Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 56
C++11 and beyond: Resource
Management by Smart pointers: Part 1

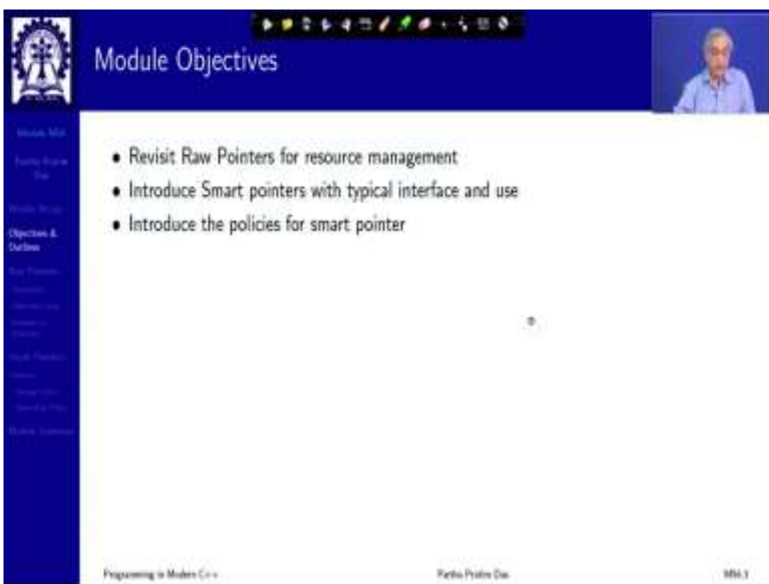
Welcome to Programming in Modern C++. We are in week 12. And I am going to discuss module 56.

(Refer Slide Time: 0:35)

Earlier in the previous week, we have learned about various important C++ features. Continuing on Rvalue reference, we have seen how it works as a universal reference and template type deduction, we have learned more about how move work as an optimization of

copy, we have learned about Lambda expressions, that is undimmed functions, recursive lambda and so on. And various features of C++11 and some of C++14 which are around classes or non class types, variadic templates and so on, kind of you are Now, rich in C++11 familiarity.

(Refer Slide Time: 1:30)



In this context, in this module, we are trying to discuss about resource management using what is known as smart pointers. We will revisit raw pointers for resource management and introduce smart pointers interface and so on. This is not a language extension resource smart pointers existed in C++03 also, it is more like a style of doing things and it is a standard library support that we are going to discuss.

(Refer Slide Time: 2:05)

So, this is a outline. So, first, quickly about the raw pointers, now, the question is what is our motivation for doing this discussion. Now, naturally, we always want to get a get into the culture of writing good C++ code, which has to be correct, bug free, maintainable, high performance and so on. And that is the basic object. Now, what creates a major problem in this whole endeavor, one of the factors is dynamic creation and destruction of objects. On one side, we know, that this is a major strength of C++, but it is also a bugbear of these languages.

Because it needs that this resource management the dynamic other than the dynamic resource management, everything is automatic, the compiler does it either it is explicitly automatic or it is in the static area. So, when which has an automatic like behavior, like you do not need to make a decision about when to allocate memory, create an object and initialize and when it is

safe to clean up that object and return that memory to the free store. So, because of this requirement, which comes from the enormous power of the dynamic object management, often we the programmers failed to do things correctly properly and that leads to the tag that C++ is unsafe, it is a memory leaking language and so on.

So, resource management talks about freeing the client from this lifetime management of the objects, it can eliminate memory leaks and other problems. And when we talk about resource it does not necessarily have to be only address locations. It may be memory, it may be file, it may be socket, it may be mutex, it may be a database connection and so on, so forth. The basic idea is put an effective resource management in place so that the dynamically managed objects can be managed as automatic objects. That is on one side we want the advantages of dynamic object creation destruction.

On the other side, we want to avoid the problems by giving them a lifetime management which is like the automatic objects. So, this is the basic idea of the resource management.

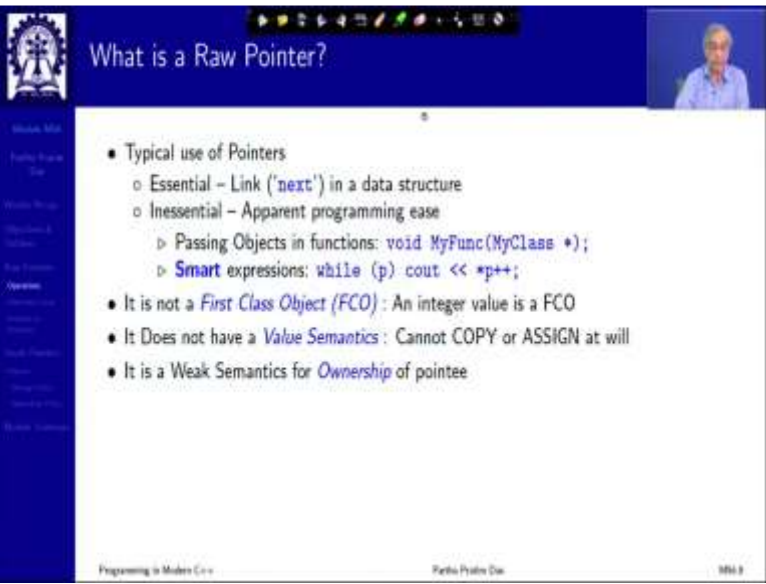
(Refer Slide Time: 4:57)

Now, the resources as we know, are held by pointers. And in this discussion, we will call the pointers as raw pointers to distinguish from the version of pointer that we are going to introduce, which is smart pointer. So, just to I mean you know, all this, but just to remind you that what are the operations that a pointer can do? It is a dynamic allocation results in a pointer or the address ampersand operator will give you a pointer.

Deallocation is called on a pointer, de-referencing is done by this pointer star indirection by pointer arrow, I can assign pointers, I can do null check on pointers, I can compare pointers. I can cast pointers whether or not it is a good idea, that is a different thing, but I can address of operator, I can do arithmetic, the meaning of that arithmetic is little different, as you know, but I can add an integer or subtract an integer from a pointer, I can indexes and an array.

So, there are so many things that I can do with pointers, of course, the code being these two. Now, some of these are really useful, whereas some of these are really a pain. For example, the address arithmetic that we are allowed to do often is a pain in terms of correctness, unless we are working with a really really low level code, where we have to keep track of addresses.

(Refer Slide Time: 6:31)



So, the typical use is in creating links or passing parameters to functions and so on. Now, the main problem is that I mentioned this earlier also that a pointer a raw pointer is not a first class object, like an integer, that is it does not have a value semantics, I cannot freely copy and assign it. If I copy the pointer, the pointee does not necessarily get copied. If I assign something it does not happen in that way. So, there is a very weak semantics for the ownership of the pointee who owns that the pointed object, that's a very ill defined area.

(Refer Slide Time: 7:16)

So, these are the typical ownership issues that we know, for example, this is where the pointer to MyClass is assigned a new MyClass object is created and the ownership is given to p. So, p now, also you need to know, p to get to this new object. But if I assign p to 0 or null, then simply information is ownership information is lost. And we say that memory has leaked, because there is no way to get to this object anymore, that dynamically created object it will still know, will be held in the memory, but I will not know. So, that is a basic ownership assignment problem.

Then I have a copy problem, I similarly create one put it to p, I take another pointer. So, I have this p, I have created one object MyClass object. And I have made another pointer q copying p. So, p also q also points here. Now, I delete q, if I delete q, then I also delete, it means that I actually delete that allocated object. So, unknowingly when I do delete p at a

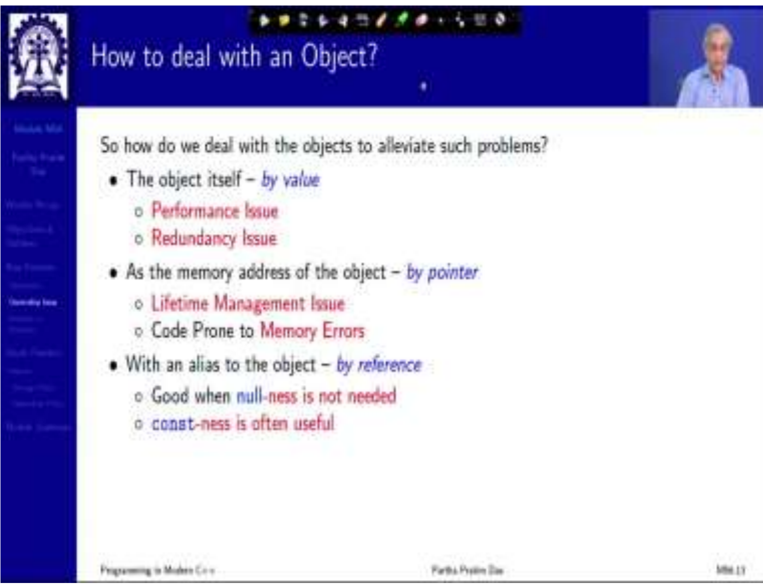
later point of time, the object is already not there. So, I get into a double deletion mean error. And it gets more complicated when I have them in the try catch kind of environment.

(Refer Slide Time: 8:53)

Let me show you a fuller example. I have two pointers p and q. A MyClass object with that pointer, I am calling the function, a function of the MyClass and other MyClass object, I am calling the function of the MyClass. Now, if this function of MyClass can throw, then it can throw here, throw an exception here, it can throw an exception here, in either of that, the try has to break and get me to the catch clouse. Now, at this point, I have to release the resources that have been allocated within the try block. Otherwise, as I go out, these resources cannot be released further.

So, I have to do the, I have to delete them. But if they do not throw then these do not happen. So, I have to delete them anyway. So, the code the same code gets repeated in multiple contexts. So, that is the basic problem that is going to happen in case of using the raw pointers. It does not happen with the automatic variables, because if you have defined anything within the tribe as a as an automatic variable as you go out of that it will automatically get released.

(Refer Slide Time: 10:36)



So, as we know, that there are three ways to deal with an object one is by value, one is by pointer and one is by reference, if you do it by value, you have performance issue, redundancy issue, if you do it by pointer, you are getting these kinds of lifetime management issues, memory issues, if you try to do it by reference, then you can do it in only some limited context when null-ness is not needed or const-ness is very useful.

(Refer Slide Time: 11:08)

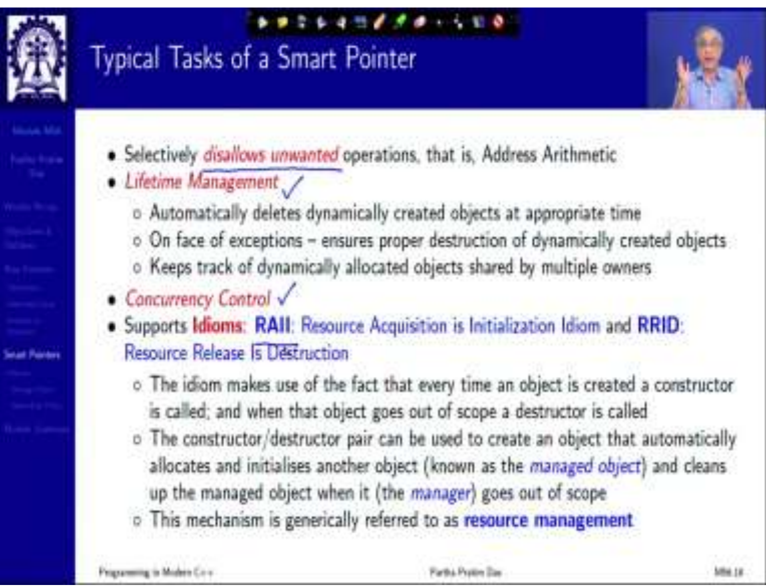
So, if we compare reference and pointer from that perspective, then we can use reference when null reference is not really required, but in many places someone you cannot use reference to build a link. And once created, it does not need to change that is the reason you cannot build a linked data structure with reference it has to be a pointer.

(Refer Slide Time: 11:32)

So, these are some of the problems with the pointers that we have. So, smart pointers are conceptualized to solve this kind of problems, alleviate this kind of problem. So, what is it a smart pointer, is a C++ object. Smart pointers stores pointers to dynamically allocated objects, so, smart pointer has a raw pointer inside which holds that object but it improves the raw pointer by implementing proper strategies in its construction, destruction, copy and assignment, dereferencing and so on, if it does move, then in the move operators and so on.

But grossly it must mimic the raw pointer syntax and semantics. That is the basic idea. We will see that earlier existing code also has to work if I change the raw pointer to smart pointer, but these will have to happen.

(Refer Slide Time: 12:36)



So, what are the common tasks? The first thing the smart pointers do, they disallow what is called the unwanted operations that is address arithmetic. I mean, unless you are a real, you are doing a real system level programming, you do not need address arithmetic. It is a very bad idea to use. So, smart pointers do not have address arithmetic operators.

But it helps him lifetime management, it automatically deletes the dynamically created object at an appropriate time on the face of exception it can create the actual destruction. It is useful in concurrency control and it supports what is called the basic resource management idioms RAII Resource Acquisition is Initialization Idiom, and RRID that is Resource Release Is Destruction idiom.

So, what is RAII mean? RAII means that as soon as you acquire a resource, you initialize you do not let it move around. And you do that. So, in the smart pointer, I said there is a raw pointer pointing. So, as soon as you create the smart pointer at that point itself, you must acquire the raw pointer that is pointing to something and that object which it is pointing to is called the managed objects.

It does RRID in the sense that if you now, release the smart pointers, it must release the managed object as well. So, the smart pointer is the manager and the object that is dynamically actually allocated is a managed object. They must work in complete sync with RAII and RRID. And this is referred to as Resource Management.

(Refer Slide Time: 15:00)

Now, this is a typical interface of a smart pointer. So, what do you what all things you need a constructor and we are saying we will do RAII. So, the smart the constructor must not be implicitly invoke able it must be explicit, I must know, that I am creating a smart pointer and it does not have a default it needs a pointee. So, resource allocation is initialization the moment this construction is happening, it is initialized the pointee that it contains is initialized with a given pointer to a allocated managed object.

Similarly, for it can be it should be possible to copy it by construction and constructed by copying it should be possible to copy it by assignment. And when we do these kinds of copy operations, we have to consider whether that copy is a smart copy, I am sorry, whether that copy is a deep copy or that is a shallow copy also.

(Refer Slide Time: 16:30)

So, this is the basic thing then we need the destructor and what the destructor has to do, destructor will necessarily call delete pointee by calling the delete pointee, it ensures that RRIDs and for that if the manager is managed at this smart pointer object is getting out of scope. So, therefore, it is getting deleted, which is getting destructed then the managed object will also be also vanish. So, Now, we can see that this pointee is basically a dynamically created object and is to be dynamically constructed and destructed, but its semantics of lifetime is dependent on the constructor and the destructor which we know, pretty much can be used in the as an automatic object.

(Refer Slide Time: 17:36

So, this is about the basic lifetime management issues. Now, we need it to be a pointer. So, we need to overload. So, this is basically solved by overloading that they can dereference the smart pointer to get the value that you are referring to. So, whatever the pointee is referring to, so, that is why this is T&, this a. So, if I draw it out, then this is the smart pointer, this as the pointee, the object is object. So, if I dereference this, I should get this and that is what is being done by overloading this dereferencing operator. Similar thing happens with the indirection operator, it basically returns me the raw pointer and this is the pointee member.

(Refer Slide Time: 18:48)

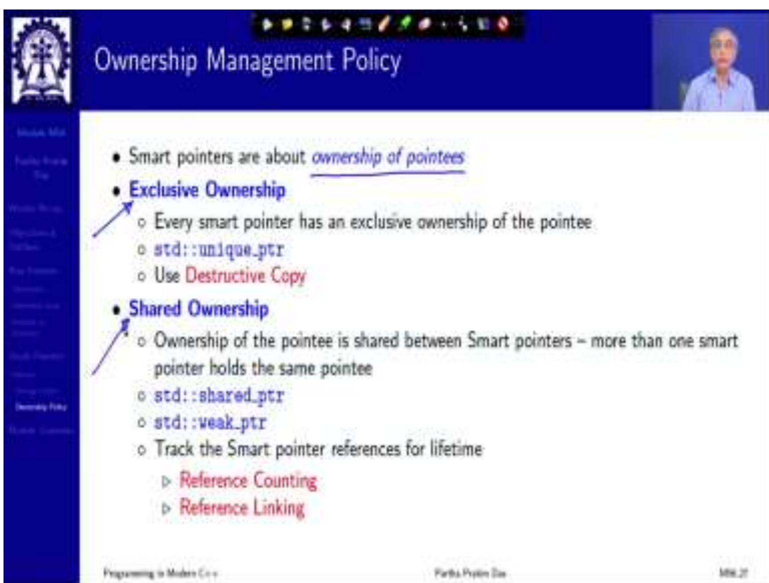
Typical Smart Pointer Use: Mimic a Raw Pointer

```
// A class for use
class MyClass {
public:
    void Function();
    // ...
};

// Create a smart pointer to an object
SmartPtr<MyClass> sp(new MyClass); // RAII: sp takes ownership of the instance

// As if indirecting the raw pointer
sp->Function(); // (sp.operator->())->Function()

// As if dereferencing the raw pointer
(*sp).Function();
```

Now, how do you use it, exactly as you use the raw pointer. So, I have a I have MyClass with function, I create a managed MyClass instance I do a new MyClass and pass it immediately to the constructor of smart pointer, templatized with MyClass. So, it is immediately sent here. So, as soon as the resource is allocated it is captured by this sp and then I can do as I do in a pointer similar way I can do sp pointer function because sp pointer function is sp dot operator in direction and what will that give me? That will give me the pointer, the internal pointee pointer and on that the function will be called which is a normal code or it can work as dereferencing exactly in the same way.

So, with this two operator overloading and these basic construction, destruction, copy move kind of operations, I can have a smart pointer, which behaves exactly like the raw pointer.

(Refer Slide Time: 20:14)



In terms of the, I mean, which mimics the raw pointer, but can Now, behave in a multiple of different ways. So, there is a basic charter expectation of how a smart pointer will be. It must always point either to a valid allocated object or is null. It cannot ever point to something which is an invalid object. It deletes the object once there is no more reference to it. If that object is not being referred, it will have to be deleted, it must be fast that is with minimal overhead. Raw pointers only will be converted to smart pointers explicitly. It can be used with the existing code where I am not going to rewrite the earlier code.

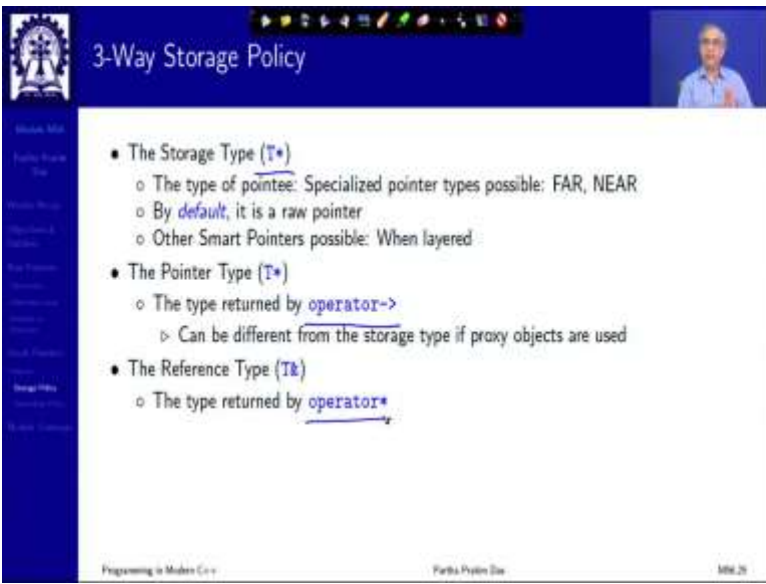
Programs that do not use low level stuff will be written exclusively using this pointer, no raw pointer is needed only if you do low level stuff that is where you need the address arithmetic kind of operators and therefore, you will need to have the raw pointers. It must be thread safe,

it must be exception safe and it should not have problems with circular references, we will see what does that mean and programmers vouch that they will not keep raw pointers and smart pointers to the same object.

(Refer Slide Time: 22:01)

Now, to achieve the charter, certain policies have been identified for smart pointers that bring in flexibility and also lead to different flavors of smart pointers according to requirement. The policies include storage policy, ownership policy, conversion policy, null test policy, et cetera. We will use we will discuss some of them in this module and the rest in the next.

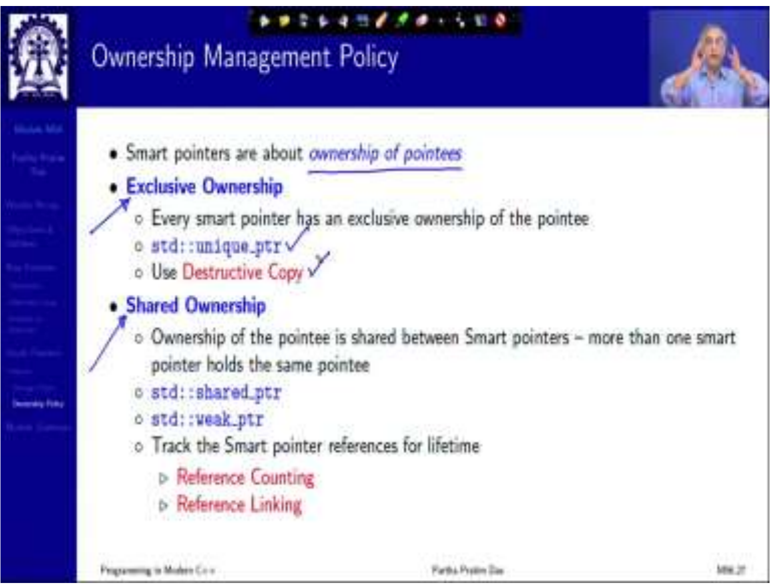
(Refer Slide Time: 22:32)



So, first the storage policy it is very simple that you have to define for a smart pointer there are three storage policy requirements, one is what is a storage type? What is the type of pointer that you are actually storing? It could be a far pointer and near pointer, layered pointer, layered pointer means a smart pointer which is pointing to another smart pointer, it could be layered. What is the type of the pointee, the object that you are dealing with that is a T* and what is the point, what is the type of the reference, the type that is returned by operator star.

So, what is the actual storage type and what returns by indirection operator and what returns by dereferencing operator, these are, often times these all will be same based on the same T but it they can be different as well.

(Refer Slide Time: 23:37)



What really makes a difference and that is the smartness typically most focused on is the ownership policy that we identified at the very beginning discussing raw pointers and the main problem of raw pointers is they may not own the object that they are pointing to, their ownership is broken. So, the smart pointers are about ownership of pointers and two types of ownerships are identified, one is exclusive ownership.

That is in an exclusive ownership, every smart pointer has an exclusive ownership of the pointee. This is only this the object being pointed to is pointed to by only this smart pointer none else can point to it. The library provides a smart pointer called unique ptr for this purpose, and exclusive ownership means destructive copy.

I will explain what destructive copies. Shared point is, on the other hand, deals with the fact that the point is shared between multiple shared pointers that is more than one smart pointer is pointing to it. Library provides two types of shared ownership, shared_ptr and weak_ptr. We will explain later. And the since multiple pointers are pointing to the same object, I need to keep a count of how many pointers are pointing to me for the lifetime management, that is called reference counting.

(Refer Slide Time: 25:32)

Now, first the exclusive ownership, since the ownership is exclusive, what will happen if I copy, the ownership has to just get transferred that is a source will no longer continue to own the pointed object, the source says one pointer will become null on nullptr. And the destination will own this, this is what the unique_ptr shared pointer smart pointers do this is implemented in copy operations.

(Refer Slide Time: 26:08)

Ownership Policy: Destructive Copy

```
template <class T>
class SmartPtr {
public:
    SmartPtr(SmartPtr& src) { // Src ptr is not const
        pointee_ = src.pointee_; // Copy
        src.pointee_ = 0;        // Remove ownership for src ptr
    }
    SmartPtr& operator=(SmartPtr& src) { // Src ptr is not const
        if (this != &src) {          // Check & skip self-copy
            delete pointee_;         // Release destination object
            pointee_ = src.pointee_; // Assignment
            src.pointee_ = 0;        // Remove ownership for src ptr
        }
        return *this; // Return the assigned Smart Pointer
    } ...
};
```

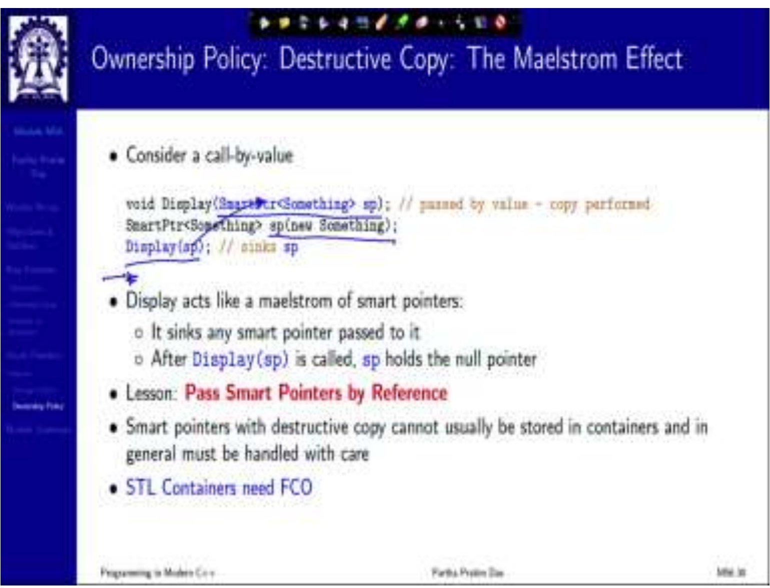
So, you can easily see that I have the pointee, so, if I have copy constructing, I am simply copying that that means is a shallow copy to our referencing here, but the important part is I set source dot pointee to null. So, I removed the ownership of the source that is the reason. The signature of this particular copy constructor is different, it does not have a const. You can

also provide necessary move construction here. Similarly, for the copy assignment operator, you do not have the const because you are going to take away for assignment you are going to take away the ownership again.

So, you check for the self copy, which is the same you delete whatever you have been pointing to because you are taking ownership of a new object. So, the earlier object that you are pointing to, you are the only person who is pointing to it because you are unique exclusive. So, that object that pointer will get lost. So, that object has to lifetime has to also end, so, you delete that copy from the source and set the source to null, so, that you have taken pointer.

So, this is basically so this is what is known as destructive copy, which is the core idea of exclusive ownership.

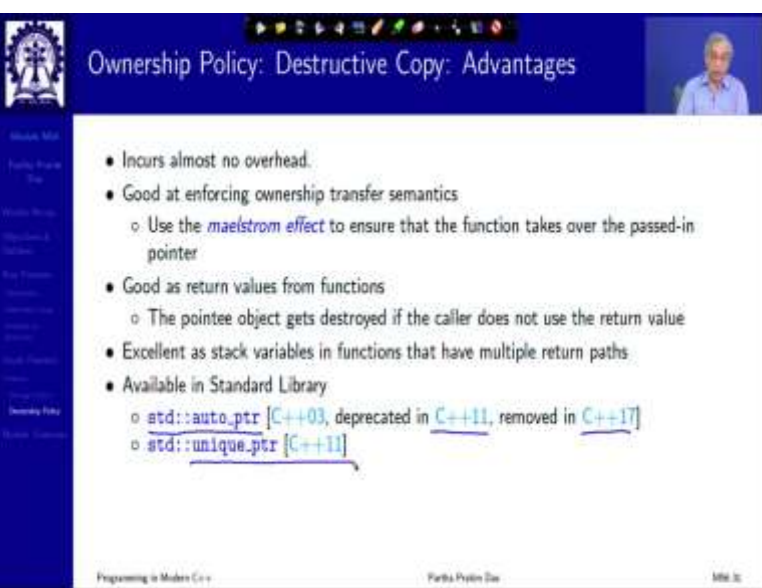
(Refer Slide Time: 27:37)



Now, that creates some effects if you use the destructive copy smart pointer as a parameter as a call by value parameter. So, suppose I have done this call by value parameter to a function and I have a smart pointer exclusive ownership smart pointer created and a call display. Now, what will happen as a call display, this sp will get copied to the formal parameter of the function display. And as you do the copy the ownership will get transferred to the smart pointer that exists in the display as a formal parameter and the actual parameter sp will lose the ownership.

So, after display returns the display will point to nothing, it will have a null pointer, it holds a null pointer. So, it is whenever you do not want this to happen, you must pass the smart pointers by reference. And that is that is precisely what the STL containers need.

(Refer Slide Time: 28:56)



So, the advantage is it has almost no overhead and it is good for enforcing transfer of ownership semantics. You can use the Maelstrom effect, Maelstrom is like a whirlpool, which pulls something down, to ensure that the function take over the past in pointer if you want. It is particularly good as return value from functions.

And in terms of the standard library support C++03 had only one type of smart pointer and that was this destructive copy it is known by auto_ptr but subsequently it has been deprecated in C++11 And it has been removed from C++17 that from C++17, if you have auto_ptr in your code, the code will simply not compile. And Now, what you have in place is a unique_ptr. We will talk more about that later.

(Refer Slide Time: 29:52)

Now, the other side if you have a shared ownership then the basic idea is multiple points are there is one object and multiple smart pointers are pointing to it. Now, therefore, how do you decide when to release this object? You cannot release this object as long as some smart pointer is holding it, because that pointer must be using that is a shared logic. So, what you do this very simple idea, you have a reference count that reference count says how many smart pointers are pointing to it. So, often the shared ownership smart pointers are called reference count smart pointers.

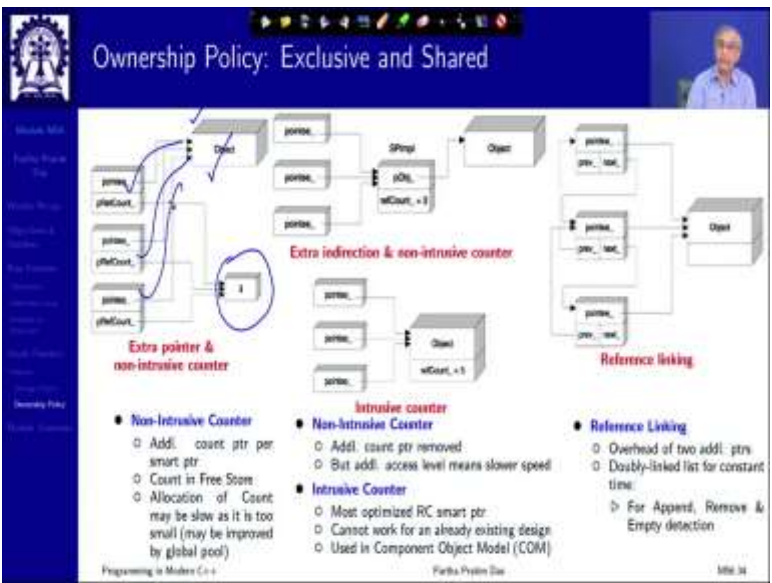
Now, how do you keep the reference count there could be several strategies it could be you could keep it as a part of the object, you could keep it separately and so on, so forth. Another way could be that you could just link those references in a chain instead of you can have this as a chain and both ways chain, you can change the references instead of keeping account. So, that if you destroy a smart pointer, it checks if it is the last one in the chain, which is easy to see. And if it is the last one, it will destruct the pointed object otherwise, it will simply disappear itself from the from there.

And there are two variants of this shared_ptr and weak_ptr we will see why we need this. And it's implemented in these different copy move those kinds of operations.

(Refer Slide Time: 31:41)

So, you have two kinds, exclusive ownership and shared ownership, so on copy in exclusive ownership, the source is set to null, in shared ownership, the reference count is increased. On delete, in exclusive ownership, the pointee object is deleted in shared ownership, the reference count is decremented and if by that it becomes 0, then the object is deleted. So, this is the basic idea.

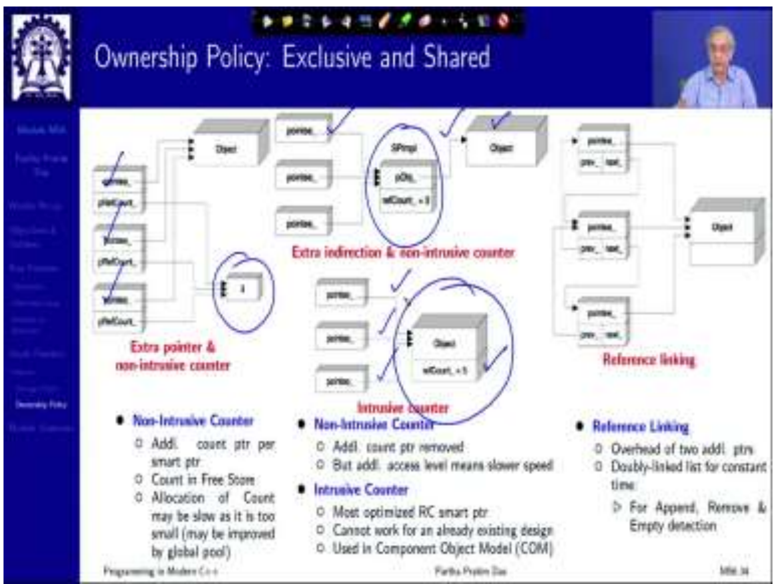
(Refer Slide Time: 32:14)



Now, these are the different kinds of reference mechanism accounting mechanism that you could have, this is like this is the object that you are pointing to. There is a separate object called is a counting counter object, which could be allocated on the heap, which is keeping the count and every smart pointer has a pointer to the object and a pointer to this count. So, if

you have to do any check, if you are created, then you have to go and increment this if you are the smart pointer is getting destroyed, you have to destroy this and if you have to access then you simply dereference these links, simple.

(Refer Slide Time: 33:02)

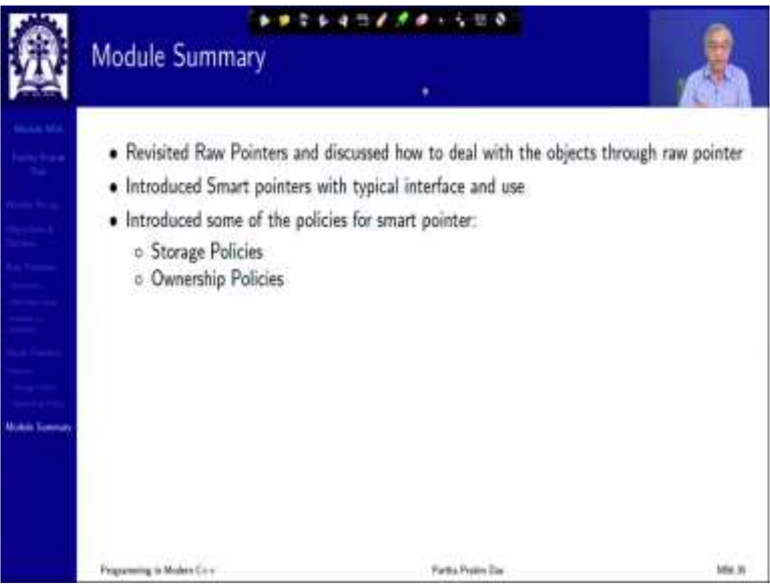


Now, the other way could be that instead of having the pointee in every smart pointer, you could have that in this counting object itself. So, that is a counter and this so every one has now, just one pointer, so you save on space.

But now, to get to the object you have to dereference twice or you could the most efficient is if you can keep it as a part of the object itself, counter is a part of the object itself, in which case you just have to dereference in one more raw over it. But the catch is if it has to be a part of the object, then it will not work for the existing projects because the objects are already there. So, it is a developer who will have to provide this.

So, only in projects, which are closely managed by a company like the windows COM objects, Component Object Model from Microsoft, they use this kind of a model, but in general this cannot be used, otherwise you could link the references.

(Refer Slide Time: 34:14)



So, these are the different test strategies by which the shared ownership policy can be implemented. So, to summarize, we have visited the raw pointers and discussed how to deal with objects through raw pointers. And we have introduced smart pointers with typical interface and use and we have discussed policies of storage and ownership policies, we have discussed the basic exclusive and shared ownership, but there are more to add to this, which we will do in the next module. Thank you very much for your attention. See you in the next module.