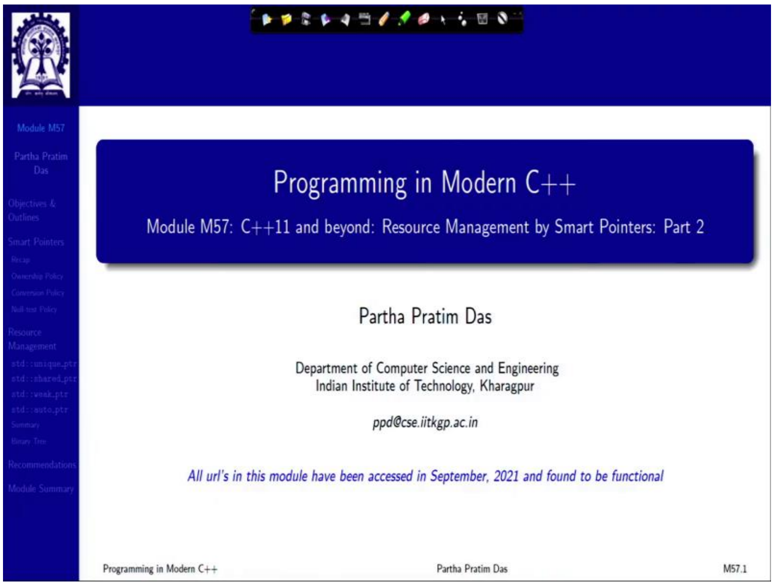
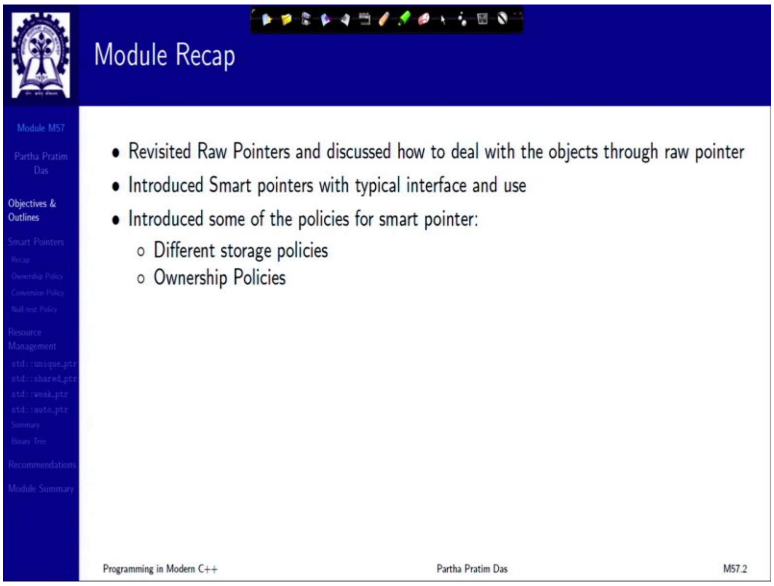
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 57
C++11 and beyond: Resource
Management by Smart pointers: Part 2

Welcome to programming in modern C++, we are going to discuss module 57.

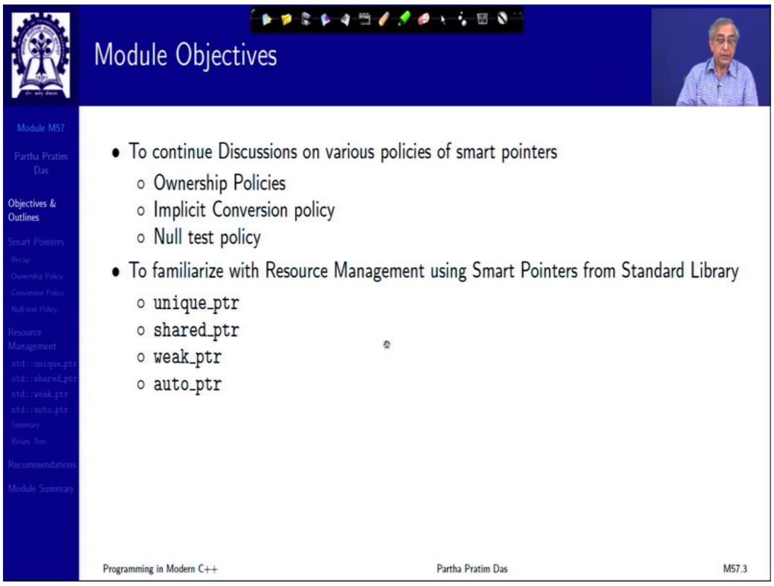
(Refer Slide Time: 0:33)





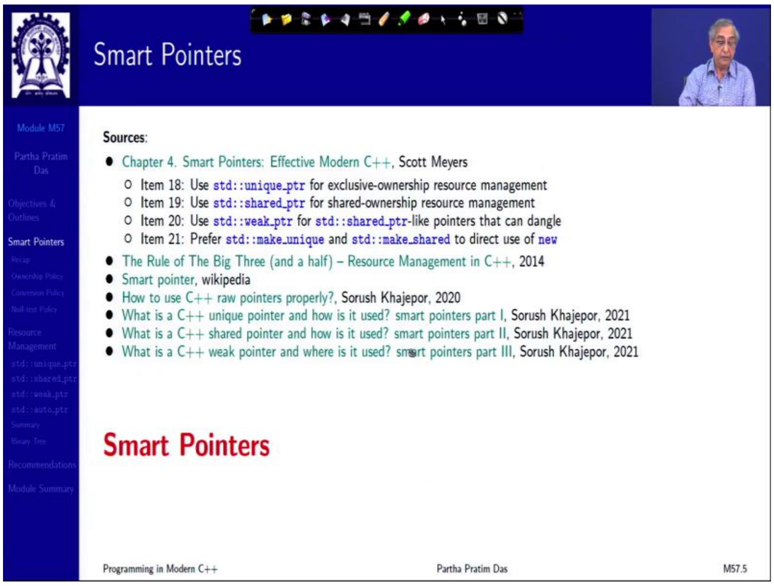
In the last module, we revisited the raw pointers and discussed how to deal with objects through raw pointers. And we have in a way, taking a look again at all the different problems that raw pointers cause in terms of managing the dynamic resources, which cause the safety issues as well as memory leak issues in C as well as C++ programs. Now, we introduced the concept of smart pointers with the typical interface and use and discussed about some of the policies.

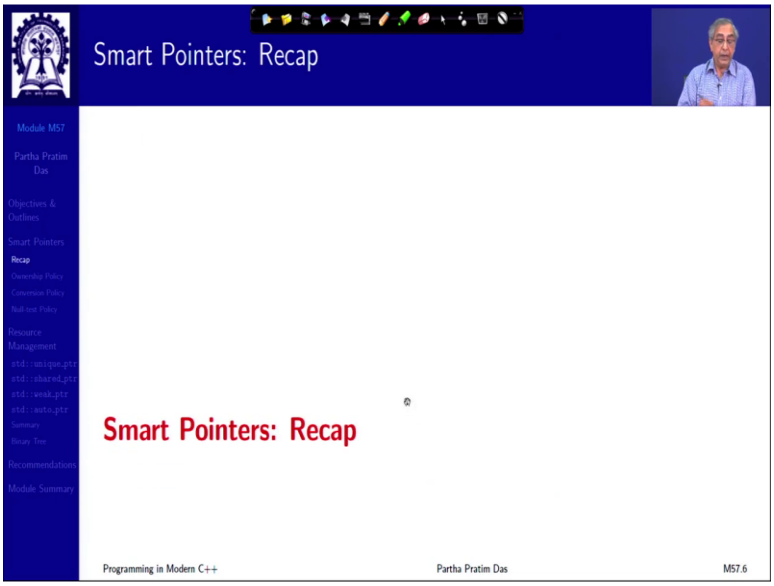
(Refer Slide Time: 1:17)



We will continue on that and complete on the ownership policies and discuss other policies and then take a look into the standard library support for C++ for smart pointers that are useful for the resource management. These are the contents will be outlined will be available on the left.

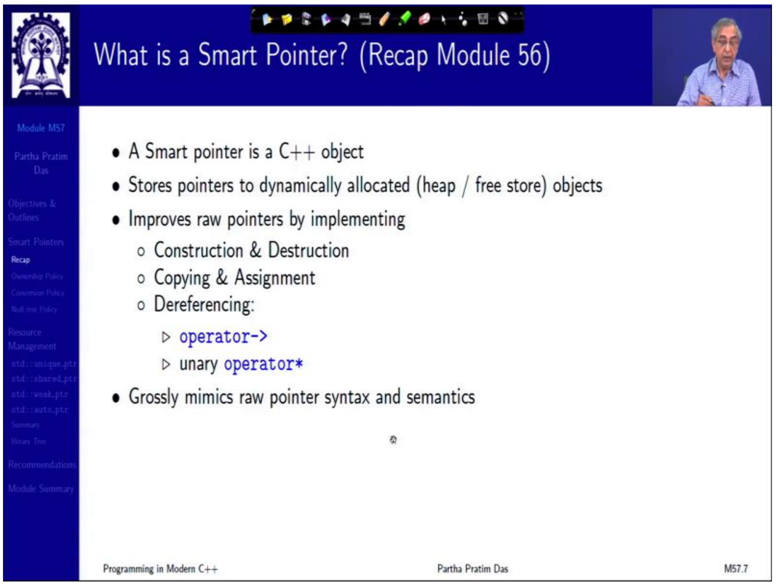
(Refer Slide Time: 1:38)





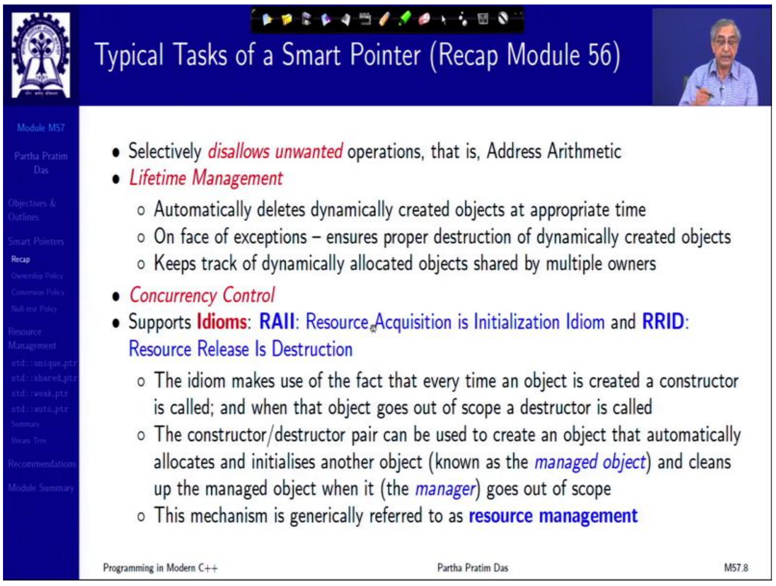
Now, before we start our discussion today, just let me quickly take a do a recap of what we have done the key points.

(Refer Slide Time: 1:48)



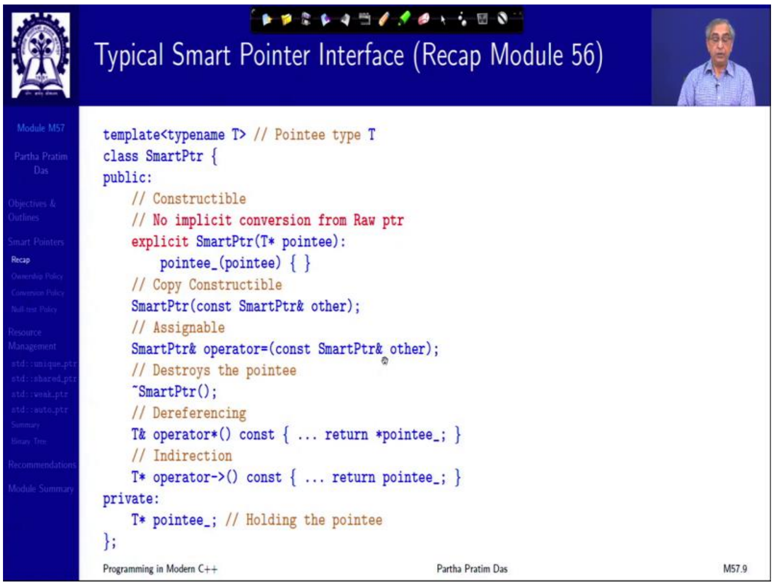
The first smart pointer is a C++ object and it stores pointers to dynamically allocated objects. So, it improves the raw pointers by implementing various strategies in its constructors, destructor, copy and move assignments dereferencing operators and so on, but grossly mimics the raw pointers syntax and semantics.

(Refer Slide Time: 2:13)



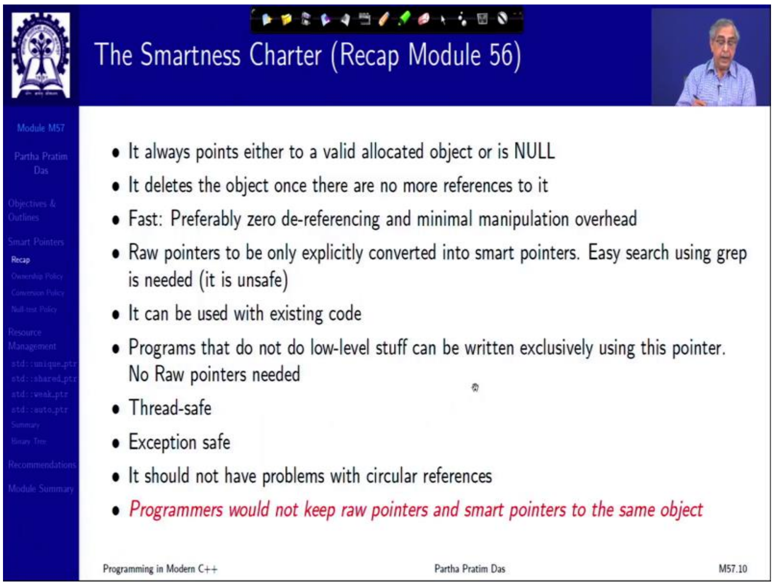
The main highlights are that smart pointers disallow unwanted operations like address arithmetic which is one of the biggest problem area for bugs, it allows the lifetime management by managing the dynamically created objects according to the protocol of the static objects which the smart pointers are helps in concurrency control and supports resource acquisition is initialization and resource release is destruction idioms which help really the resource management.

(Refer Slide Time: 2:53)



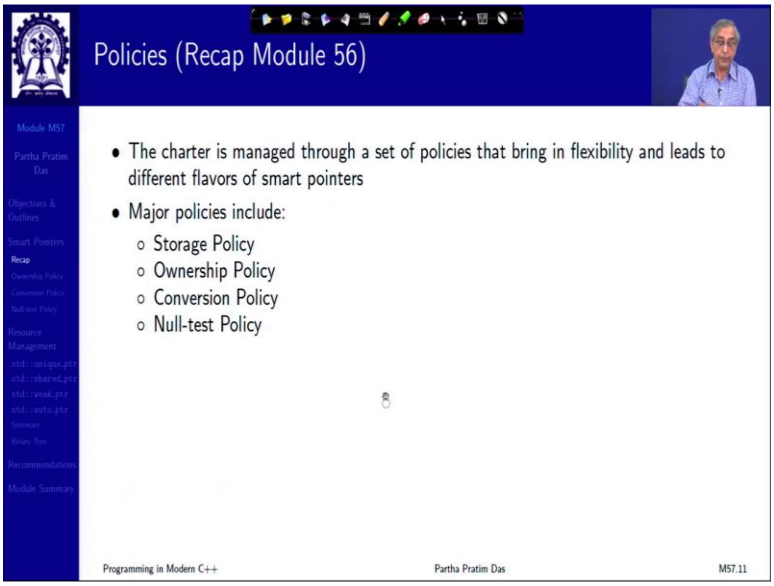
So, this is the interface we had seen the constructor must be explicit, so that you cannot convert raw pointers implicitly, we have copy constructor, copy assignment operator, but most importantly, you have overloaded unary operator *, dereferencing and indirection operators.

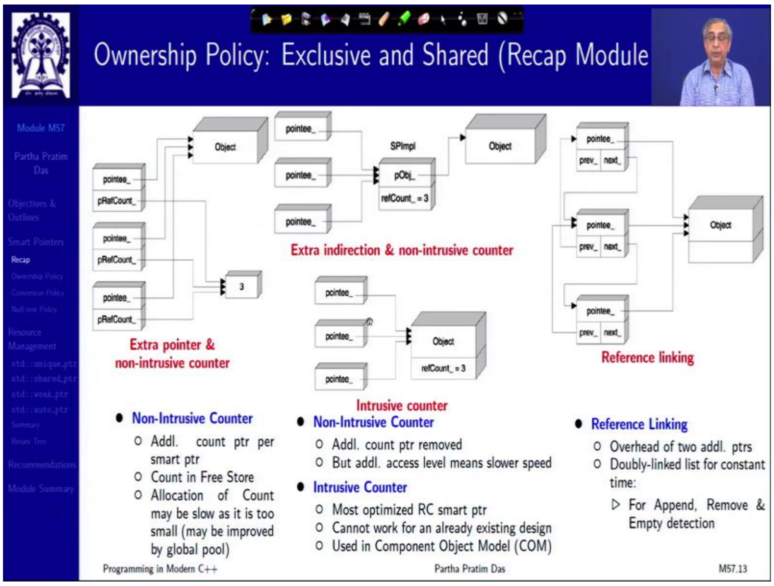
(Refer Slide Time: 3:14)



And we looked at the basic charter of things that the smartness that the smart pointers must have primarily being that it will either point to a valid object or it will be null, it will not be able to point to an invalid object and once it is deleted, the object that is being pointed to or managed by this smart pointer must also be deleted. So, and then there are many others like it must be useful with the existing code, thread safe, exception safe and so on, so forth.

(Refer Slide Time: 3:52)



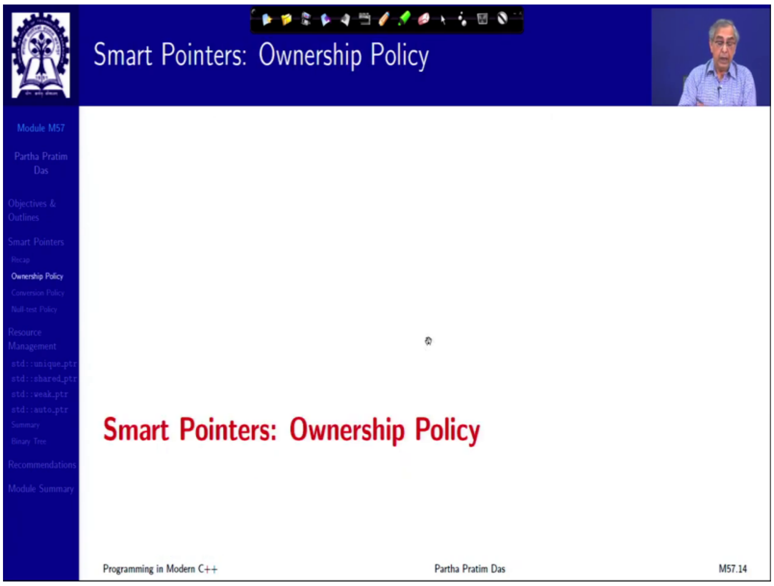


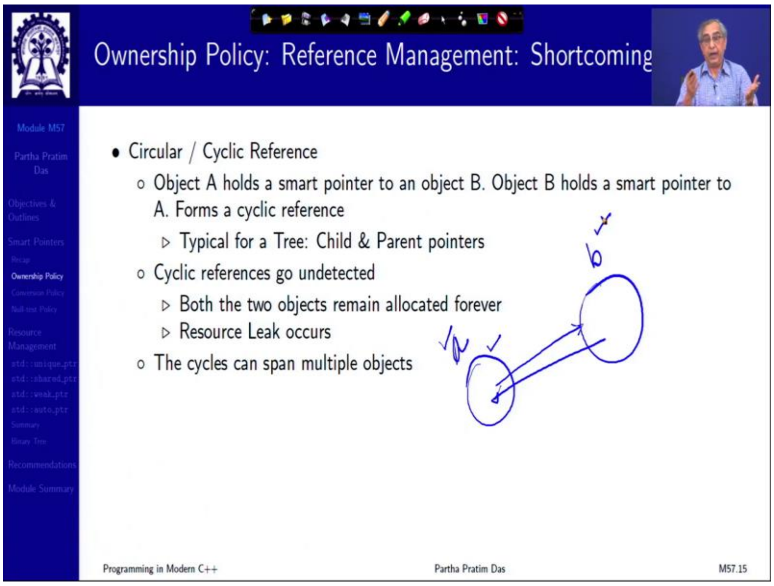
And we have looked at the storage policies and some of the ownership policies, in terms of the ownership policy, we have learned the two basic types of ownership that is, that could be exclusive ownership, wherein a smart pointer owns an object, a managed object in an exclusive manner. So, that if it is copied to another smart pointer, then that particular ownership will get transferred to the new copy and this pointer will become null.

So, this is exclusive ownership and naturally, if you delete and exclusive ownership smart pointer, then the pointee object will also get destroyed. The other is a shared ownership policy where multiple smart pointers can manage or can point to the same object. And so when you copy, a reference count goes up.

When you delete, the reference count goes down and when the reference count becomes 0, then that managed object is deleted. So, in different constructor, copy constructor and operators copy operators are where the strategies are implemented and we saw a variety of different possible strategies for implementing the particularly the shared ownership policy.

(Refer Slide Time: 5:15)



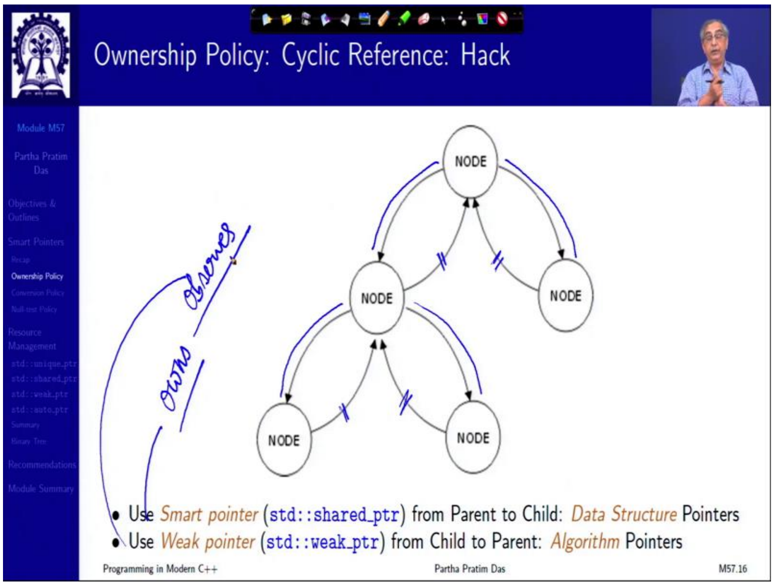


Now, moving on from this on the ownership policy, exclusive ownership is fine. So, when you have only one pointer that can point to an object like say in a single linked list and so on, then you do not have a problem, but when you have shared policy, then you have more than one pointer can point to the same object then there is comes a question of circular reference or cyclic reference, the idea is simple if I have an object which is pointing to another object, now, if this object also points back here.

Now, the question is this has a smart pointer pointing to the object a as a smart pointer pointing to b, object b has a smart pointer pointing to a. Now, if I try to delete object a, then I cannot do that, because I it is pointing to a valid object the reference count is more than one. I cannot delete b either, because it is pointing to a valid object a, the reference count is one.

So, if there is a circularity, then I will not be able to delete any of these objects. So, these objects will remain there forever. And it is so resource leak will happen. So, circularity of references cause a problem in terms of the shared ownership policy.

(Refer Slide Time: 6:53)

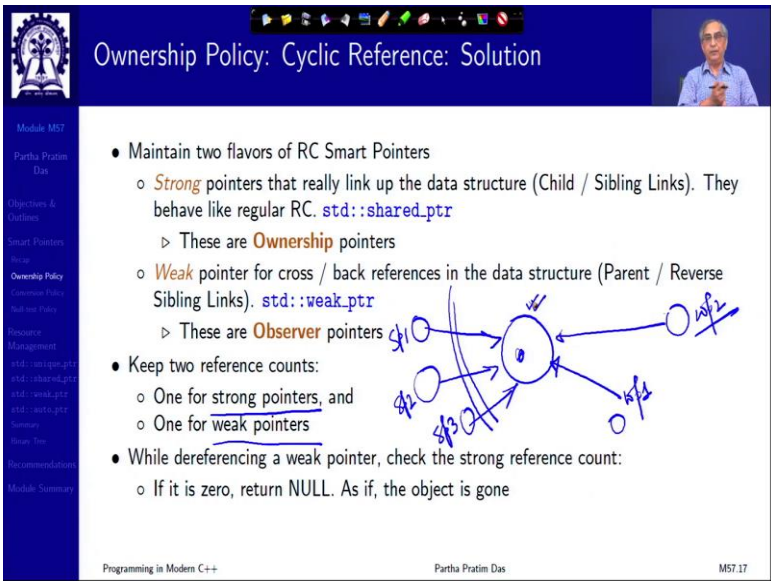


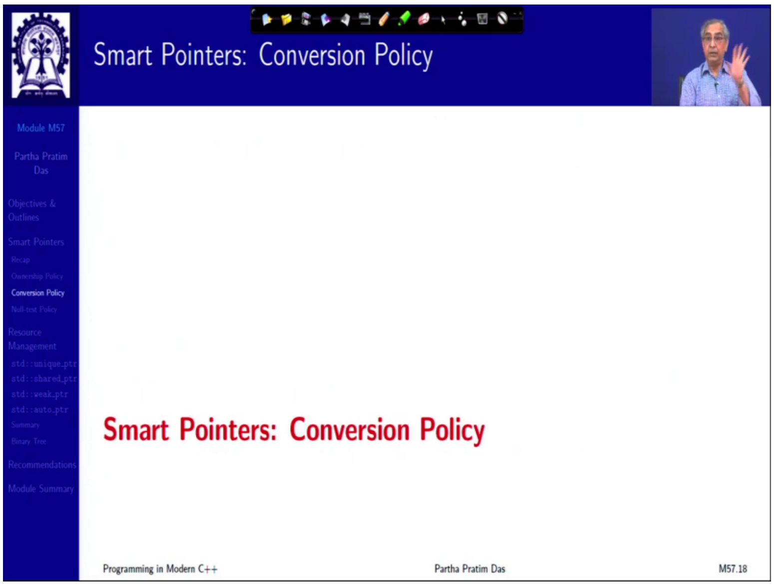
For example, think about this, you will not be able to delete any of these five nodes, because there is always more than one here, which are being pointed to. So, to take care of that, what is done is we introduced two different types of pointers. One is called a shared pointer.

For example, if I can show you with certain colors, let us say this is a shared pointer, this is a shared pointer, this is a shared pointer, and said this is a shared pointer. A shared pointer is basically a pointer of the data structure which holds the structure together. The other ones, we considered to be a different kind of pointer.

We just mix it references back to some node, but really not owning the node. So, smart pointer owns the other type, we call it the weak pointer and smart is not a good word, observes, is some smart pointer holding this object or not. But the fact that a weak pointer is pointing to an object is not a restriction for that object to be deleted, whereas if a smart pointer is pointing to it, then that object cannot be deleted.

(Refer Slide Time: 8:23)





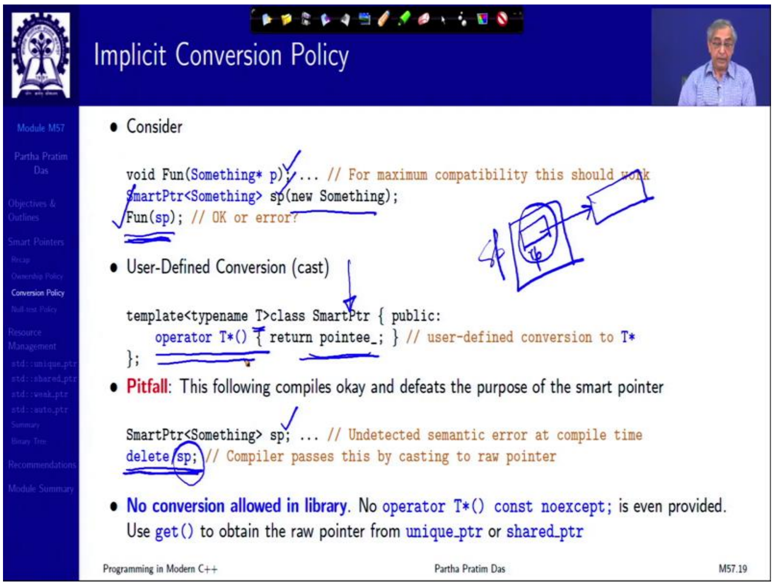
So, this is the simple trick that is introduced. So, often, the smart pointers or the strong points are really the links in the data structure. Whereas the weak pointers are primarily the algorithm pointers, they just observed. They just keep track. Now so how do you keep track of whether there is a smart pointer or there is a whether there is a strong pointer or whether it is a weak pointer and so on.

So, instead of one reference count, which we are having earlier, now, you have two reference counts, a strong pointer count and a weak pointer count. So, strong pointer count tells you that how many pointers are owning this object. So, I can have an object, I can have multiple smart pointers, which are pointing to it sp2 multiple smart pointers are pointing to that.

So, as long as all of these smart pointers exist, at least up to one smart pointer exists, this object cannot be deleted. I also have weak pointers. I also have weak pointers pointing here. The weak pointer does not own it is just keeping track whether there are smart pointers which are pointing here. So, a weak pointer, but it is pointing to the object in a certain way. Now but since it is not owning it, even when a weak pointer is pointing to an object, that object can be deleted.

And because it is pointing to the object, it can serve as the purpose of referring to that object, I can use this pointer and in some way there are restrictions, but in some way I can access that object. This is the basic idea. I am not going into the details of the implementation, but this basic idea, if you keep in mind that will be fine. The next is a, so, we have the storage policy, we have ownership policy.

(Refer Slide Time: 10:39)



The next is what is known as a conversion policy. Conversion policy says something very simple if I have a smart pointer. So, what do I have? I have a smart pointer. So, basically an object which insight, we have a raw pointer, which is actually holding that object. This is the structure. Now, if I have a smart pointer, then can I convert it to its raw pointer, for example, just look at something in some class.

So, I have defined a function Fun which takes a raw pointer to something and have defined a smart pointer sp to this allocation, which is a smart pointer to something. Now, the question is, is this call allowed? If you look at the type, the type of the formal parameter is something * whereas that is pointed to something whereas the type of the Fun is a smart pointer that it is an object where it is a C++ object where the dereferencing and indirection operators have been overloaded.

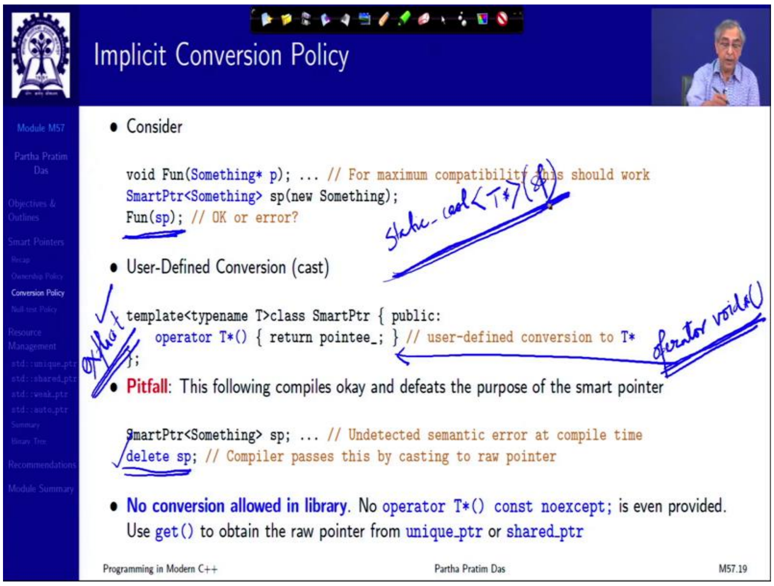
The question is should this be allowed, as such they should not be allowed because they are of different types. So, this can be allowed provided a smart pointer by default can be converted to a raw pointer. So, how will that happen, that means there has to be a conversion. Now, since it is to be converted to a raw pointer, which is built in type, we cannot make use of the constructor of the built in type to do this conversion as we have learned.

So, if this has to be allowed, then the smart pointer class has to provide a conversion operator. So, say it provides an operator T* where T is a, so, if this operator is also provided, then what will happen, if I pass it a pointer of pointer sp of the smart pointer T type, then it will automatically return the internal pointee, return this internal pointee and give me the raw pointer.

So, if I provide this, then this conversion will be allowed. That is good. But there is a big problem. The problem is suppose I have a smart pointer sp. And suppose I have written delete sp, it is a mental over it, because pointers we delete. Now, delete sp should not be allowed, because sp actually is an object, it is not a dynamically allocated pointer on which you can call the delete operator.

So, delete sp should have given me a compilation error, but it will not, it compile fine because the compiler finds that sp is of type smart pointer, which has a conversion to the raw pointer. So, it will do the conversion, get the raw pointer and delete that. Something which is semantically wrong and compiler should have given an error does not give an error.

(Refer Slide Time: 14:13)

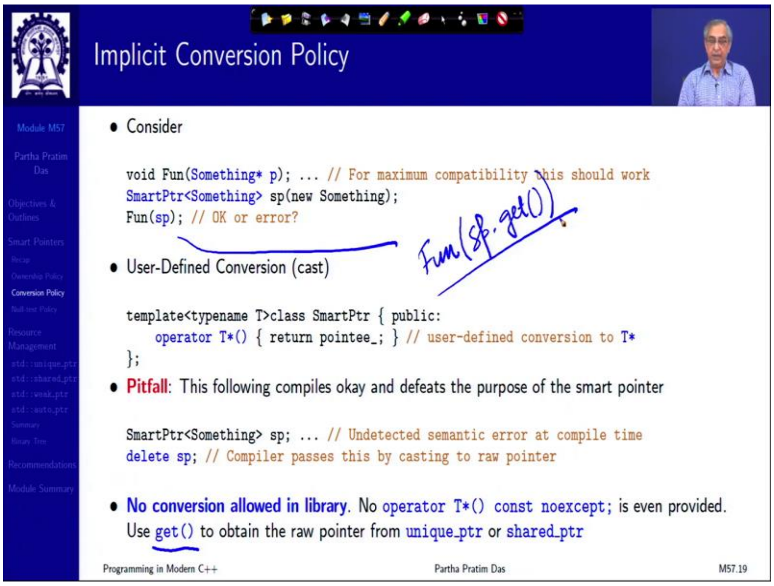


So, the conversion implicit conversion is not a good idea. There are different there could be different ways of providing the conversion and then my blocking it for example, we have learned about explicit keyword for making the conversion operators explicit you could use that, if you use that then again it will not be this will not this problem will go away because this will not compile. So, this will also not compile.

So, it will then have to write static_cast<T*> then or something * then sp because it is explicit you. So, this syntactic beauty will in any case go away or it could if you are working in C++03 where you do not have the explicit keyword The other trick or hack that you can do that you can provide another operator conversion operator void*.

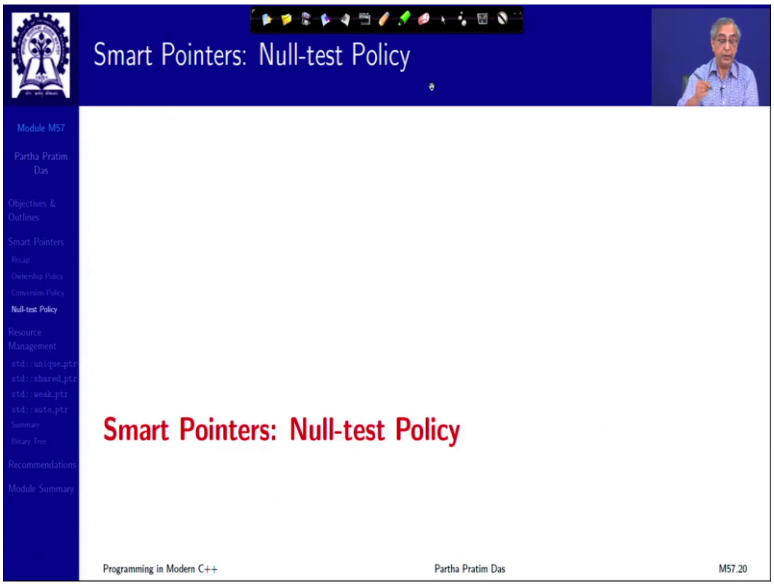
If you do not explicitly is the preferred, if you do not have you can alongside operator T* you can provide operator void star then also what will happen the implicit conversion will not be allowed because these two overloads will not be resolvable so, the compiler will see that I do not know only explicit ones will be. So, whether they do it by this mechanism of C++11 or do it by this hack of C++03, I have to, to avoid such risks, I have to use an explicit conversion in this way.

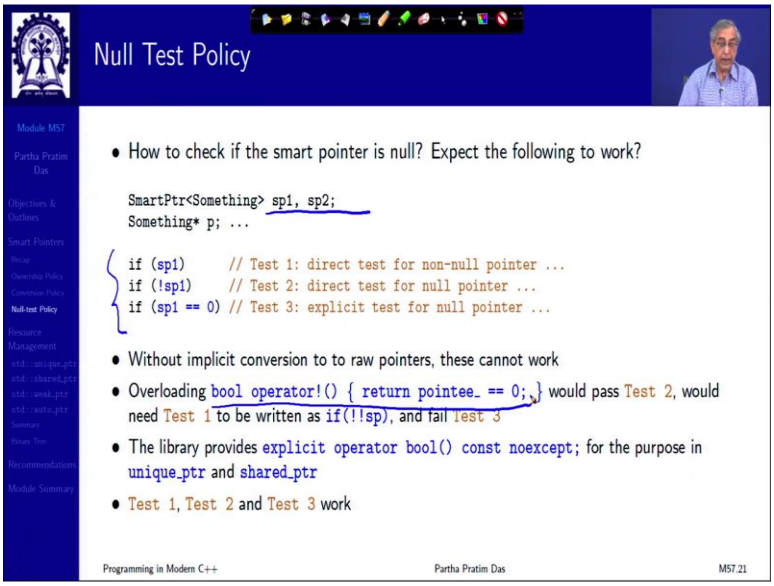
(Refer Slide Time: 16:08)



Now, so, the language committee while discussing deliberated over that and said that if this has to be done, then it is better that the user says that I am doing it. So, it does not provide any conversion operator, smart pointers in the standard library does not have a conversion operator rather it gives you a function member function get() by which you can get this raw pointer simple. So, if you have to write this, you have to write this as Fun(sp.get()) and that will get you the raw pointer, of course, this is much less cumbersome looking and less typing compared to the static cast. So, this is what is available for the unique pointer and the shared pointer.

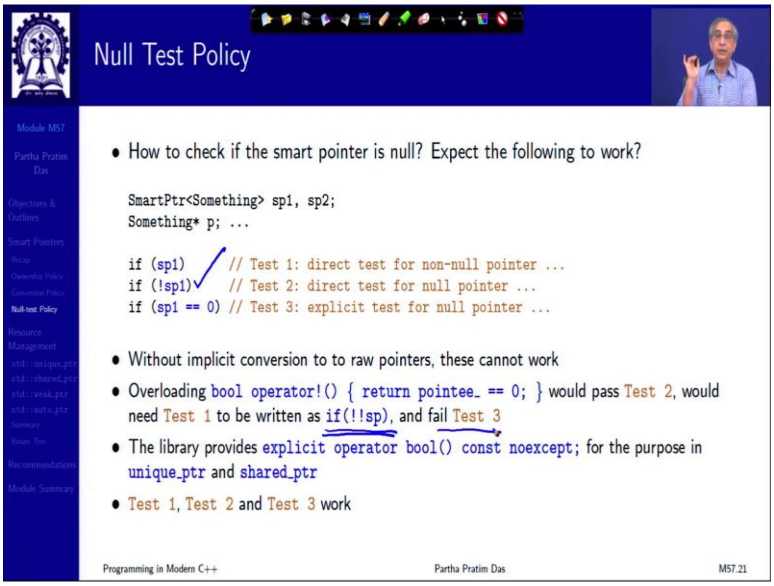
(Refer Slide Time: 16:55)





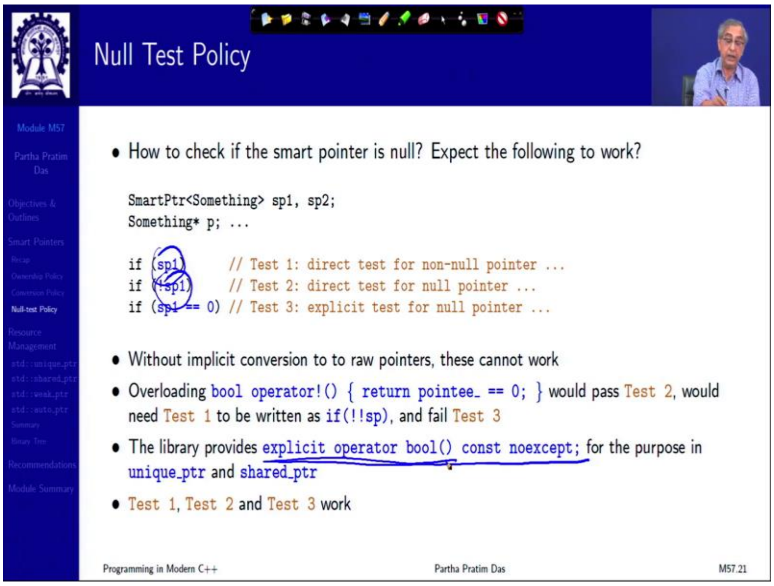
This next question is null test policy we often check pointers for null. So, given a smart pointer would I be able to do these checks this is we check if it is not null, if it is null, if it is equal to explicit check and so on. This is a very common way of checking, naturally since we do not have an implicit conversion this will not be possible, because obviously these are objects so, they do not have a mapable operator. So, one way you can make it work is you can overload the negation operator and say the negation operator checks for equality of the pointed to 0.

(Refer Slide Time: 17:38)



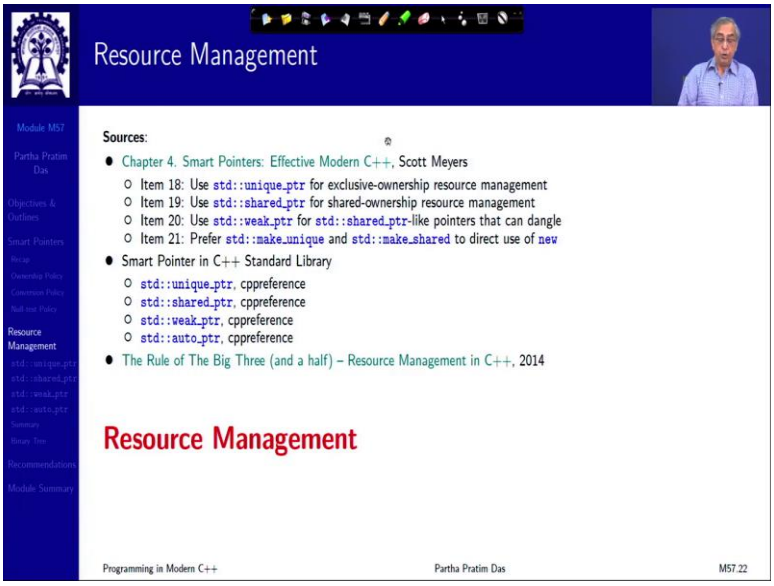
If you do that then naturally this will work is the case the test 2 will work, test 1 will still not work because the negation operator does not get invoked. So, test 1 you will have to write it in a peculiar way like if bang bang sp. So, negation twice it makes it and test 3, you cannot make it pass.

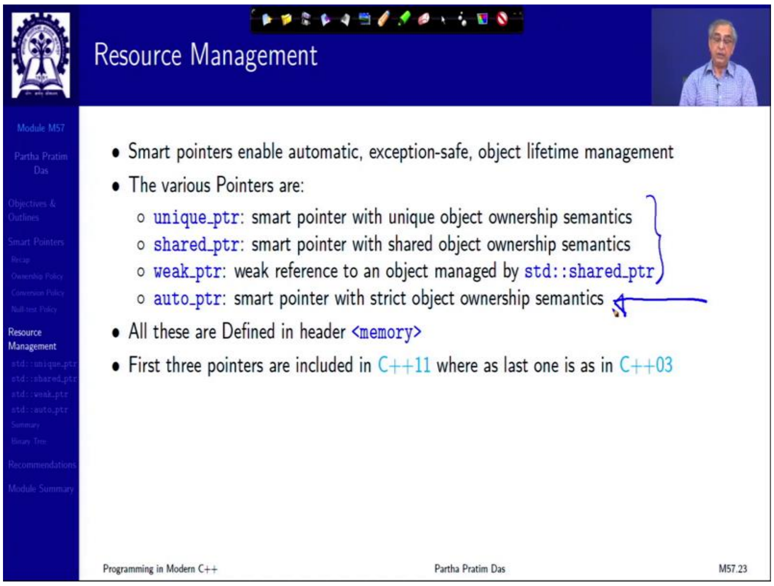
(Refer Slide Time: 18:00)



So, again through consideration what the language committee has provided, instead of doing all this it has provided a explicit conversion to bool which is a very specific one, which is explicit operated bool and that if you have an explicit conversion to bool then in the context of such tests, the explicitness is not considered this particularly for operated bool, if you do not remember please go back and see the discussion on the explicit. So, explicit operator bool is provided in unique pointer and shared pointer, so, that all of these three tests will work. So, that is the null test policy.

(Refer Slide Time: 18:50)

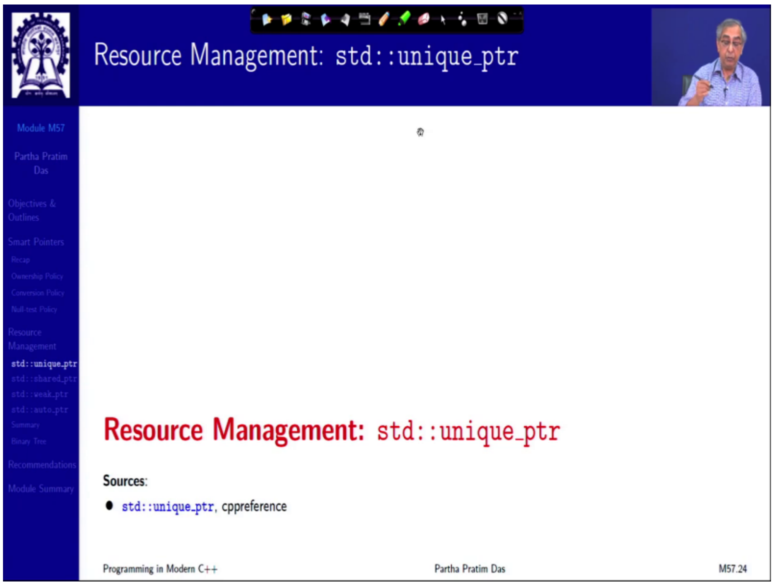


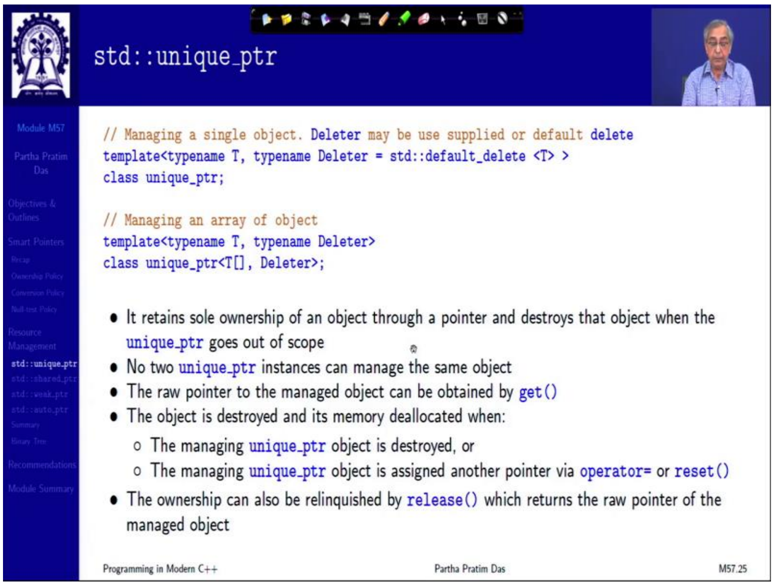


So, having said that, now, let us quickly take a look at what are the different smart pointers we have, there are four kinds unique pointer which is exclusive ownership, destructive copy, shared pointer, which shares the ownership, weak pointer, which refers to an object managed by some other shared pointer, so that the cyclic reference can be avoided.

So, these are the three smart pointers which are critical for the study, you do have a pointer, smart pointer called auto pointer, which exists in C++03. But, so, we will we have included it but it is deprecated in C++11 and in C++17 this has been completely removed. So, unless you are restricted to use C++03 only do not use the auto_ptr. All of these are available in the memory component of the library.

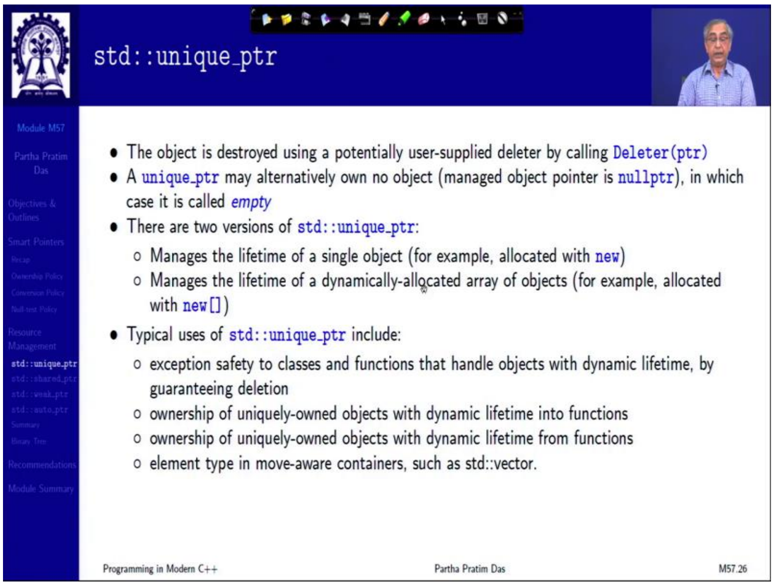
(Refer Slide Time: 19:48)





So, what is a unique pointer, unique pointer is simply that at any point of time it holds a single object, if you copy, the ownership transfers as simple as that. You have a get() to get the raw pointer, if you destroy the unique pointer, then that object being managed also get destroyed if you assign it to other unique pointer, the object gets transferred, if you reset the unique pointer, then also the object is destroyed. So, these are the different features the unique pointer has.

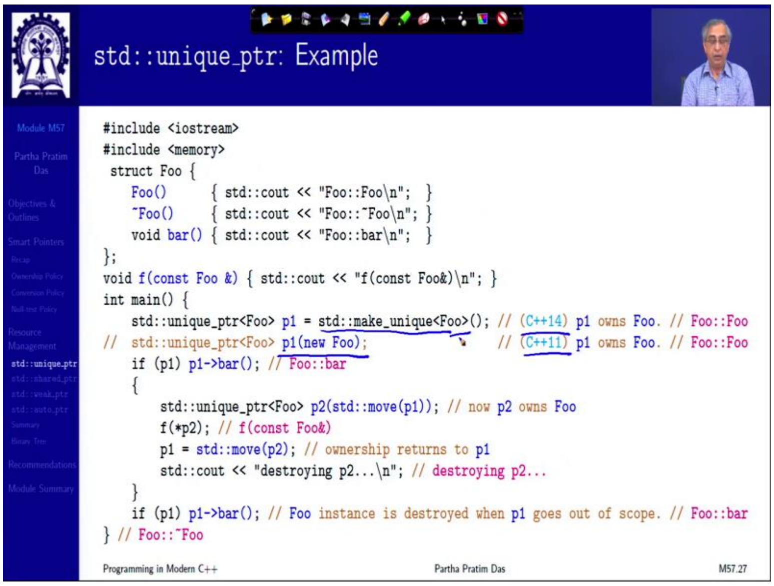
(Refer Slide Time: 20:23)



One special thing about unique pointer is unique pointer can also be created to an array of dynamically allocated array of objects. So, it can be to a single object or a dynamically created array of objects, it has a complete exception safety and all the kinds of assignment and copy problem, exception path problems we had talked off, they will get solved by this use of unique pointer, it is used to pass parameters to functions to get values returned from function.

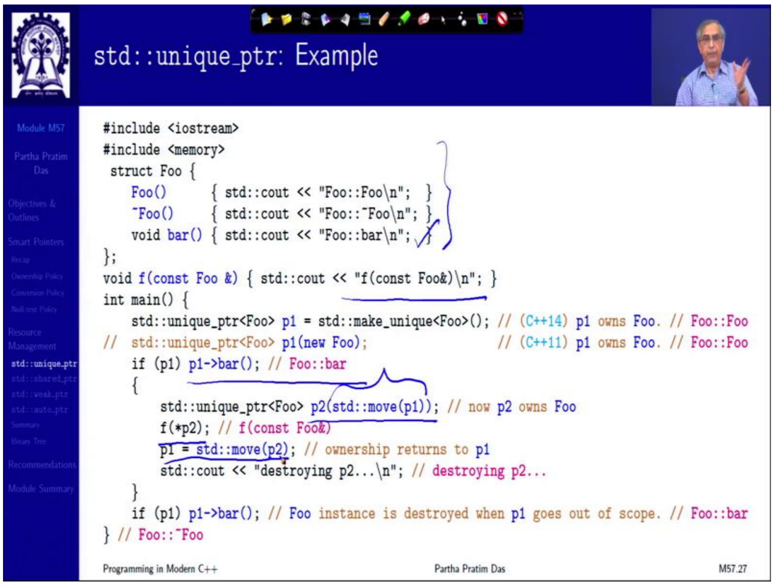
And several containers use unique pointer like in like I have a vector. So, in a vector, as you have, we have discussed that it is an kind of array of pointers where the, but every element is held through a pointer. So, those are all can be unique pointers, because obviously every pointer will hold only unique element that exists there.

(Refer Slide Time: 21:26)



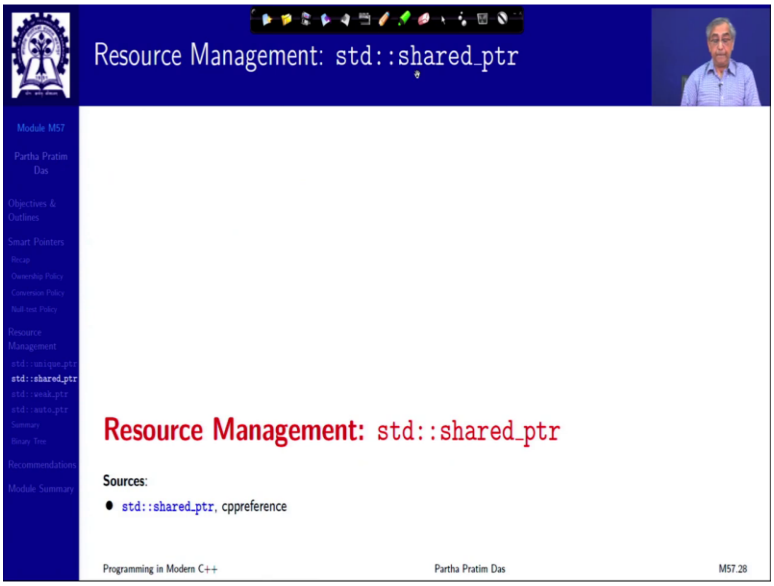
So, most useful, kind of pointed, here I have given a sample program, which you can execute and see that the basic behavior of ownership transfer is happening properly. So, for example, you create a unique pointer, you instead of writing this, I am sorry, instead of writing this you can also write this new Foo and use that to initialize a unique pointer. That is a C++11 style of initializing the unique pointer. C++14 gives you something nice it gives you a STL function make_unique and you can create unique pointer through that and you should actually always use that.

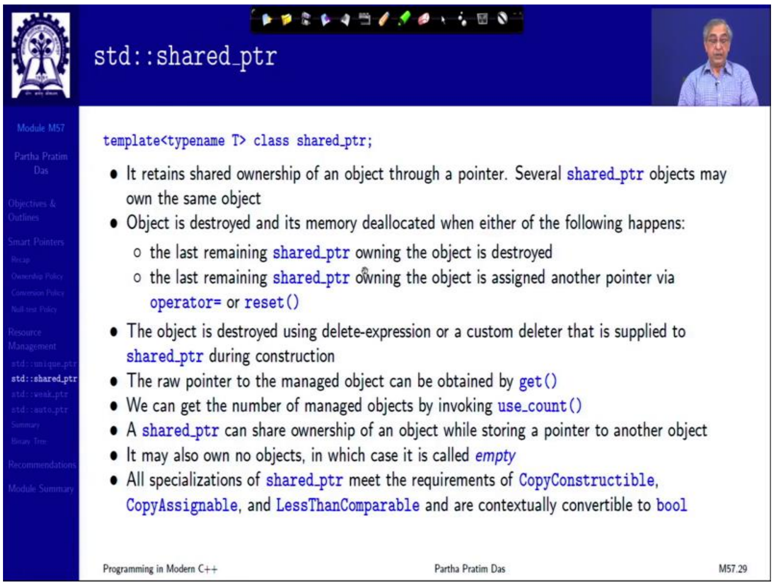
(Refer Slide Time: 22:15)



So, p1 is a unique pointer to a to an instance of Foo. So, if you do Foo pointer bar, then it will execute this bar function. If you move this unique pointer p1 to another unique pointer p2 can see a steady move for the purpose of making sure that you take the move semantics, if you do that move, then certainly the ownership will go to p2. So, now if you do start of this in f, you will get this because it now owns that Foo object, if you make an assignment, the ownership will come back and so on. So, you can create two lessons.

(Refer Slide Time: 23:07)



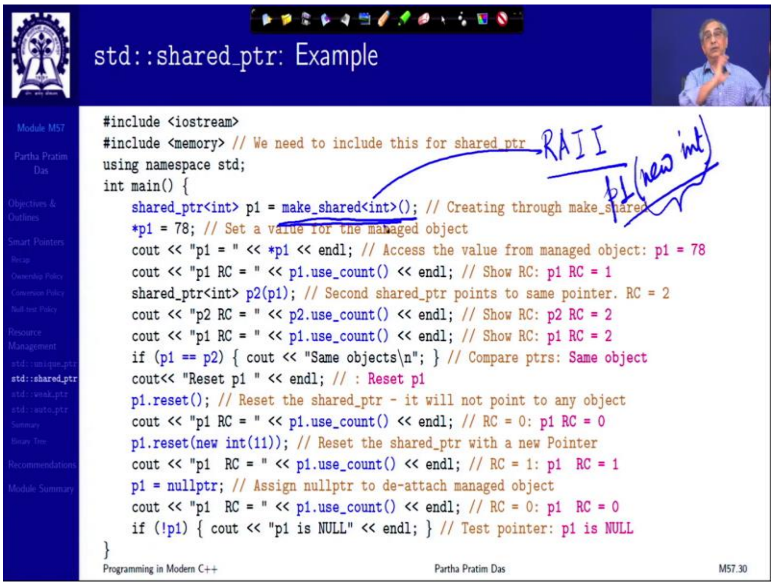


Next is the shared pointer. That is the most interesting, the shared pointer is a pointer which can where multiple pointers can point to the same object in a shared collaborative manner. And the object managed object gets destroyed only when the last shared pointer pointing to that is being destroyed.

Others as long as there are shared pointers remaining who are managing that object, other shared pointers can be destroyed without destroying this object. Of course, you can if you assign also the ownership gets transferred, ownership get copied basically not transferred here, ownership get copied. So, the reference count will increase if you reset then it will get released.

You can get the, we mentioned it you can get the raw pointers using get. And at any point of time, you can do a use_count() to know how many smart pointers are pointing to this object. That is you can get the value of the reference count. So, that is a very nice design.

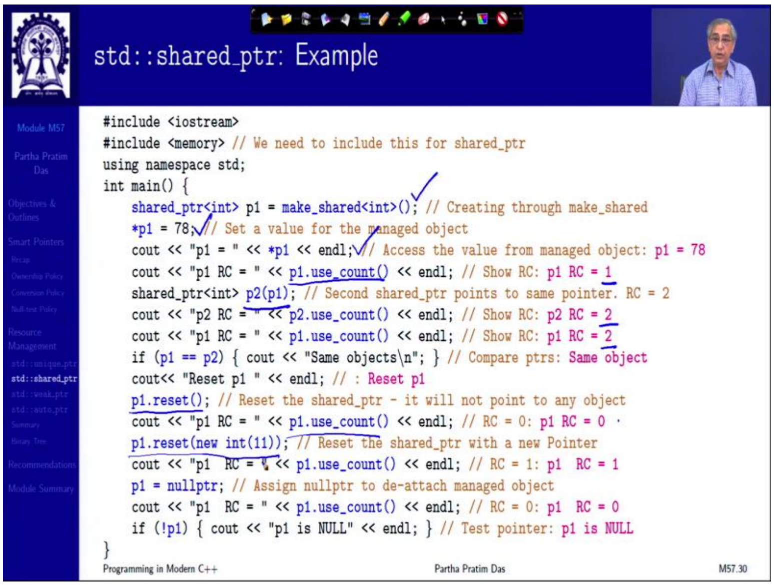
(Refer Slide Time: 24:24)



Again, another sample program which shows you various different ways of dealing with the shared pointer, shared smart pointer. Like unique here you have other STL function makes_shared to construct a shared smart pointer. Now these whether you are doing make_unique or you are doing make_shared, this basically follow this RAII because you can see that you have not even done a new, you did not even required to do new that is done from inside this.

So, when you do not even get to see the raw pointer that got created in the dynamic allocation, so, the difference you could have done, for example, here you could have written like this p1 new and int, you could have written this. If you are doing this then new int, which is an rvalue a temporary object is at least you can see that that is identity of the raw pointer. And then that raw pointer is being owned by p1, which is the shared smart pointer. But if you do make_shared, then you do not even get to see that so resources acquired and immediately initialized into the shared smart pointer.

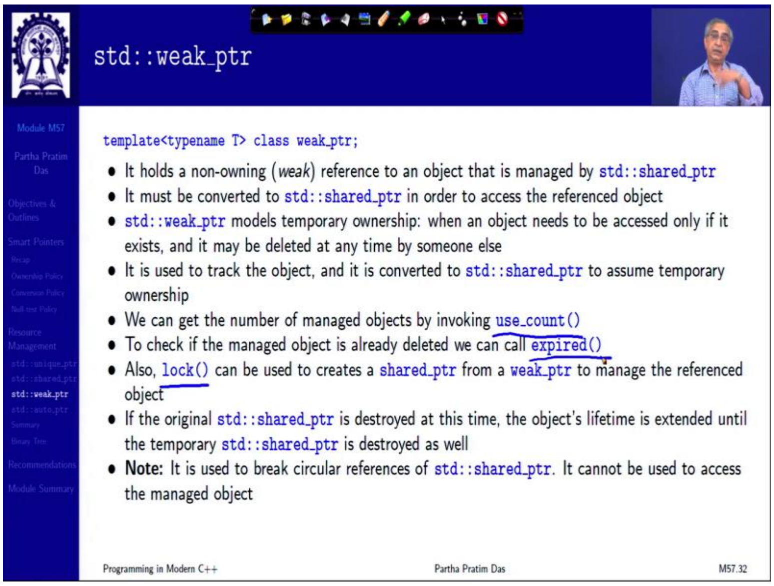
(Refer Slide Time: 25:54)



So, you can have this, like you do in a typical raw pointed way, you can set a value, you can print that value, you can see how many are pointing to this, right now it is one, you do a copy construction of the shared pointer, the count will go up to two, both of them both in p1 as well as in p2, now two pointers are pointing.

So, if you take a look share count for p1, as well as share count for p2, both will look to be two because the count is of the total number of shared pointers that are pointing to this object. And so on so forth. If you do reset, then the object will go, use count will fall back to zero. You can reset and in that process set a new object. Not a very good idea, though, and so on, so forth. So, that is a shared pointer use.

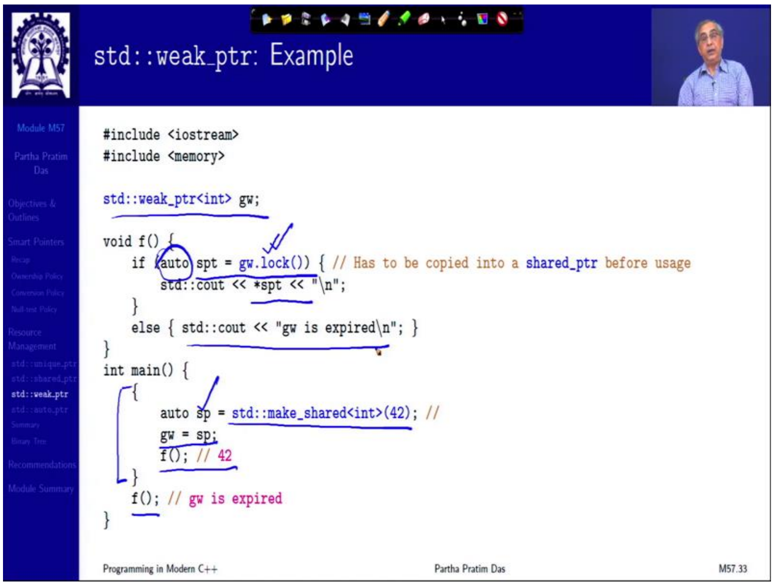
(Refer Slide Time: 26:56)



Coming to the weak pointer, weak pointers are you cannot weak pointers do not exist by themselves. Weak pointers can be it is a non owning reference, it is an observable vision reference. So, they can observe objects managed by other shared pointers. So, the weak pointer directly cannot access the object, you can take a weak pointer, and from that you can convert to a shared pointer and access the object.

And for that a particular function is given it is called lock(). If you do lock on a weak pointer, it gives you a shared pointer and you can use that shared pointer. Like in shared pointer, weak pointer also will give you the count, use_count() which is the number of weak pointers pointing to this object. If no weak pointer is the, sorry, if the object is no more existing, then the weak pointer which we can call the expired function and see that if there is the object is existing or the object has already expired. So, these are the different operations you can do.

(Refer Slide Time: 28:14)

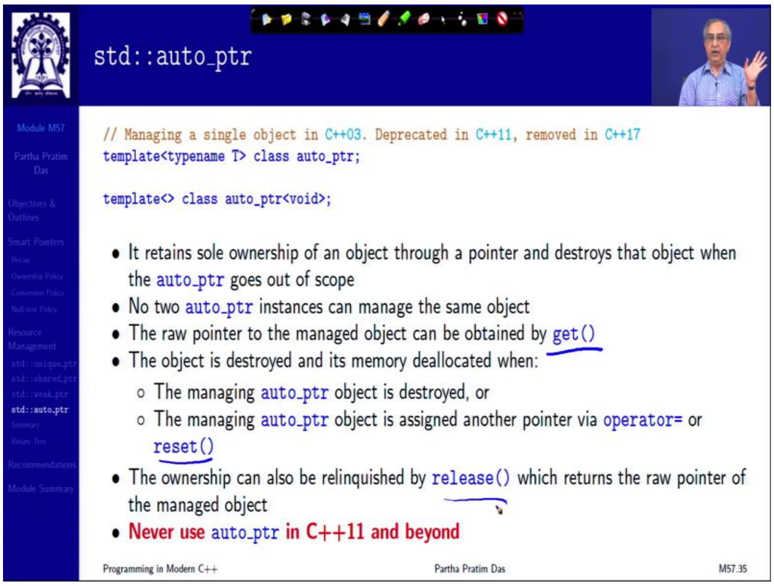


Here is a simple example. So, we have a shared, we have a weak pointer and we have done a lock. So, from that, by that lock, we have got a shared pointer by doing a lock the type is set by auto and we can make use of that because with the weak pointer, we cannot directly access object.

Then we have created something from make auto, make shared and then we have assigned that to a weak pointer. So, weak pointer also observes it and if you print f, it will be able to print f but once you come out of the scope, naturally, this will get deleted because in an automatic scope.

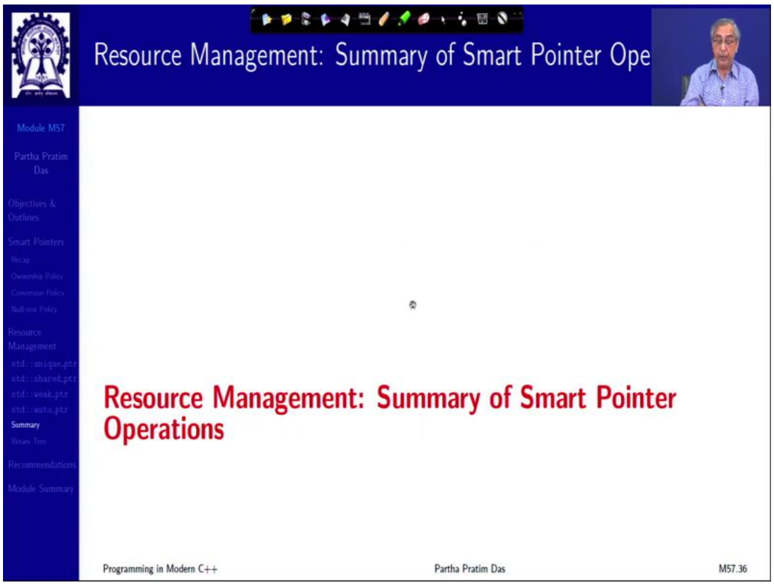
So, once this shared pointer gets deleted, obviously the managed object will also get deleted. So, the weak pointer will find that it has actually expired. So, now when you call f, this lock will fail and you will print that the weak pointer is expired. So, that is the basic logic.

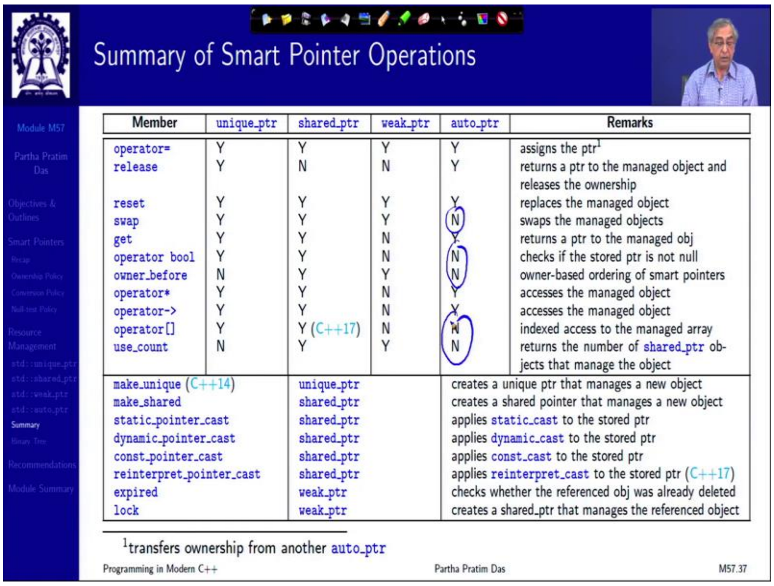
(Refer Slide Time: 29:36)



You have an auto pointer, this is historical legacy of C++03 which is somewhat like the unique pointer but it does not have most of the facilities of the, but it could just be whole unique objects and the ownership gets transferred by assignment and you could extract the raw pointer by doing get, do reset, release and so on. And but most of the other operations like checking for null test and all that are not available. So, it is not something which is advice to be used at all in C++ onwards.

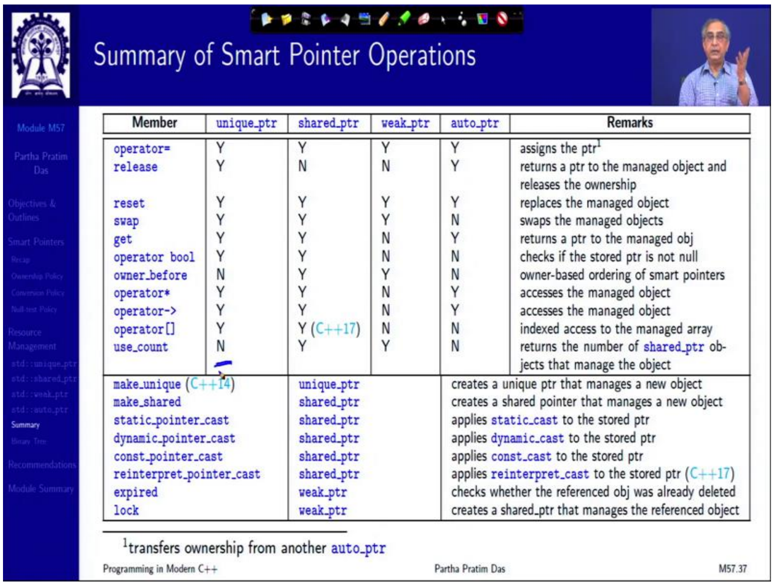
(Refer Slide Time: 30:26)





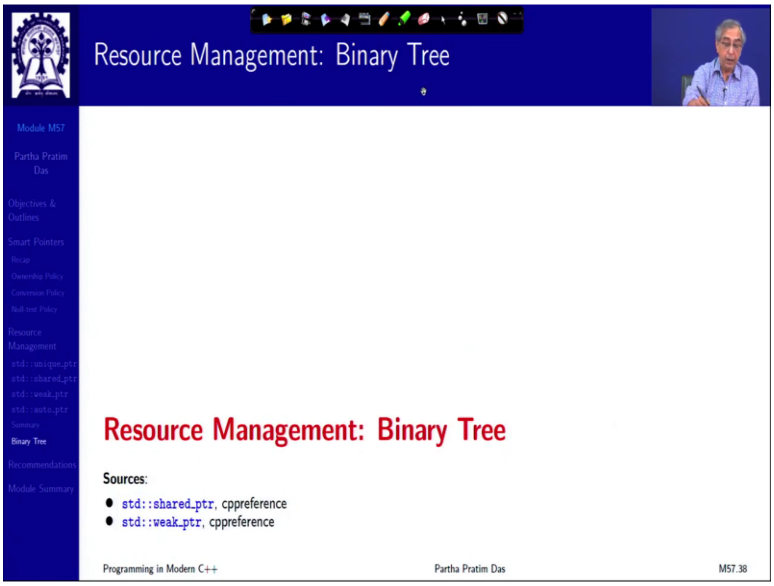
Now, here as I usually do I have given summary one single slide chart of what are the different I mean, these are not an exhaustive one, but these are the major member functions that you have and which member function is available in which kind of smart pointer and what does it mean. So, you can study this and you should be able to understand why you have so, for example, you can see that the reason you do not use auto is most of these members are not available in auto.

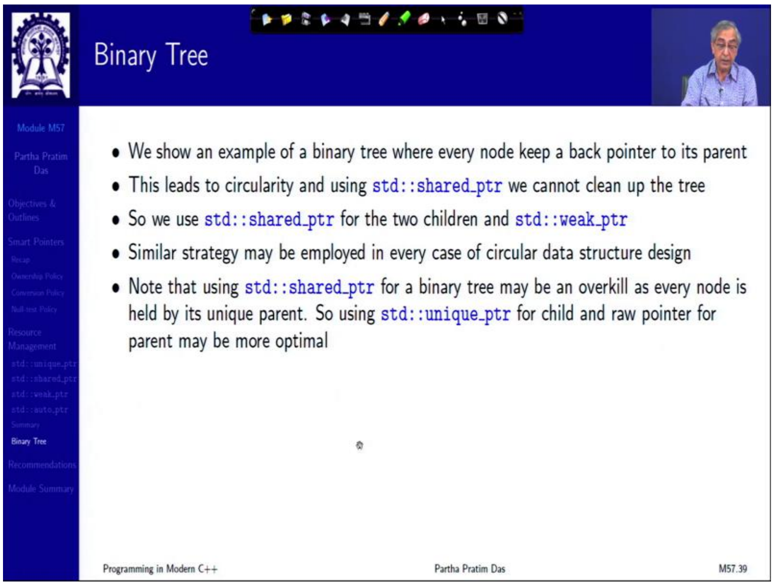
(Refer Slide Time: 31:03)



Whereas, some are available in some, for example, use_count() is not available in unique_ptr because in unique_ptr use count is always one, that is not it is an exclusive ownership, so it is not much meaning. So, this is a quick reference that you can have.

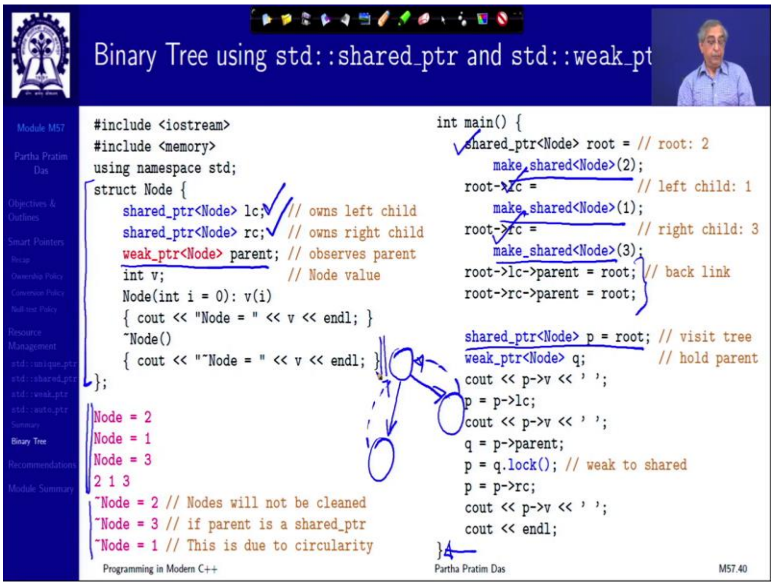
(Refer Slide Time: 31:20)





So, finally, we end with an example to see we had talked about having shared pointer and weak pointer to break the circularity. So, let us see did we have we been able to do that.

(Refer Slide Time: 31:35)

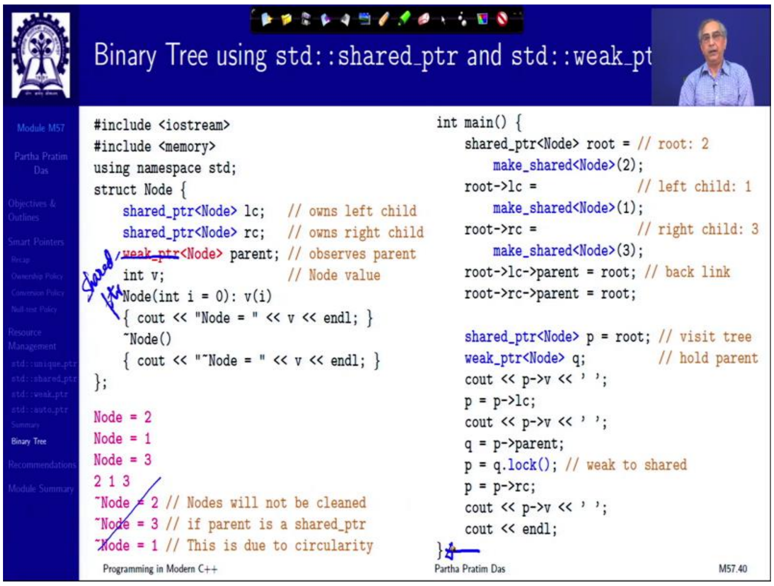


So, I have given here a very simple node design for a binary tree. So, you have a node, you have two child pointers to others, and each child pointer has a parent pointer, the child pointers are smart pointers, left child and right child whereas the parent pointer is a weak pointer. So, that is how you break that cycle.

And then you can make a root, set the child node of the left child of the root, right child of the root, set the parents of both the children and you can just create another shared pointer to the root and using that shared pointer, you can check the values this is all that gets displayed.

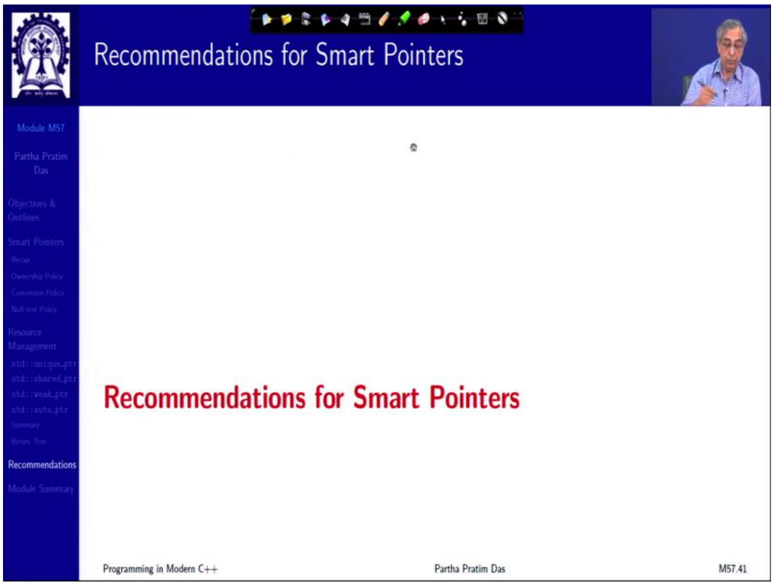
And finally, you are not doing any delete or anything, but as the program ends at this point, naturally, these shared pointers that have been created goes out of scope. And therefore they are automatically deleted and the corresponding nodes are deleted and therefore, the destructor prints this messages.

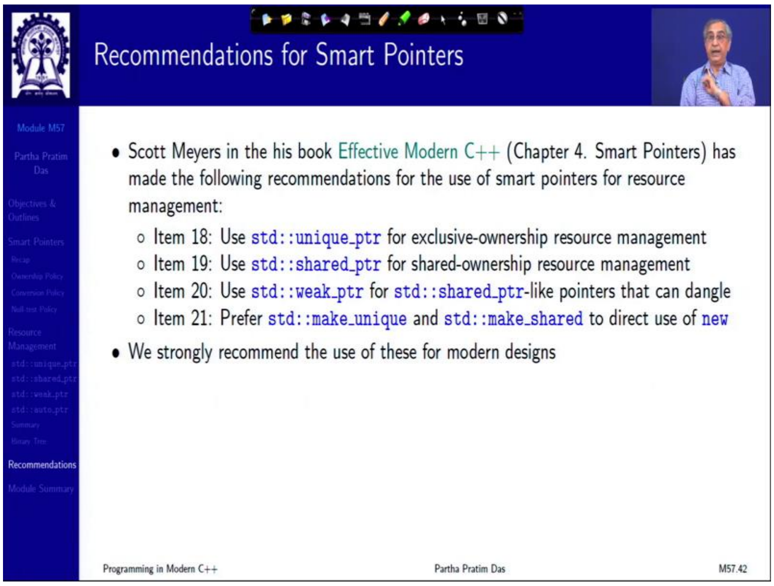
(Refer Slide Time: 33:02)



Just as a final check, just as a final check that this indeed is a solution you need it change this week pointer to shared_ptr. If you do that, the circularity will come back, try that and we will see that these messages will not come because now at this point, even though these three objects are going out of scope each one of them has a reference count, which is not 0. So, it cannot be destroyed. So, that is the that is the basic story.

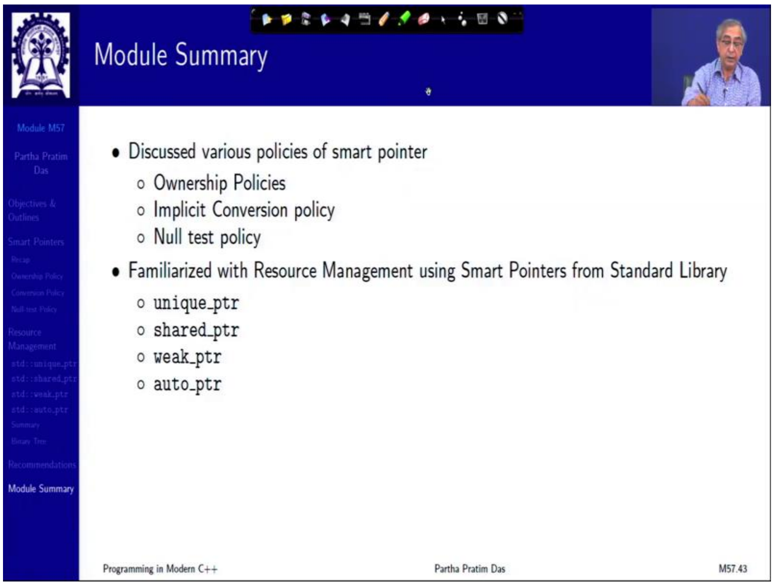
(Refer Slide Time: 33:43)





Finally, I would recommend that you study the chapter 4 of effective modern C++ the book by Scott Meyers, which is an excellent discussion on smart pointers particularly in modern C++ and there are four items which make four recommendations and that you must always follow. Use unique pointer for exclusive ownership use shared pointer for sharedly own resource management use weak pointer when shared pointers can dangle and make use of make_unique and make_shared do not use new along with these pointers.

(Refer Slide Time: 34:23)



Thank you very much. We have discussed about the different policies and introduced the smart pointers in C++ Standard Library. Thank you very much for your attention and we will meet in the next module.