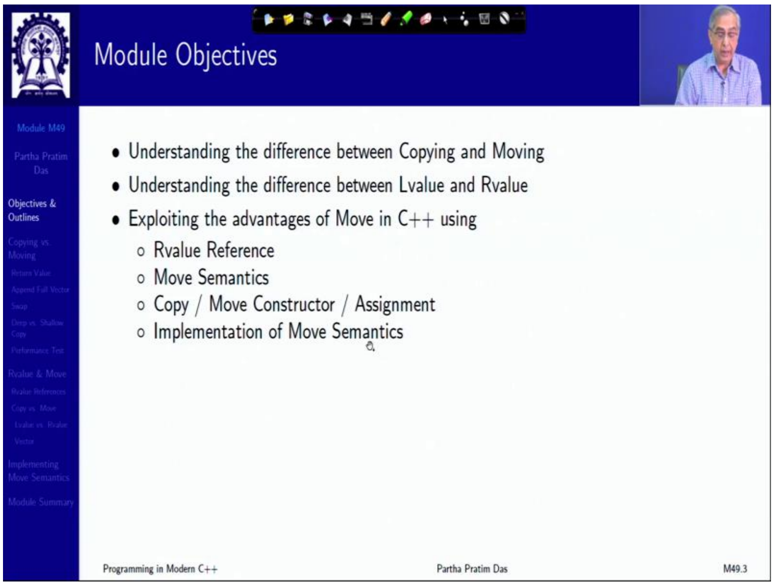
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 49
C++11 and beyond: General Features: Part 4:
rvalue and Move/1

Welcome to Programming in Modern C++ we are in week 10. And we are going to discuss module 49.

(Refer Slide Time: 00:33)

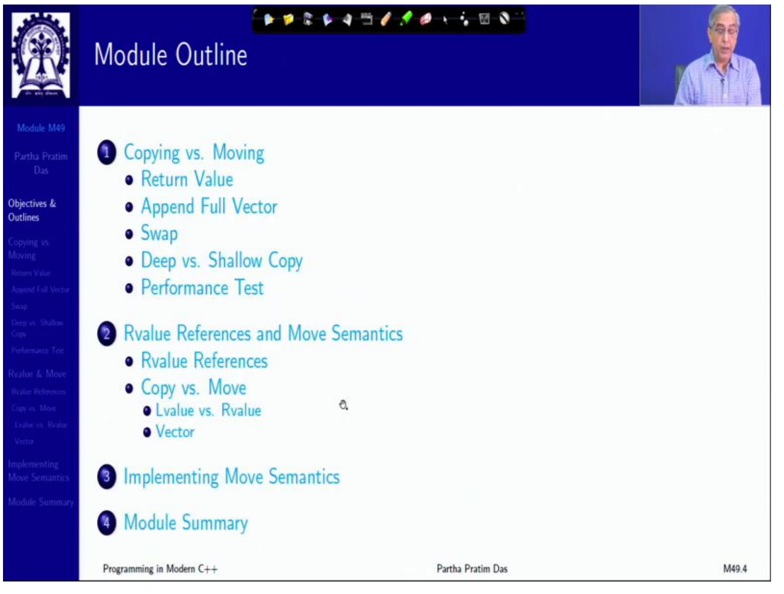


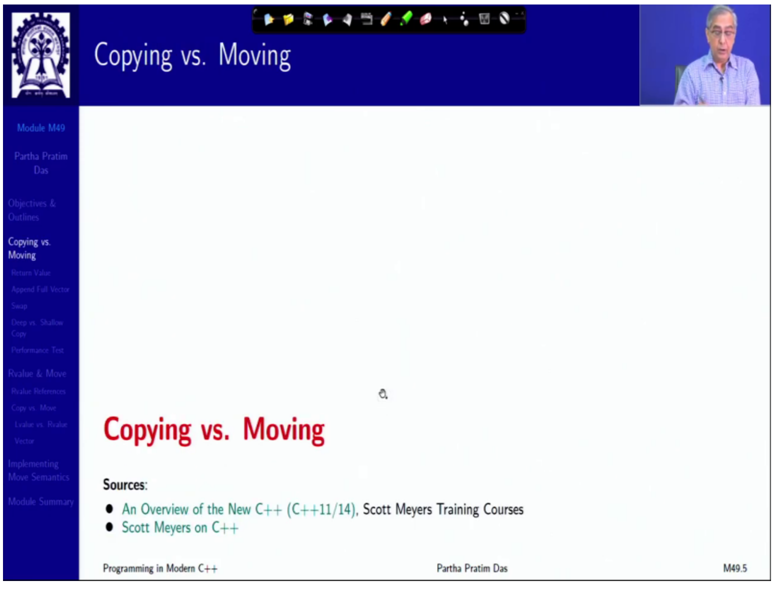


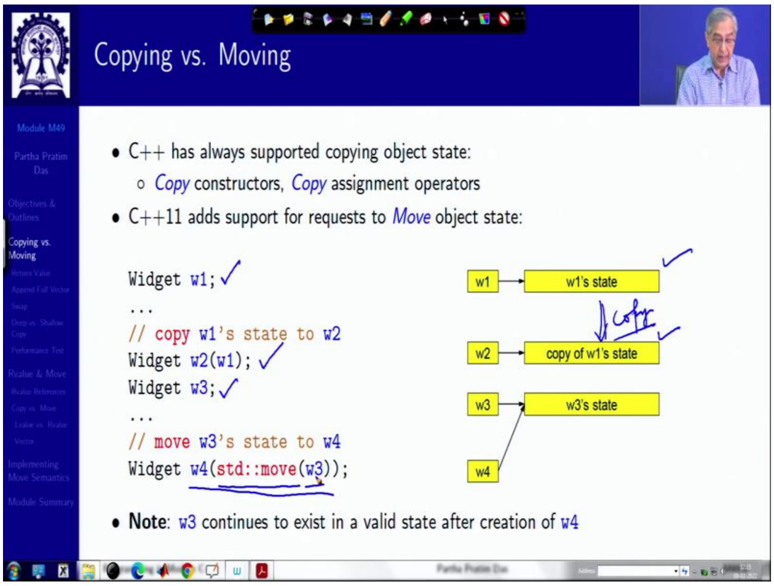
In the last module, we have continued to discuss different general features of C++11, these six features were discussed, they are diverse and kind of supports different requirements. In this module and the next we are going to discuss something which is also a general feature, but is fundamentally very, very significant for C++11 extension of the language, particularly for making it a lot more efficient in execution than it used to be.

So, we need to for this, we need to understand the difference between copying and moving something so fundamental. And the difference between lvalue and rvalue. And we take advantage of move in C++ using what is known as rvalue reference and to move semantics.

(Refer Slide Time: 01:23)



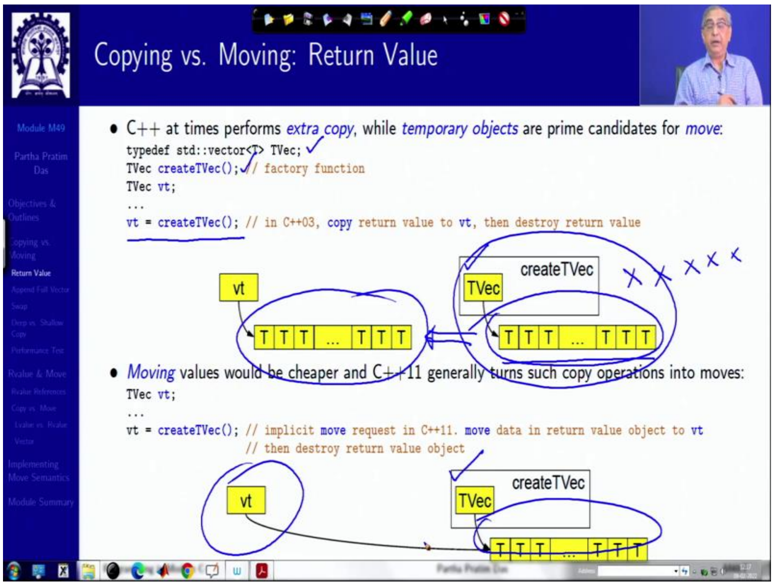




So, this is the outline. So, let us see the difference between copying and moving. So, C++ has supported copying of states of objects, we know that and C++11 is providing support to request moving objects. So, what is the difference? Suppose, we have an object of object w1 of widget type. So, I can use either we can use this kind of way to construct object w2 where the copy of the state of w1 will be created. And w2 will basically mean that it is a copy of this.

Now, let us create another widget object w3. And let us write something like this what it means we will come to that we say std::move(w3), by that what we mean is we do not want to copy the state of w3, but we want to move the state of w3, which means that after this both w3 as well as w4 will share the same state. So, this this, this kind of sounds like the issue of deep copy and shallow copy, which will come to very soon.

(Refer Slide Time: 02:51)

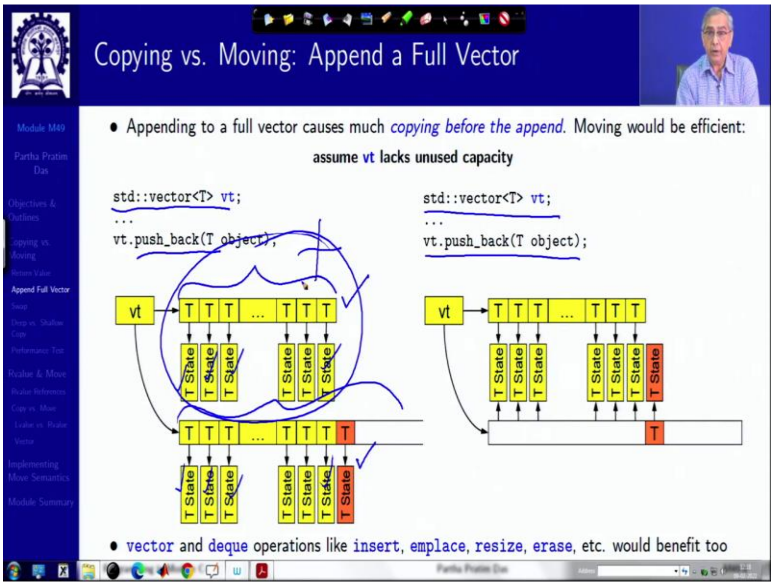


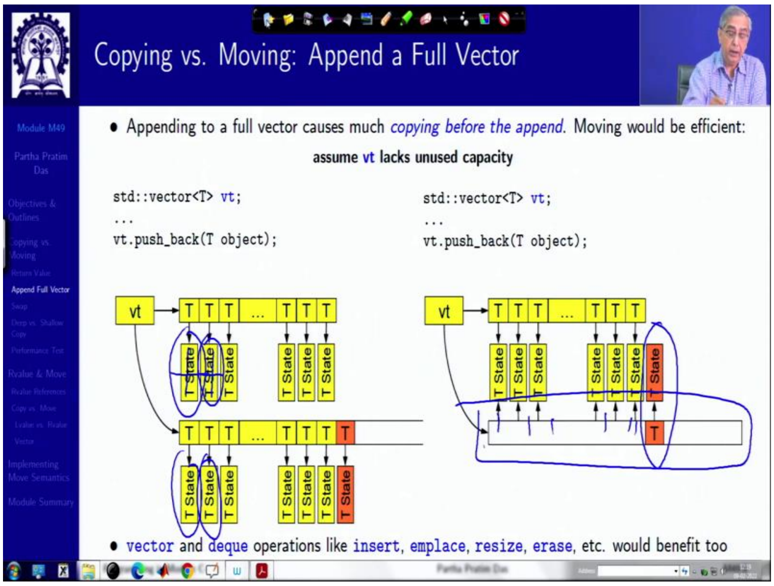
Now, this becomes useful particularly for say a return value. So, in C++03, if you have a vector and you have a create vector function, which creates a vector and returns you then if you initialize this vt with the create vector, the create vector is returning a vector, so that vector will be copied.

So, create vector is returning a vector and this vector will be copied. So, all elements as many as are there will be copied. Now, in C++11, if I can make use of the move semantics move request, then I can make it that whatever is returned by the vector will not be copied, but that returned object itself will be moved into vt. So, what is the advantage that two advantages one is -- see or rather one primary advantage is that for making the copy, I have to make copy of so, many objects the vector could be really large.

So, so many constructions and after that this is a return value, this is the return value of create. So, this cannot be used any further this cannot be accessed any further. So, I have to delete each one of them. So, I have to copy make copies and then delete them, which is useless. I can just use these objects, just take them directly. Because in any case, since is a return value does not have a name, I cannot reuse it in any other context. So, that is that is the insight between copy and move. I can move and get much better performance.

(Refer Slide Time: 04:44)





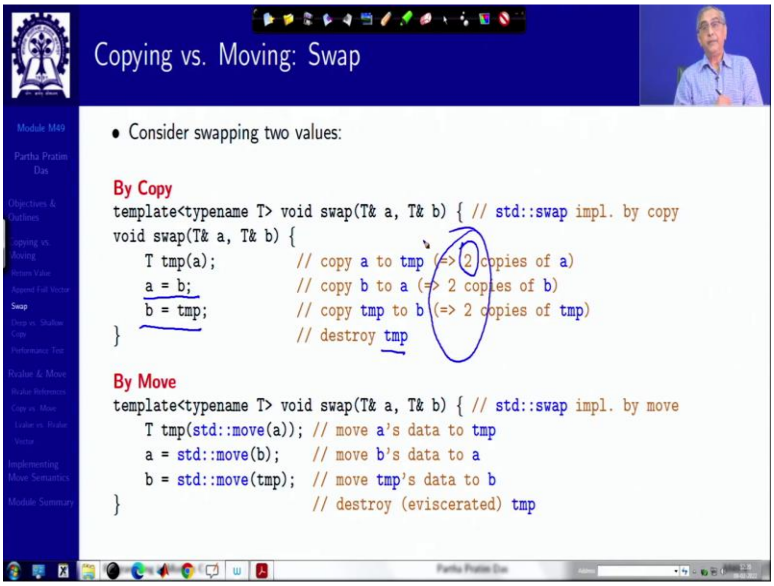
Think about appending to a full vector. Suppose I have a vector, say again. And suppose it has become full. What happens if the vector becomes full and I try to do a push_back I am trying to do a push_back of it, of a some T object, appropriate object. What we will have to do? It will have to create another new space for the vector, copy all the existing elements, and then put the push_back element at the end.

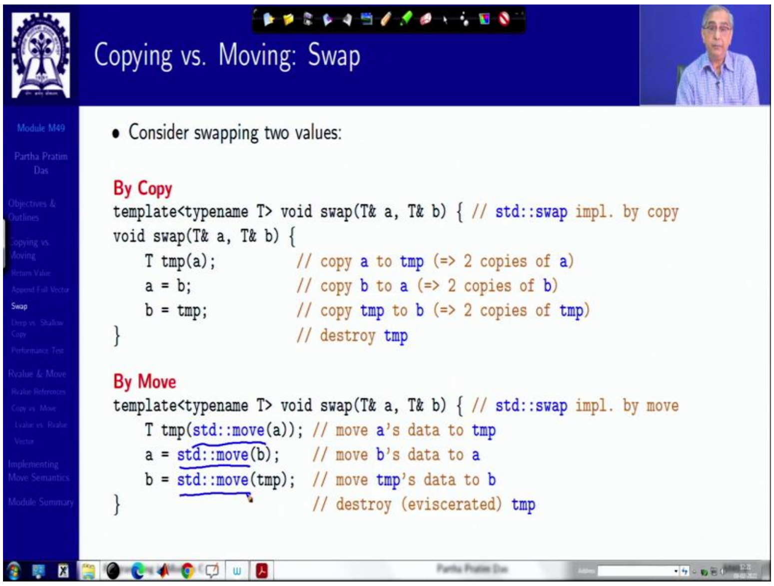
So, this is our original state of the vector, all these T objects were existing, since it has run out of space, I take a bigger space, and then I copy each object, this entire vector, and then I add the new element, this is what is freaking. Extremely expensive, because as many elements are there, those many copies and there deletes all of these will have to happen.

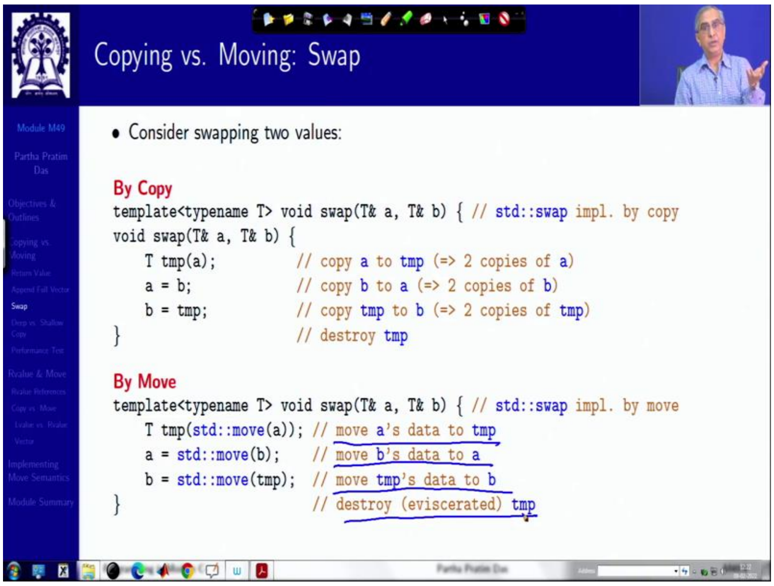
Instead, the vector is going to change. And after this, this has been done, this old vector is of no use. So, if you are, in any case, going to delete it. So, why not, we do a move. That is we do not copy and construct the object and delete the earlier one, do not take this, say, I am taking this making a copy, deleting this, taking this making a copy deleting this, instead of I will just have the new vector allocation.

And let those existing objects be might, without actually doing copy, just moving that data thing, no creation, no destruction, and then have the new one. So, it can be tremendously great use if I could make use of this move. The question, obviously, is to decide when to copy and when to move.

(Refer Slide Time: 06:52)







So, that is that is the tricky part, which we will have to think about a very common function that we always write swap, swapping, two variables that templatized function A and B have the same type it swaps. And this is a code that all of us have learnt initialize a temporary and use that to swap. So, what happens when you initialize when you create the temporary with a now we are copying a to the temporary, so there are two copies of a, one is in a one is in tmp.

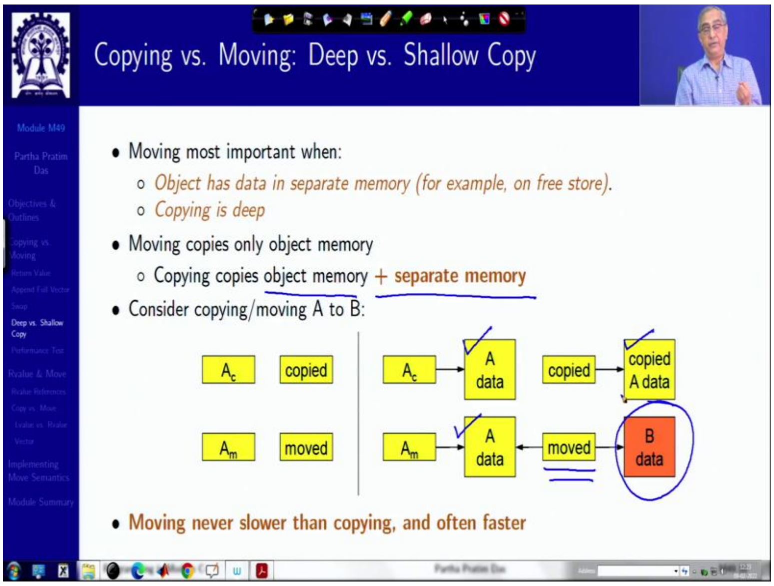
Then you are making a copy assignment to b, you are copying b to a. So, what happens there are two copies of b one in a and other in b and finally you do copy tmp, copy assign tmp to b, so there are again two copies of tmp, one in b, another is in tmp and then you destroy them.

So, every time you can see that you have 2 copies, which is unnecessary. Because you did not, you did not want that all that you wanted is they just get swapped. So, kind of if I had pointers to them, logical thinking is if I had pointers to them, I can just swap these pointers in terms of actually changing the objects.

So, you can do that in C++11 by just saying this std::move, we just saw some time back, which tells that do not copy the object, but move the object. So, initially, I need this, I need this temporary to do the swap. So, I create the temporary, but I do not copy the state of it, I move the state of it into that. So, a becomes tmp is now holding a, a becomes free, it is not holding anything meaningful. So, then I move b into a.

So, a now holds that state of b and b becomes free does not hold anything meaningful. So, I then move tmp into b. So, b becomes takes the state of tmp which was the original state of a and tmp does not hold anything meaningful. So, I can destroy tmp without doing anything. I do not just need to I did not do a copy. So, I did not take resources I did not create resources. So, the destruct here is just dummies just call.

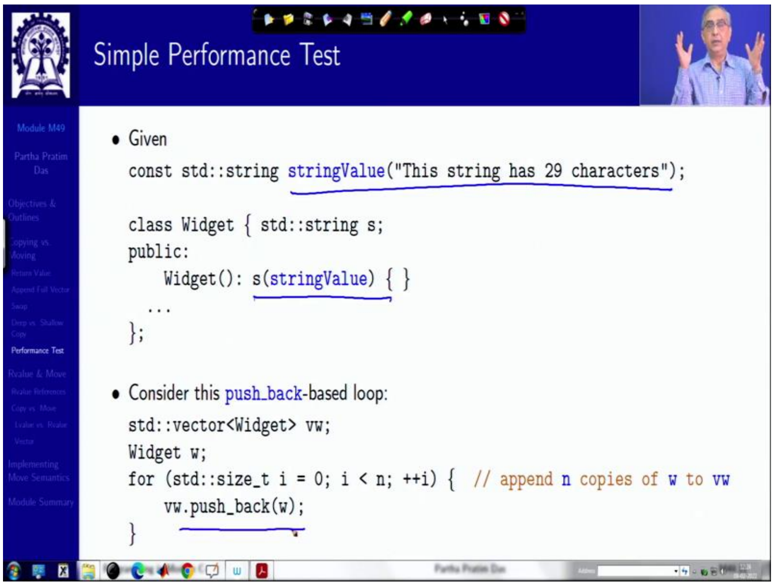
(Refer Slide Time: 09:37)



So, copying versus move is a basic question, which is deep. If we have, if we see the distinction between deep copy and shallow copy. Copy is in our connotation always deep copy we have ensured that so if I have an object, then it is in memory and it has resources like it is pointing to 10 other things 10 different pointers to different other objects, so they are in a separate memory. So, when we talk about copy, we mean a deep copy that is copy the object memory as well as the separate memory.

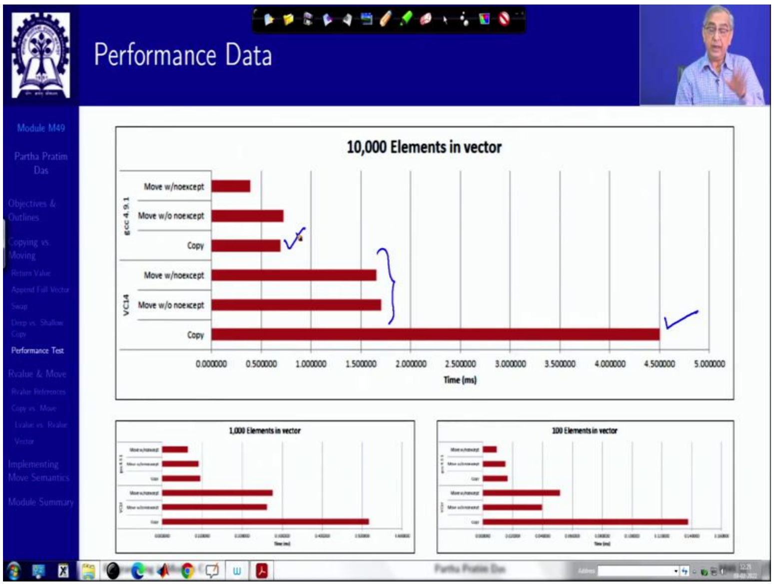
So, if I have A then I copy, then I have A’s data. And I will also have a copy of A’s data in the sep…, from the separate memory that is the deep copy we say. So, we copy the pointed objects as well, where is a move, I will have the A’s data the object memory and for move, I will not have the copy of the A’s data, but I will use B’s data itself and not create that separate memory again, just I take that memory. So, by that so move obviously invalidates the source, which copy does not. So, whenever the source from where I am I want to make a duplicate off is not required after this operation, I can certainly do move instead of doing copy, certainly it will be always faster to do that.

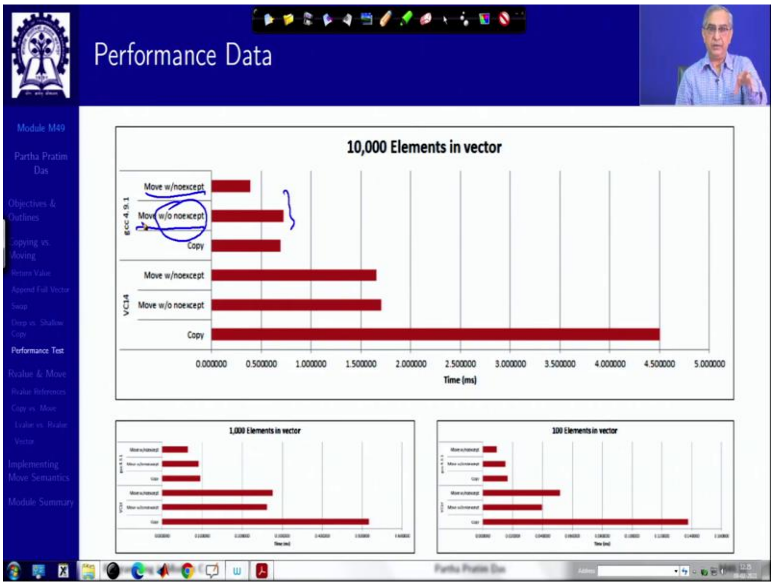
(Refer Slide Time: 11:20)

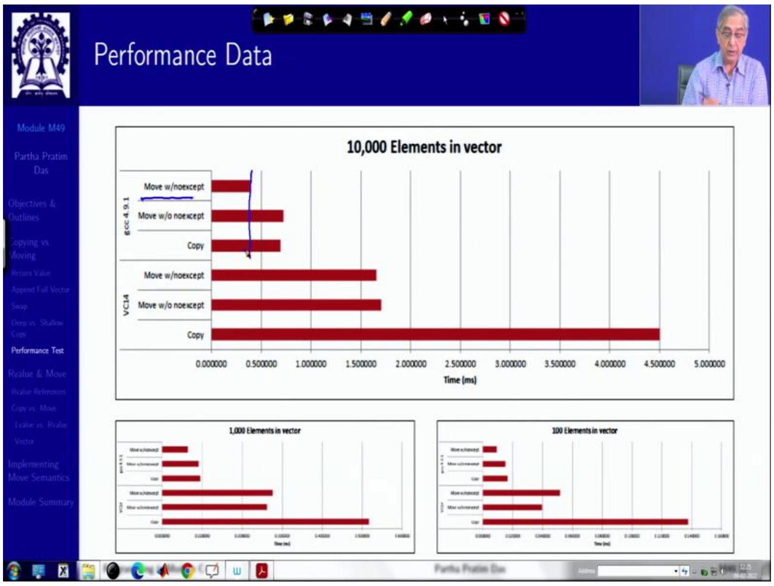


So, just a performance test, which was done decades back. So, here is a widget which has a string constant string value and it just constructs that and, it tries to do a push_back of the same value in the in a vector of widgets repeatedly it is repeatedly being pushed back is just a bulk workload to show that how does copy so in generally will copy the for the push_back, how does it impact.

(Refer Slide Time: 11:58)



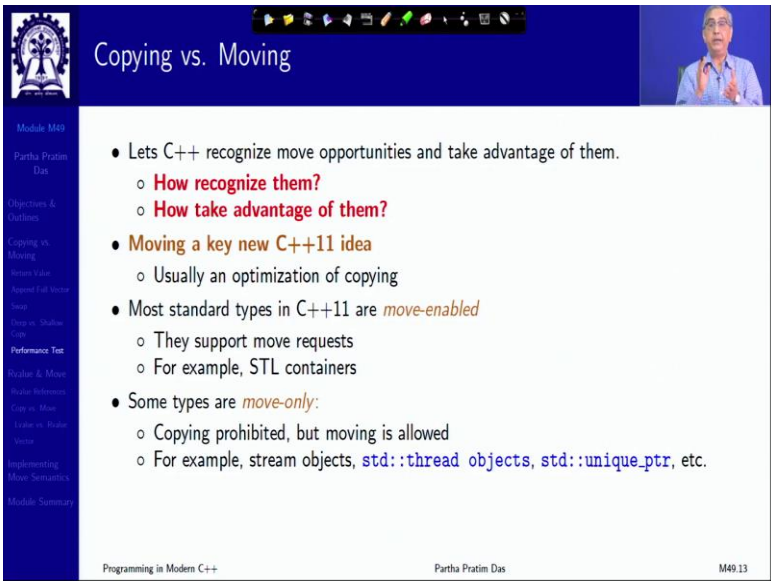




And, results are really stunning, you can you can see that with copy, if you take this much without copy with move you take this much less similar thing happens here obviously, in terms of Visual Studio compiler, you see a much bigger advantage there are some reasons for that, but the basic idea is being able to move instead of copy and you can also see the difference between move without except, move without noexcept and move with no exception.

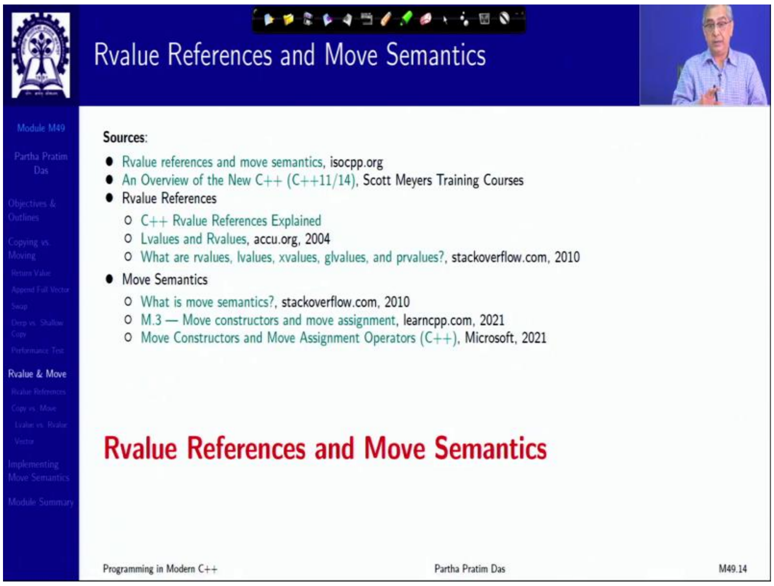
You can see that in GCC if you if you do not have no except, then you may not get much benefit, because you have a lot of exception possibility of exception handling code a data structure that you need to deal with. But if you use no except as you are doing here, then you get almost about half the required time.

(Refer Slide Time: 13:04)



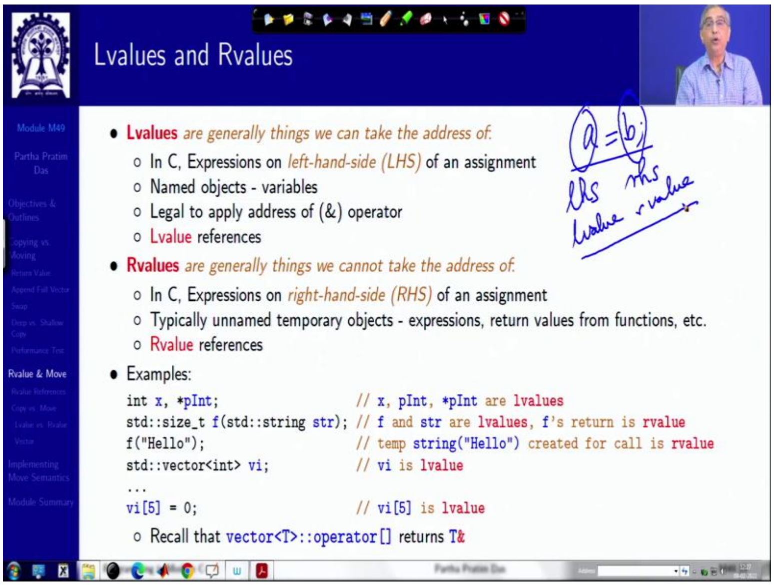
So, performance gets so that is the objective with which C++11 has focused on the semantics of move along with the semantics of copy. Now, the question naturally is how to recognize them, how to take advantage of them. And for that C++ standard has made the standard types move enabled and some of the types are moved, you cannot copy them you can just move them we will talk about those more in future.

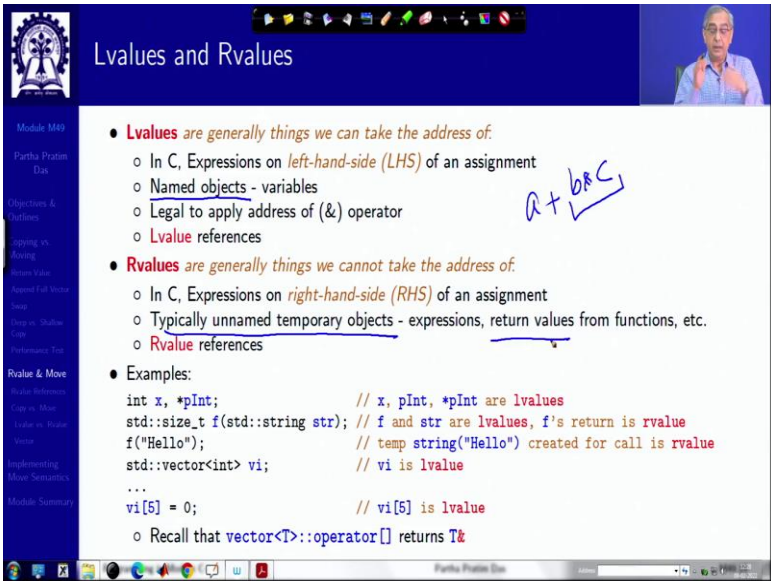
(Refer Slide Time: 13:31)

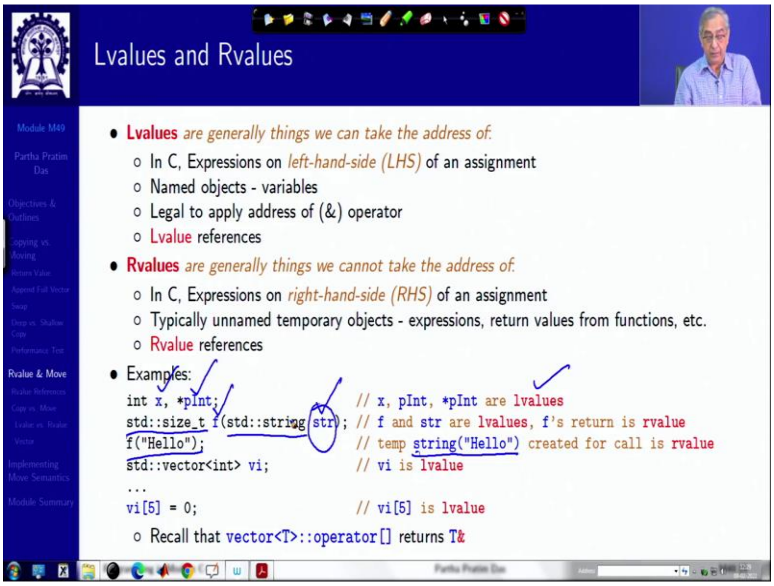


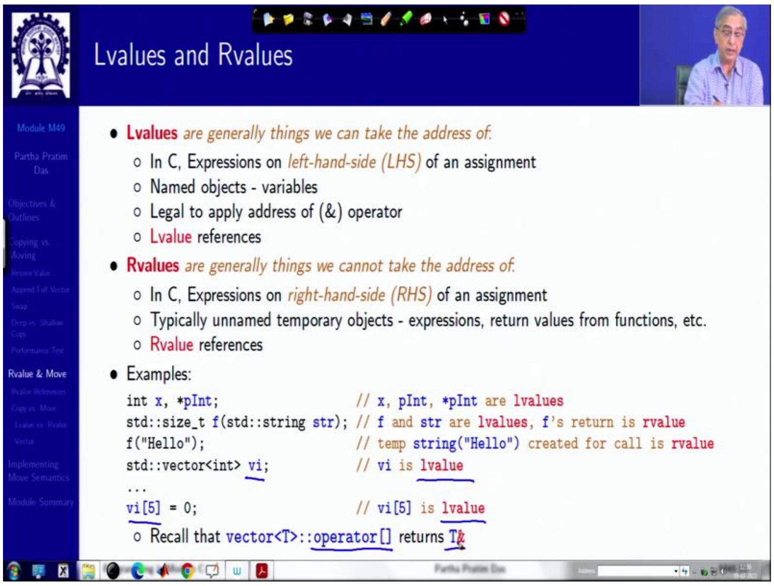
So, for this what we need to understand and I would request you to be very, very attentive and really focus on this because this is something which is simple, but the core of C++ move 11 performance.

(Refer Slide Time: 13:44)









What is rvalue and what is move semantics? So, in this let us understand that what is an lvalue and what is an rvalue. So, lvalue, rvalue, this name were given in terms of C mode in terms of C programming lvalue is something that occurs on the left hand side of an assignment and rvalue is something which occurs on the right hand side of the assignment.

Now, what it means? Is if I am doing b assigned to a, then for b, I need the value of b but for a I need the address of a where it has to go I have to locate a. So, this is the left-hand side this is the hand side and this is what we call lvalue, this is what we call rvalue. Coming to C++ connotation this besides assignment there are several contexts where you need to talk about lvalue and rvalue. So, the this basic left hand side of an assignment connotation has reduced so often, now we talk about lvalue as locator value which can be located and rvalue is something which is not an lvalue.

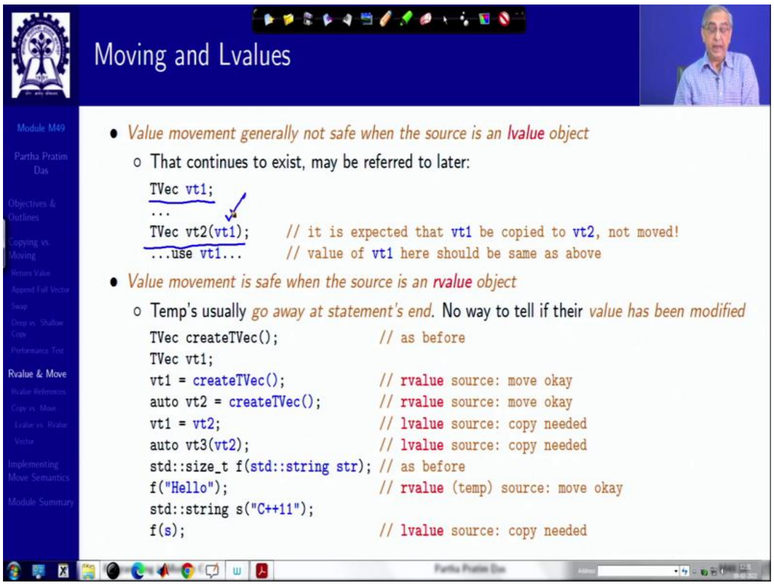
So, with that distinction, if we look at then these are lvalues are named objects variables, which you can catch hold off make computations with us the value in future. rvalues are typically unnamed temporary objects, which exists which has a value, but there is no way that you can catch hold of them like expressions a + b * c, naturally all of this cannot happen in one go, this happens and the value is generated with which you are adding a.

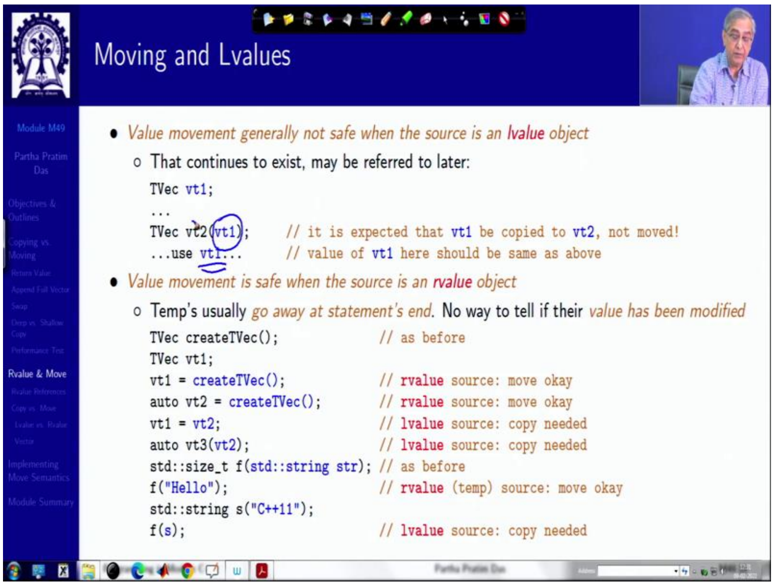
But can you access that value? No, this is an unnamed temporary thing. Similarly, return values from function these are all or different rvalues, and then you have lvalue reference and rvalue reference here are examples for your understanding int x, int *pInt, these are naturally lvalues you can catch hold of them, f() as a function is an lvalue, str as a parameter is an lvalue.

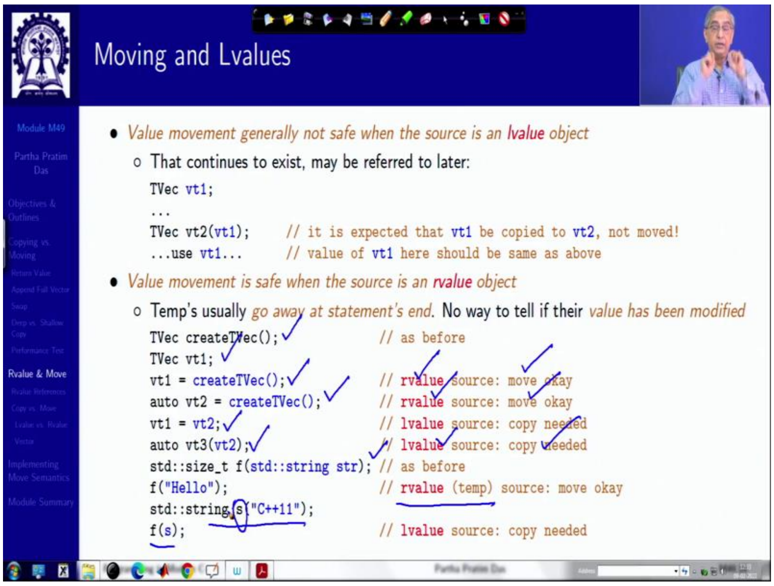
Whereas, f’s return value if the return value is an rvalue, because either you copy and keep it or move and keep it or it actually gets lost at the end of the function called this will get lost. So, in this you have a, you have a Hello within double quotes, which means it is a constant char*. Now, from that a std string has to get constructed to be called to that function, there is a construction involved. Now, that object which gets constructed and passed the str you have no hold on that object that object will get created will go to the function and will get destroyed after that after the function.

So, the str std::string of Hello, that object is not exist is a temporary and you will not be able to catch hold of it. So, it is an rvalue. Similarly, vi if I define it is an lvalue vi[5] is an lvalue. And that is the reason you will see that if you look at the in vector, if you look at the access operator operator, square brackets, it returns a reference because it is an lvalue.

(Refer Slide Time: 17:36)







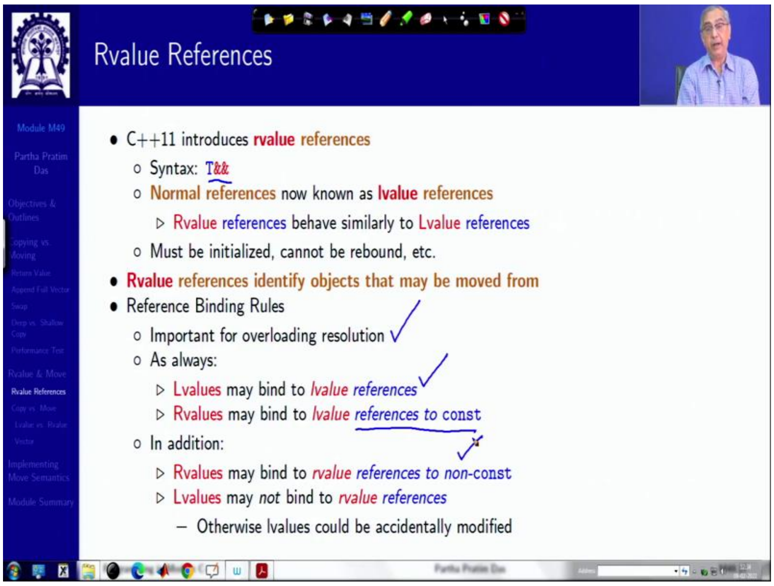
Now, how do you move what is the consequence of moving movement of values in terms of lvalues and rvalues. So, if you move a value, when it is an lvalue, that is often generally not safe. So, you have a vector vt1, you are creating another vector with vt2 with vt1. Now, there are two choices as we have seen, we can copy the state so that vt1 remains valid, and vt2 is a copy of that vt1 or we can move the state so that we do not have to duplicate both vt1, vt2 is the same state.

Now, the risk of doing that is vt1 is an lvalue, I have a name, it is a named object. So, subsequent to this construction, subsequent to this construction of vt2, I can still use vt1, so vt1 and vt2 needs to be different. So, move here is not something which is smart. Whereas if I have values which are rvalues, which are temporary, it is safe to move them. I have this function, I have an object vt1, and I have called this function and the returned object from createTVec. I am assigning to vt1. Now, the return of TVec is a temporary object which will get destroyed anyway at the end of this call. So, if I move that to vt1, I do not lose anything. So, it is an rvalue and naturally move is okay.

Similarly, if I do an initialization of vt2 not an assignment using the return value of TVec it is again an rvalue and moving is okay. Whereas, if I assign vt2 to vt1, then vt2 is an lvalue. So, I need a copy. If I do a copy construction of vt3 from vt2, it is again an lvalue I need a copy, If I have this function f() as before as we have seen Hello is a rvalue, a temporary value.

So, move is okay if I have a string like this then f() calling f(s) need to have a copy because I have s as a named object representing the string within double quotes C++11. It represents a std::string and I can use it subsequently for that. So, that is a judgement point that you would have to see that is there a way to use that object, subsequently if it is then it cannot be moved, it should not be moved, then it should be copied. So, move is not good for lvalues, but move is excellent for rvalues.

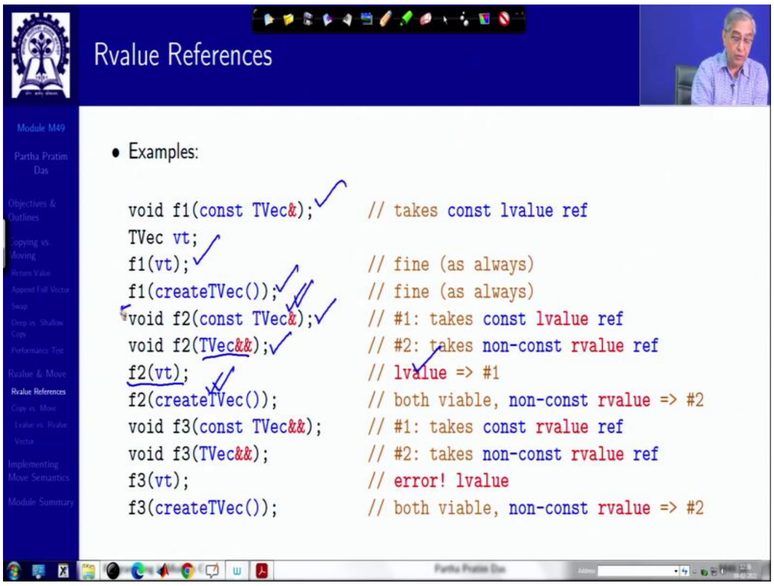
(Refer Slide Time: 20:49)

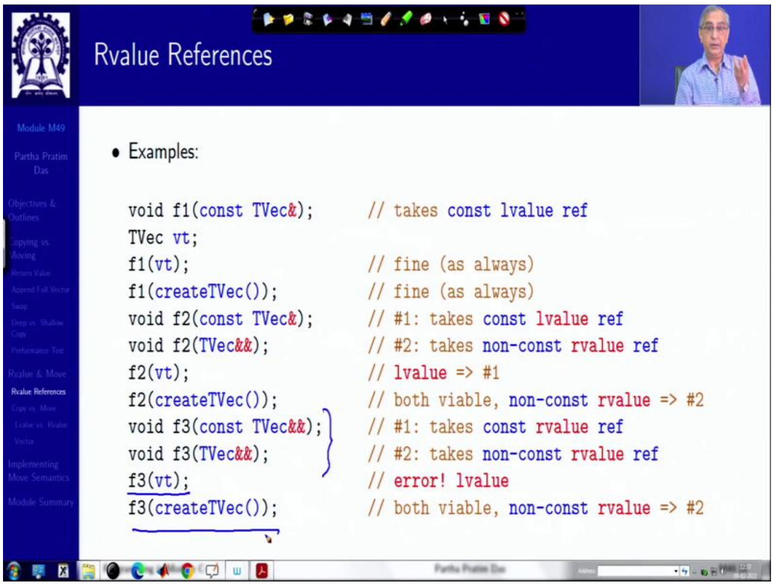


So, to be able to detect that a different kind of reference is introduced in C++ it is known as rvalue reference in the syntax, it just uses the ampersand twice. So, normal references as we have known them is now known as lvalue reference. So, rvalue reference behaves very similar to lvalue reference, rvalue reference will identify if I hold an rvalue reference to an object I know that I can move from that object that is I should move from that object.

So, it can be used in overload resolution, rvalue and lvalue references. lvalue may bind to lvalue references, rvalue may bind to lvalue references to constant that we have seen that you cannot pass an expression where there is an reference, but you can pass a constant expression, because you need to have that reference to be identifiable by the by the parameter name. In addition, rvalues may refer to rvalue references to non-constant and lvalues may not bind to rvalue references, because if they do, then the possibility of the move or possibility of further changes will be accidental.

(Refer Slide Time: 22:21)



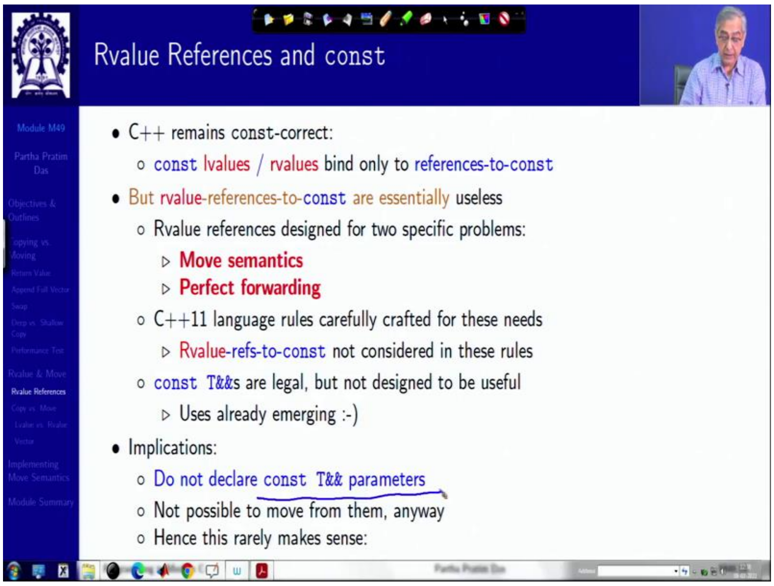


So, again a couple of example, here, you have this as a constant lvalue reference in f1. So, if you pass vt to that, it is fine, if you pass this to createTVec call to that also is fine. So, what we are doing is are basically taking a you need a lvalue and you have lvalue reference and you have passed it rvalue. So, rvalues can be converted to lvalue constant lvalues.

If I have f2() of two times which one which takes constant lvalue reference and other which takes non constant rvalue reference, then if I do f2(vt) it will is taking an lvalue. So, it will call this form if I do f2(createTVec()) then both are viable, it will take the rvalue because it is it is an rvalue. So, it can it can take an rvalue reference. So, having this overload allows me to differentiate between whether I can treat it as an lvalue or I can treat it as an rvalue. See more here you have a constant TVec rvalue reference and non-constant.

So, a constant rvalue reference and non-constant rvalue reference and you try to pass an lvalue to that now, this is an error because it tells you to actually move the object, but being an lvalue, you cannot move that object. So, this conversion will give you an error. Whereas if you do f3(createTVec()), then you have a rvalue, so both of these are possible non-constant one that will be preferred for reason that we will come to very soon.

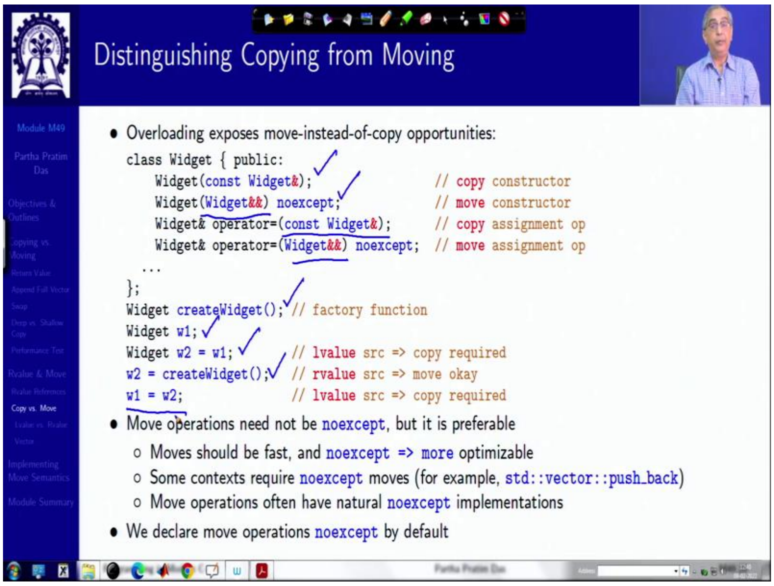
(Refer Slide Time: 24:40)



Now, the question is, what is what about const-ness? The const lvalue or rvalue bind only two references to const but rvalue references to const are essentially useless. Why? Because why did we identify rvalue because we should be able to move. Now, if we are saying that I have a reference to an object which is an rvalue, but to be treated as a constant, then naturally I cannot move it. Because to be able to move, I need to make changes in that source object. So, it kind of contradicts the semantics of it is it is legal, though it is its semantic use is really not understood so well till this time.

But it is typical that rvalue reference to const is not considered right now, but it is not illegal to write it will, but it will not let you do that move that is the consequence. So, you should not do that. And this rvalue references solve two specific problems, which is move semantics and perfect forwarding. So, remember this part that you should never declare the const reference rvalue reference parameter.

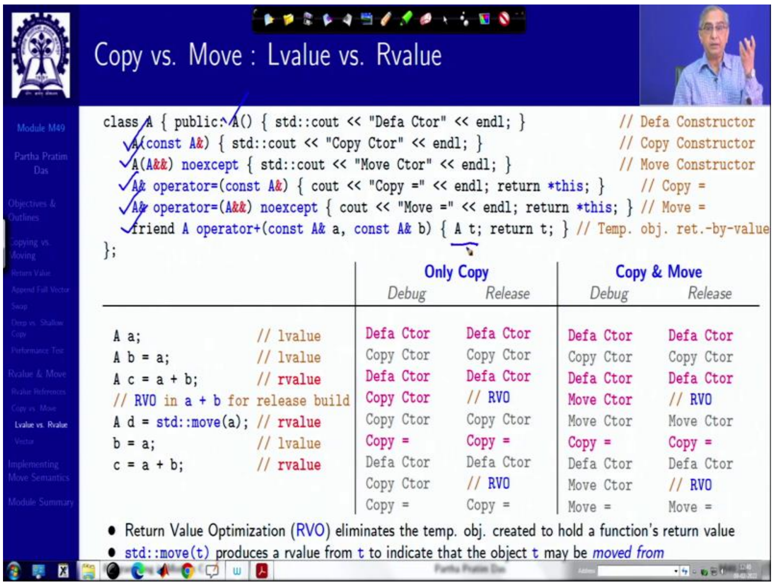
(Refer Slide Time: 26:10)

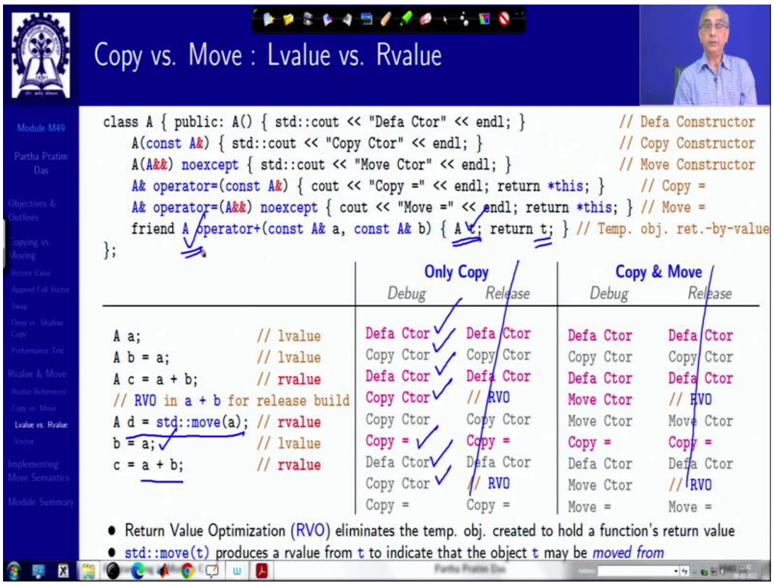


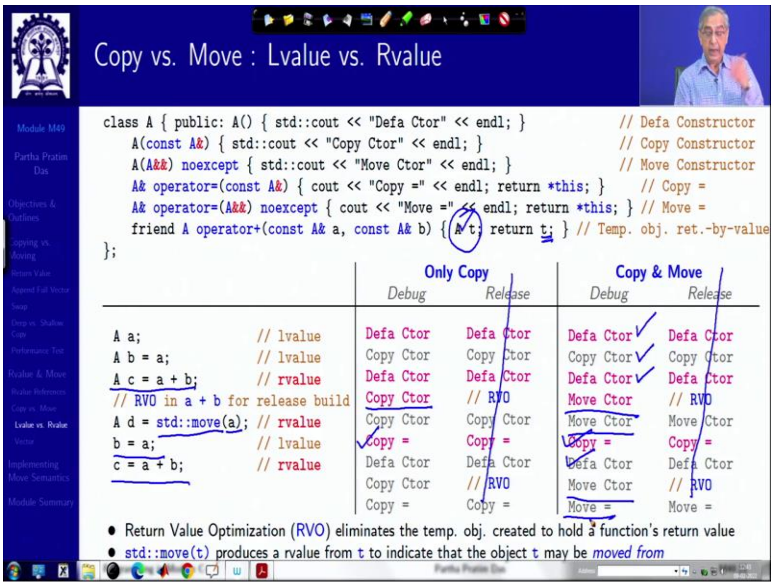
So, differentiate this we have a widget this is a typical copy constructor with lvalue. We write another with rvalue and we call that a move constructor. So, the difference is in the copy constructor, the source object state will be copied and the object created in the move constructor source object state will not be copied it will be moved and we need noexcept for optimization.

Similarly, this is copy assignment operator, this is move assignment operator, in the copy assignment, you make a copy of the source object in a move assignment you do not make that. So, if we have this function if we have the w1, we are creating w2 with that, so, w1 is an lvalue. So, copy is required. But if I assign create widget result into w2 then the result is an rvalue. So, move is okay. Whereas, if I directly assigned w2 to w1 then a copy will be required. So, that is the basic difference.

(Refer Slide Time: 27:30)







And so, here I have given a couple worked out a couple of examples in detail to show you how does this copy and move constructor as well as assignment operator work. So, here is a class A which has a default constructor, it has a copy constructor, it has a move constructor, copy assignment operator, move assignment operator and binary operator which does not as such do anything it just creates a temporary the result object and returns that object in between of course, you will have the actual computation to be done.

Now, if you look at you will see that if you do only copy, then the calls will happen like this, forget about the release part initial. So, A is declared. So, default constructor if you have only copy a is assigned to b, the lvalue copy constructor a plus b assigned to c. So, what will happen first this operator will be called. So, A t will be created and then it will be copied.

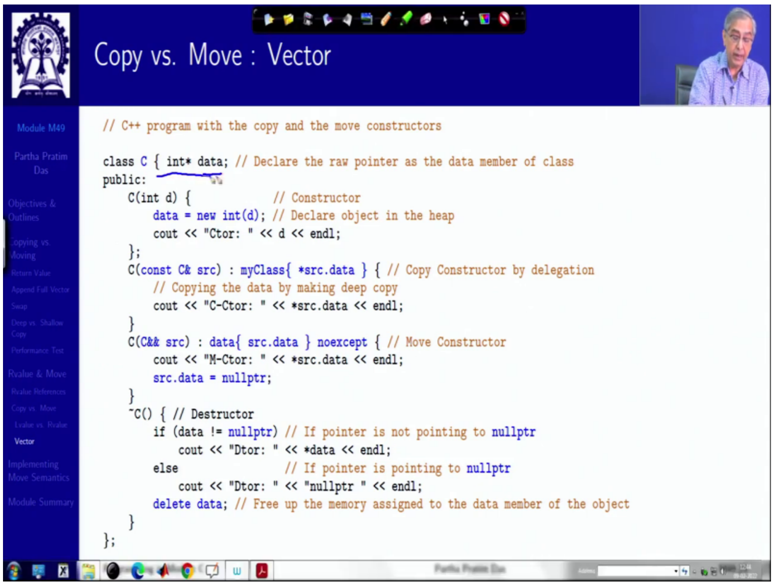
So, A t is created and then it is copied if I put this forget about this now, suppose I do an assignment of a to b the copy assignment suppose I do a + b and assign it to c default construction of t copy construction for doing the return by copy return by value you need a copy construction and finally the copy assignment happening here. So, this is what happens if you just have the copy constructor and copy assignment operator.

Now, let us say we have move constructor and move assignment operator also. We have not commented them out we also so this is same. This is same because it I have an lvalue. But when I create this I have a default constructor for this t but after that, for the return earlier I was having a copy construction. Now, I have a move construction because this return value is a temporary object, it is an lvalue.

So, I can just move the state, I do not need to copy the state. Then I have std:: move, which says that take this lvalue and a is an lvalue, but convert that to, rvalue converted to an rvalue reference. So, you can, with that, you can, you will have a move constructor, which will forcibly convert the lvalue to an rvalue and we will move.

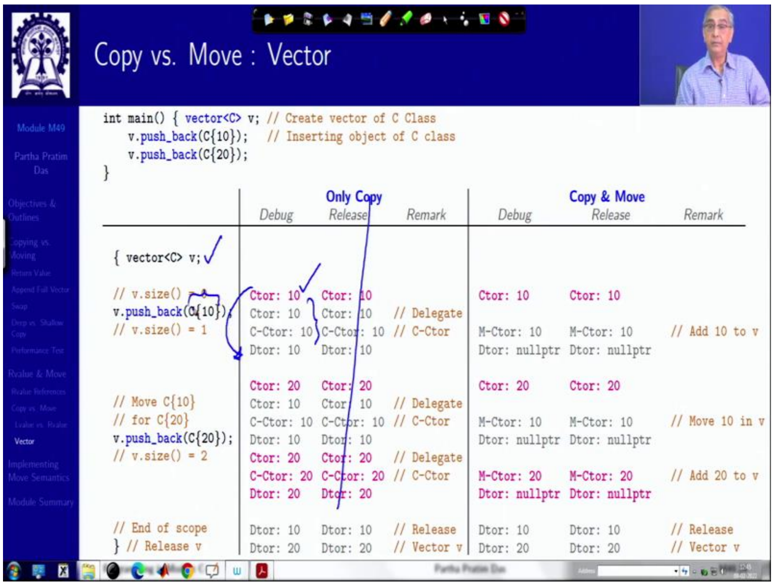
Similarly, in this assignment, I have an lvalue. So, copy copy. In this computation, I have a default, for constructing this, I have a move constructor for the return and then move assignment operator. So, this is how you start to benefit because you get a lot more of move construction and move assignment happening, which are much less expensive than you, actually.

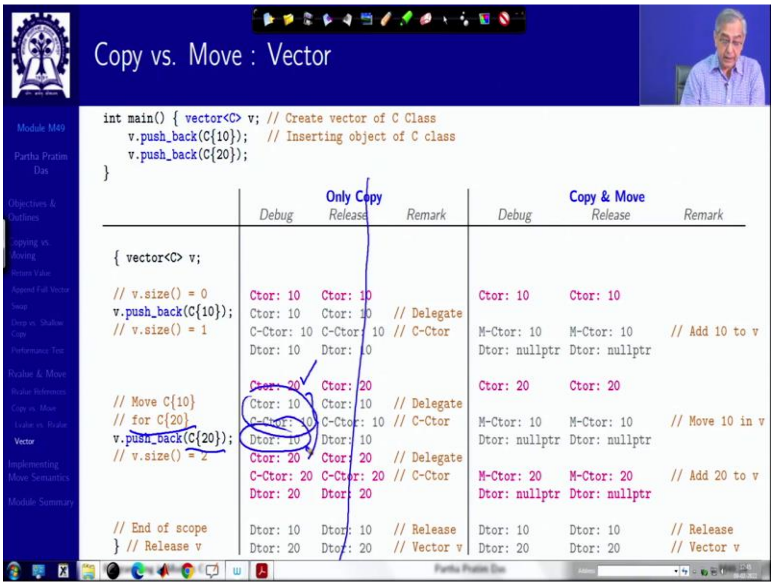
(Refer Slide Time: 31:05)

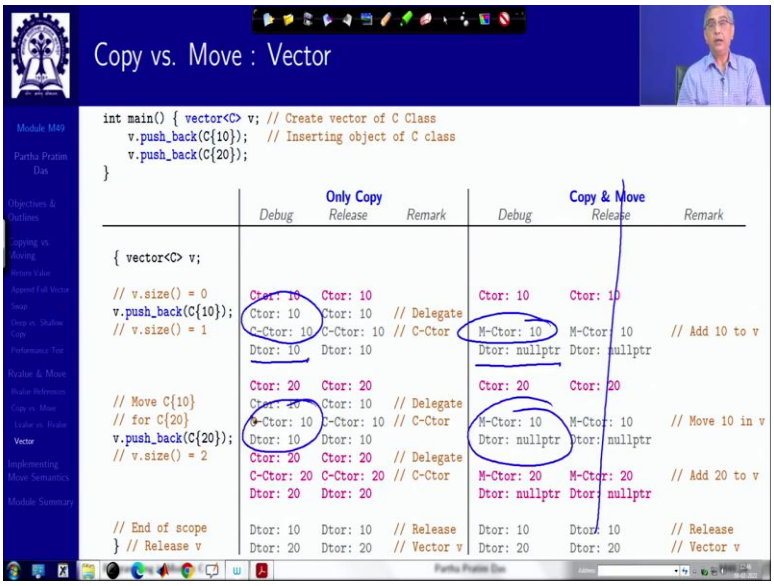


So, here I have given this detailed explanation for your self-study. Here is another example also using a class C, which has a resource in terms of an int pointer. So, with that you have a default constructor, copy constructor, move constructor and destructor.

(Refer Slide Time: 31:34)







And using that, if you create a vector and start doing push back, the main thing you will see is certainly, if you want to do a push_back, what will happen, initially, the vector has size 0, its got nothing, just this vector. Now, you do a push_back. So, it has you have to first construct, you have to first construct the temporary object C(10), that will have to be constructed.

And then you have to do a copy construction. So, here, the copy construction texts, uses the default constructor to construct and that that copied value will be put in the vector and the original lvalue, this one will get destroyed. Because this is an lvalue this is an rvalue, the temporary gets destroyed, that is fine.

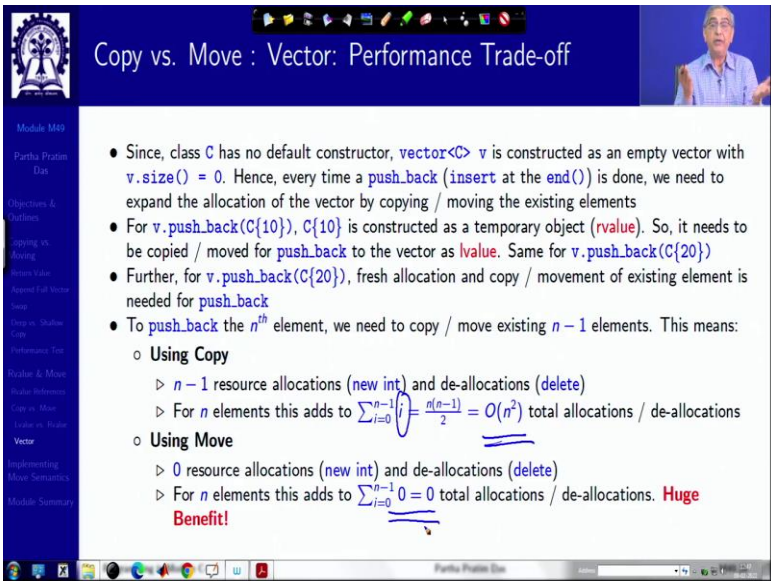
Now, if you want to insert the second one, here, the vector is already full. So, as we have seen, we have to make a move. So, create an object for 20. And then you have to create a copy of the existing value 10. And then destroy the earlier value, create an existing copy of existing, destroy the earlier value, and then make a copy of 20.

And you will have that so you have that additional task. In contrast, if you had just done move, then instead of copy construction here, you can just do with move construction, instead of doing a release, you do not need to do that, because you have already taken that resource. Similar thing, you will benefit in terms of moving the existing object instead of copying it. So, that is a basic advantage that you gain.

(Refer Slide Time: 33:33)

And again, I have given a very line by line step by step explanation for what is going on. So, use it in your self-study to get more insight.

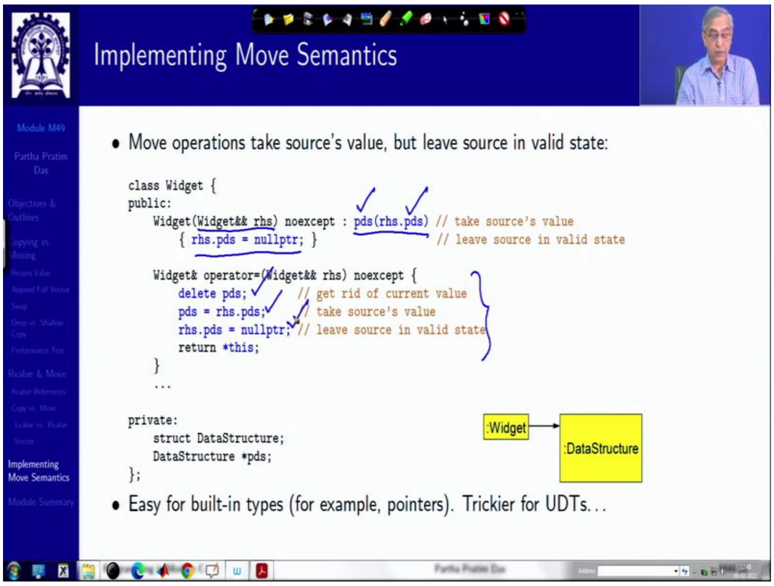
(Refer Slide Time: 33:45)



Now, if we just try to analyze, what, what advantage do we get. So, you can see that for the first copy, we can just we need to copy that object temporarily. In the vector for the second, I need to move the earlier object because the vector is full, and then copy the new one. For the third, I need to move, copy both of these existing objects, and then copy the new one. So, using copy every time for inserting the i plus first object I need to make i copies.

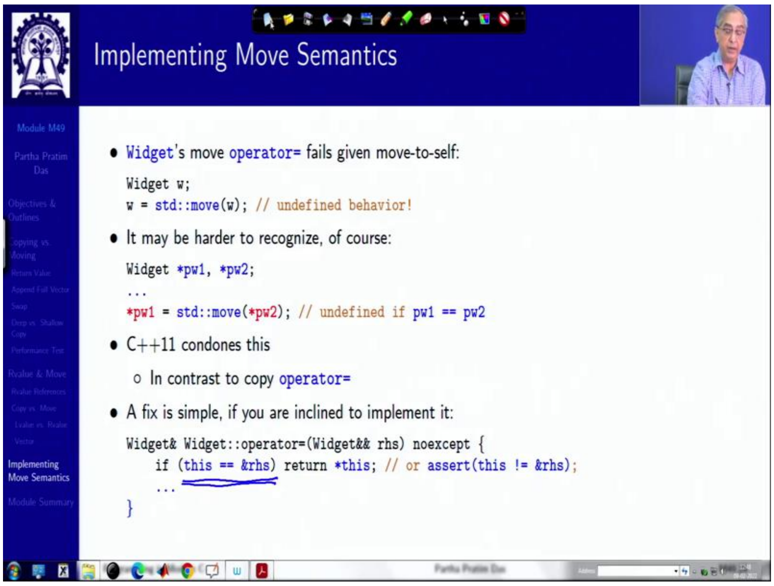
So, if I look at insertion of n objects, then there are order n square, total copy and release. That is allocation and de-allocation I have to do. Instead, if I use if I could use move, then whatever exists. I am just moving. I am not creating any new int* data resource, neither I am releasing them. So, it is simply 0. So, a huge huge benefit. And that is the incentive of why we should use the move semantics.

(Refer Slide Time: 34:50)



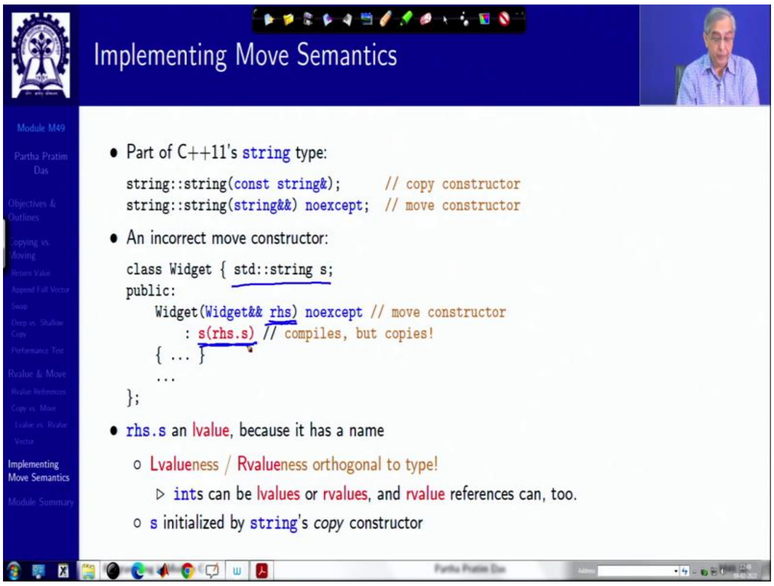
Now, implementing the move semantics is simple. So, I will just give you a glimpse here. And we will discuss more again in the next discussion that if you have a move constructor, then from the source, all that you are doing is you are just moving the source, take the sources value. And then you release it, let us set it to the null pointer. Similarly, free is an assignment operator, you can do the same thing release what you had copy and set the source to null. This will be simple.

(Refer Slide Time: 35:39)



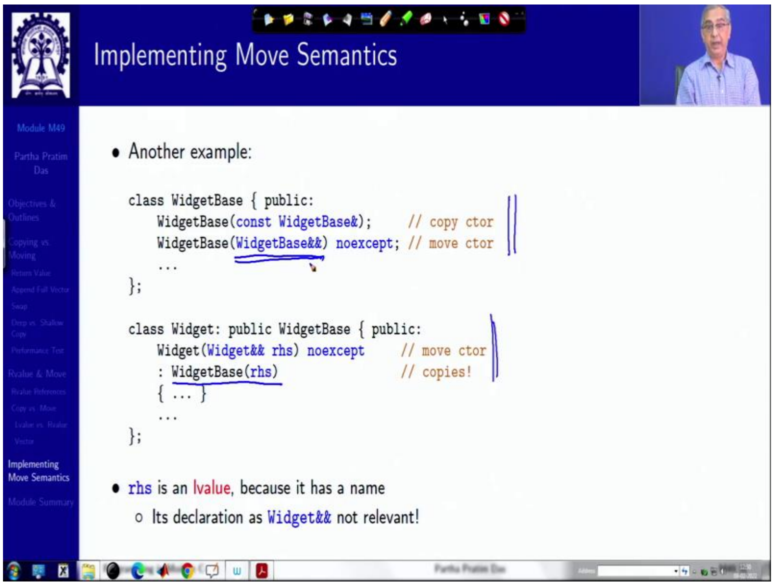
Now, the problem with this is a problem we had seen earlier in the copy assignment operator is that the self copy is a problem. So, the same thing will happen in terms of move also, that it is correct undefined behavior if you are moving from this object to itself. So, also in move, what you will have to do is to check that your source object and the target object are not the same. You already know this.

(Refer Slide Time: 36:10)



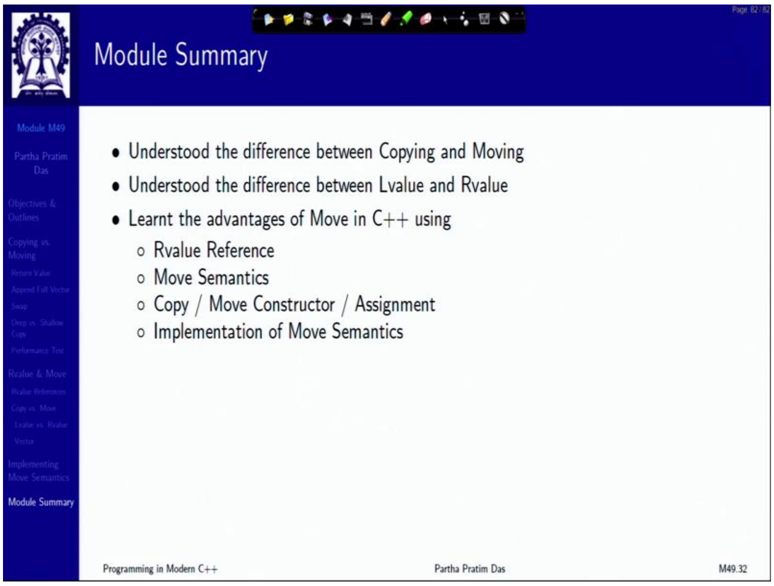
Rest of it is simple. But remember, this I will leave as questions in this, I will answer them in the next module, because I want you to think about this, that in a move constructor from moving from a source, I say I have a string s and I have written this this will compile. But this will copy you have to find the justification for why it will copy and how to solve that problem.

(Refer Slide Time: 36:48)



Similarly, if I take a different one, there is a there is a base type. And there is a derived type. And I am passing this RHS as a parameter to the base type, and I expect the move construction to happen, but you will find that it is not happening it will it will still copy. So, think over that as to reasoning as to what is the problem here particularly that the variables we are copying from our lvalues. So, that is that is this knave implementation will not work. So, we will have to see what more we will have to do.

(Refer Slide Time: 37:31)



And I will talk about that, but before that, I would expect that you have thought through this. So, here we have in this module, we have introduced something very very fundamentally important there is difference between copying and moving particularly lvalue and rvalue and the advantage of moving in C++ the rvalue semantics and the move rvalue reference and the move semantics. Thank you very much for your attention and we meet in the next module.