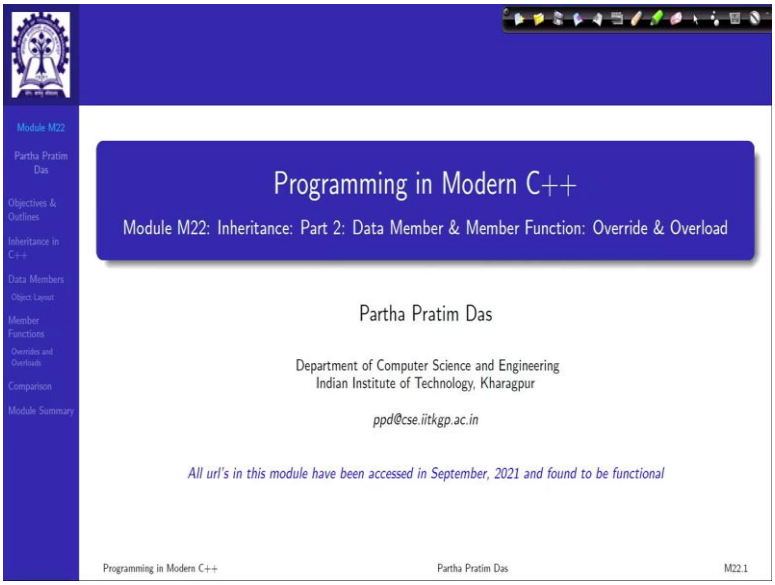
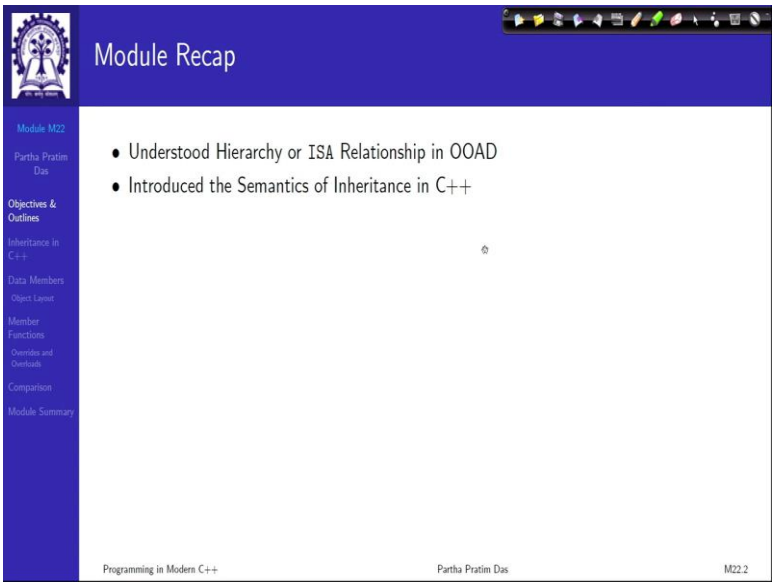
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Lecture 22
Inheritance: Part 2
Data Member and Member Function - Override and Overload

(Refer Slide Time: 00:35)



Welcome to Programming in Modern C++, we are in week 5 and we have been discussing inheritance in C++. We are going to now talk about module 22.

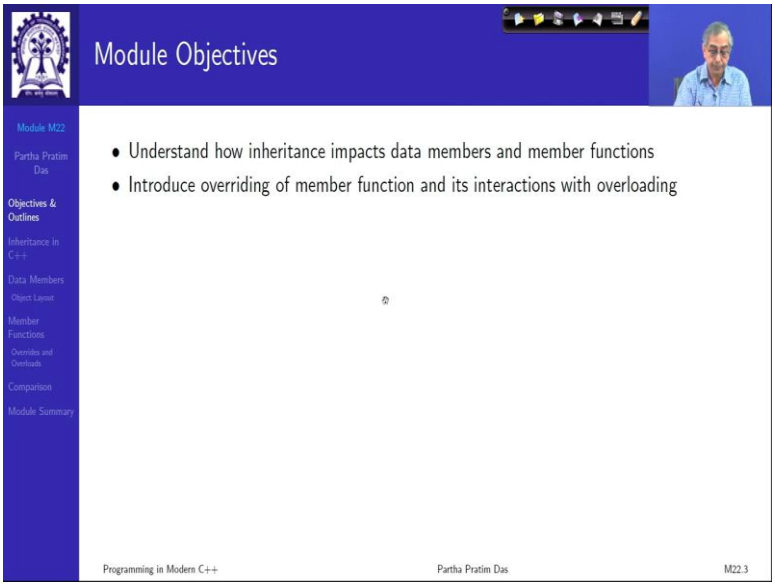
(Refer Slide Time: 00:42)



In module 21, in the last module, we understood the basic concept of ISA Relationship in Object Oriented Analysis and Design. We talked about various kinds of hierarchical

relationships that is possible in terms of single inheritance, multi-level inheritance, hybrid inheritance and so on. And we introduce the basic semantics of inheritance in terms of C++.

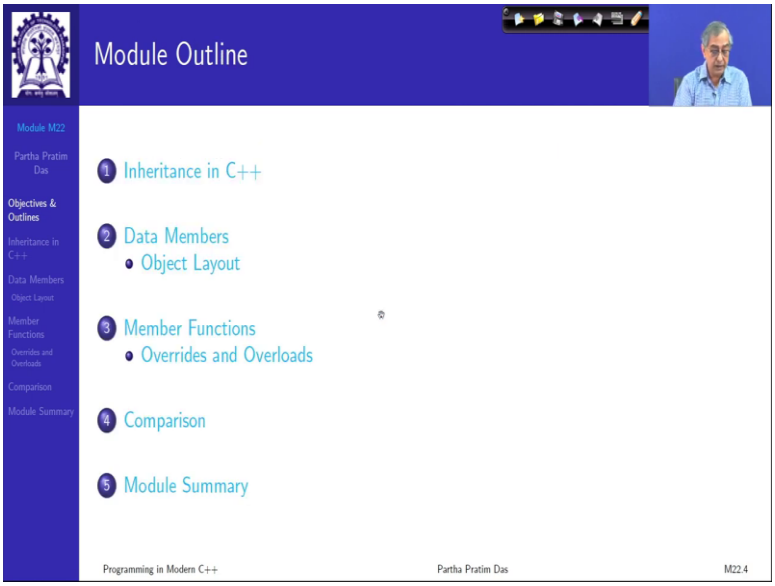
(Refer Slide Time: 01:07)



In the current module, we are going to build upon that and we are going to first take a look at the two basic semantics of inheritance that is how to manage data members between base classes and derived classes and how to manage member functions between base classes and derived classes. And then in terms of the member function, we will discuss about what is meant by overriding, overriding mind you.

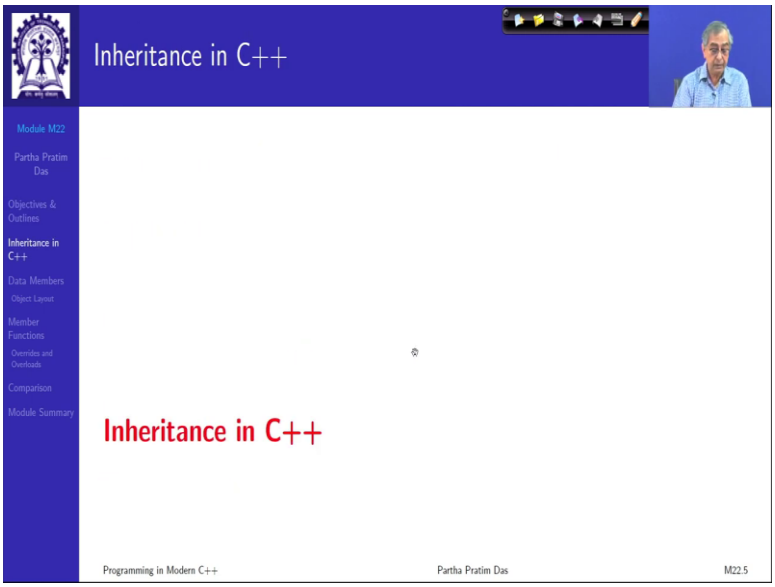
You have been familiar with overloading we are introducing another concept another term overriding what is meant by overriding and how does that interact with overloading.

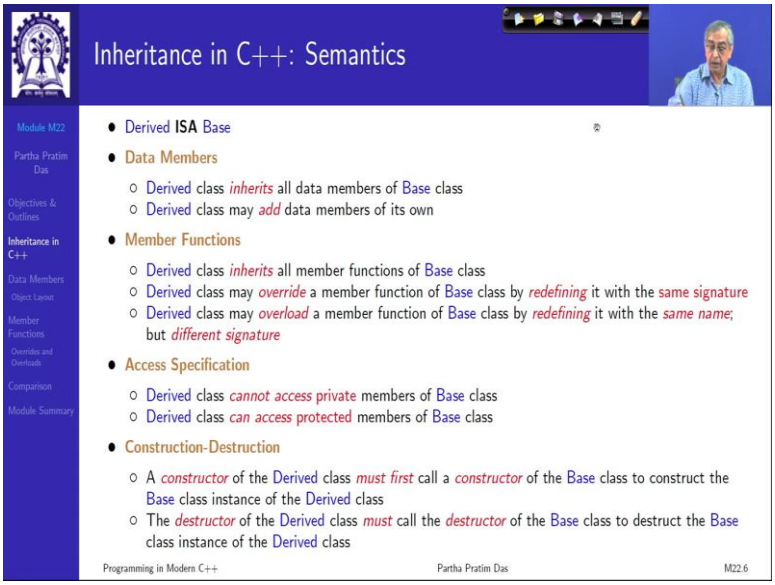
(Refer Slide Time: 01:54)



So, these are the, this is the outline as you can.

(Refer Slide Time: 01:57)





So, this is what when we left in the last module that these are for future, this is what we are going to do here that we will see that how a derived class inherits all the data members of the base class, how does it inherit all member functions of the base class, and then how can it add its own data members. The derived class can add its own data members or add its own member functions, modify the member function behaviour in from the base class and so on.

So, it is called inheritance because it works much in the same way the real-life inheritance that like we regularly inherit assets and liabilities from our parents. And when we do that, we get everything say we get assets and liabilities, we can add more assets to it, we may create

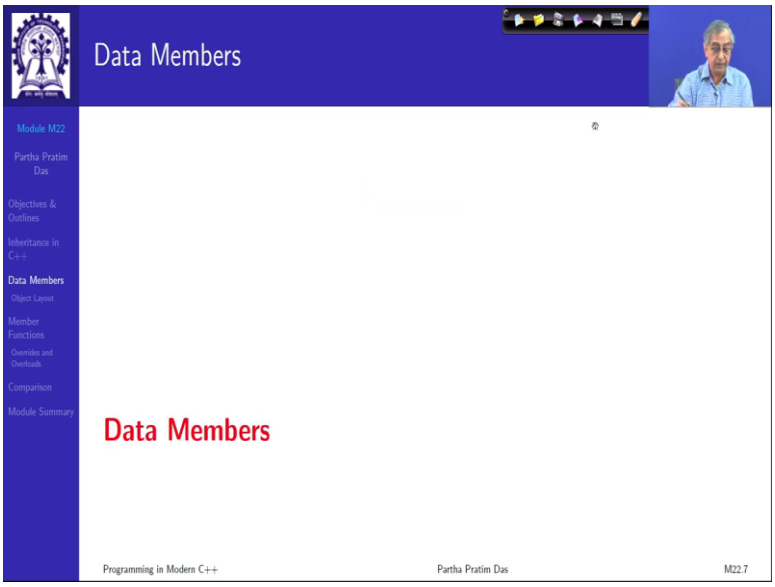
more liabilities to that, but by normally being the child of the parent, we get everything that they leave behind or that they choose to give us.

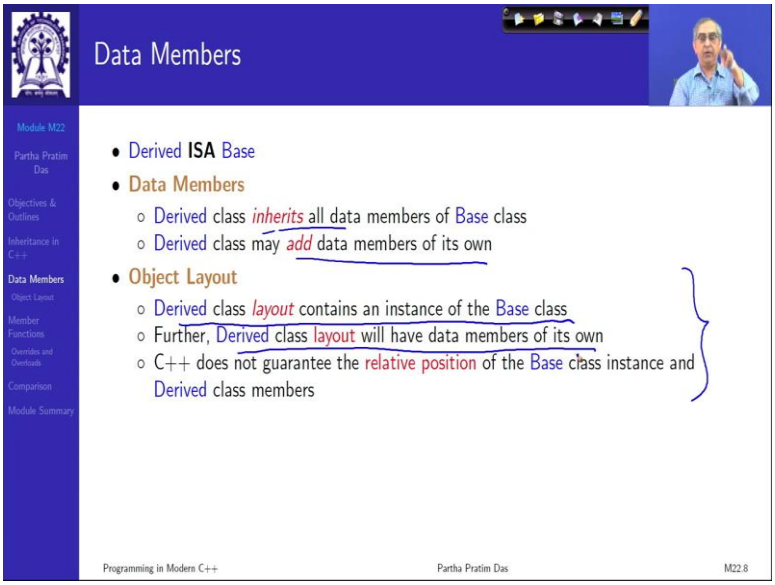
So, this is, we said this is inheriting you become an heir to that assets and liabilities and that is the same concept that we are trying to do here. And in this I would like to remind you that when you inherit, you do not have a choice what you inherit, you cannot disinherit. What it means is your parents let you inherit the assets and liabilities.

Now, what I would have liked as a child, that I will inherit the assets but I will disinherit the liability, I will say no, no, no. If my parents, had a loan for 5 lakh I have not inherited that, it is not my liability to pay, no, you cannot do that. So, inheritance comes with the fact that I cannot disinherit and the similar concept exists in the inheritance of C++ as well.

But I can add assets, I may build up more liability and I can have different behaviours what I have inherited, so with that basic idea of the of the semantics of inheritance, let us first take a look at the data members.

(Refer Slide Time: 04:28)



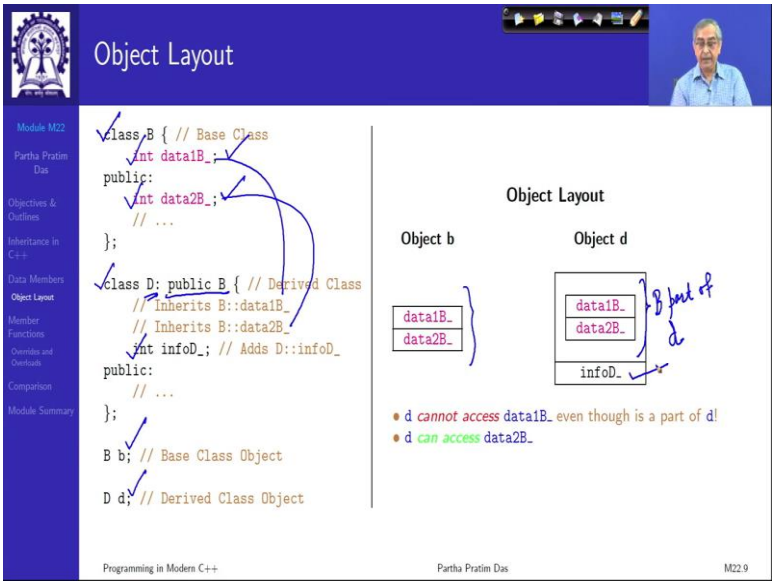


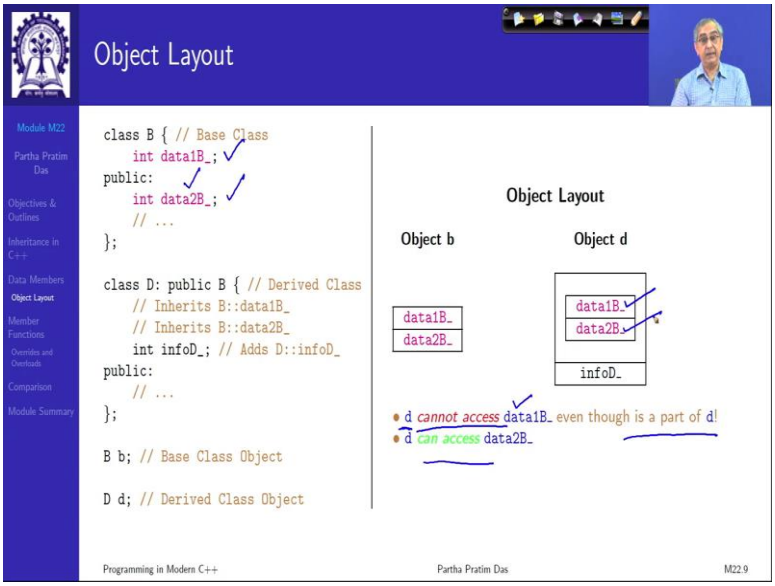
So, it inherits all the data members can add data members of its own, basic point. Now, when we do this, what happens to the layout of the object? We know that the layout of the object in a simple class is an aggregation. This is like a structure. So, data members in the order in which I have listed them they occur in that way in the memory one after the other and we have discussed about this in terms of simple classes.

Now here, since it inherits all the data members of the base class, the derived class layout must have a complete instance of a base class, we said the base class part of the derived class, and that is a new thing that is going to happen. In addition, the layout will have the data members that the derived class is adding.

Of course, in terms of the layout C++ standard does not guarantee the relative position of the base class instance and the derived class instance that is whether first the base class will be there, then the derived class will be there or in other ways or in some other form that is not guaranteed. So, whatever we are presenting here is just, what is commonly found, it is, you can actually write programme to find out what the layout is, it is not difficult to do, you can just get addresses of different data members, and you will know how they are laid out. But this is just indicative, do not think that this is prescribed or mandated by the standard.

(Refer Slide Time: 06:10)





So, to take a look at the object layout, we consider a class B, which has a private member, and public data member. And then I have class D, which is a specialisation of B, so it inherits both these data members, and then it adds a new one. So, let us see what the layout would be. So, I have a base class object small b, I have a derived class object small d.

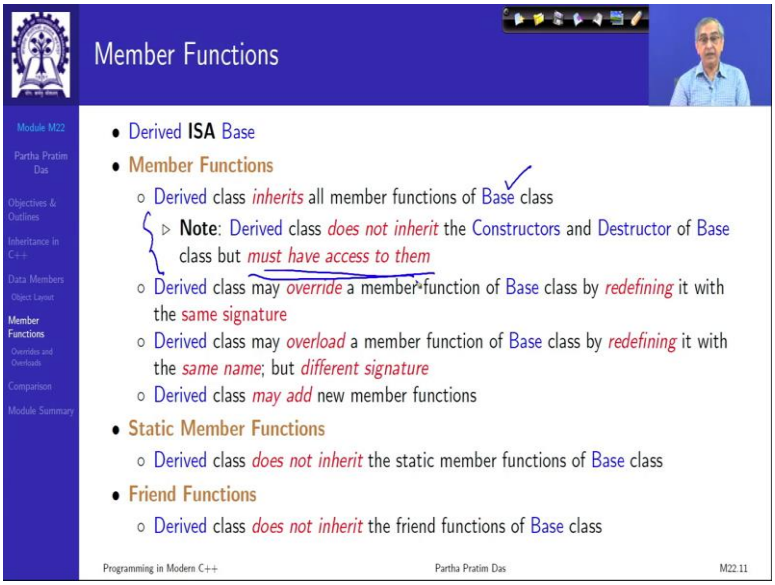
So, if I look at this instance, I will see that a base class object b will have this member and this member quite naturally, that is a simple layout. In terms of a derived class object, I will have a base class instance for it. So, this is the b part of object d, object d is of class D, capital D, but it will have a base part which is exactly the object of class B. And then it has the additional which is additional member of, member data member for the derived class.

And you can see that this is just for a single level inheritance if you have multi-level inheritance, the contentment will keep on growing. Now, what does these access restrictions

tell me, it tells us that d, the object derived class object D cannot access this data member, even though it contains it.

Now, this is a situation which did not happen before, because everything was in terms of one class. But now, I have a data member in d which is populated some way because d directly cannot do it, because it does not have access to that. So, though it is a part it cannot be accessed, whereas this is public. So, this can be easily accessed. So, this is the basic semantics of the and object layout.

(Refer Slide Time: 08:21)

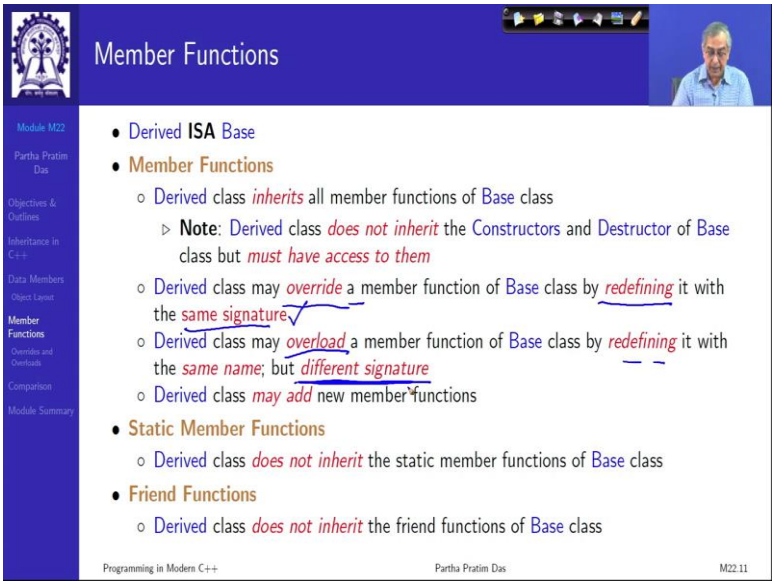


So, next, let us move on to the member functions, see what the member functions can do. So, in terms of the member functions, we say that the derived class will inherit all member functions of the base class. So, it can use all of them. Mind you, there has to, there is a caveat

to that which you must remember that derived class does not inherit the constructor or the destructor.

Because the constructor and destructor have class specific names, the constructor of the base class will be called Base, the destructor will be called ~Base. So, they are not inherited in the way that you can use them as base class function, but you must have access to them. Because unless you have access to them, you will not be able to construct the base part of the derived class object, we will see more of this when you go into construction destruction.

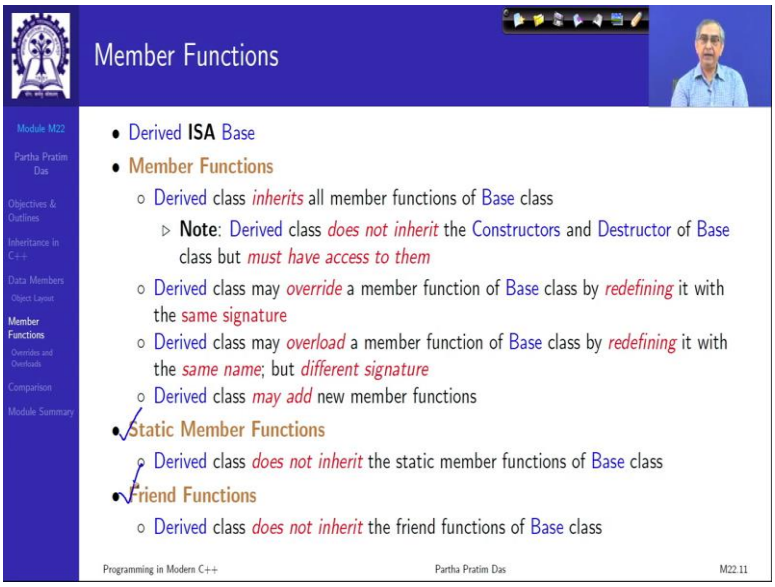
(Refer Slide Time: 09:21)



Now, there are two special properties that a derived class can overwrite a member function it is inherited, by redefining with the same signature, it does not change the same, does not change the signature, but gives a different implementation gives a different body to it. If it does not give whatever it has inherited, we would work. But here, it can choose and give a different signature to it, give a different implementation to it keeping the signature same.

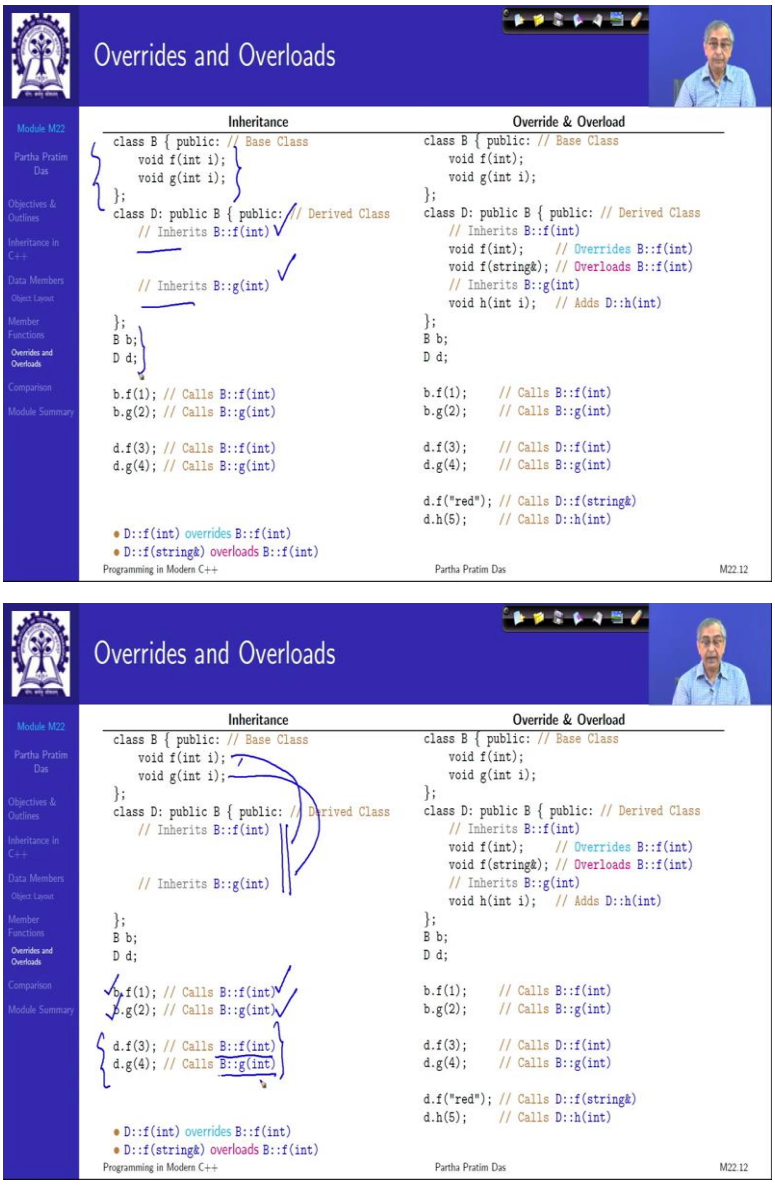
Or it can choose to overload when it redefines with the same name, but with a different signature, you know that is the basic rule of overloading. So, it becomes yet another different function, overloaded member function or it can add new member functions as well.

(Refer Slide Time: 10:15)



Mind you these are, these cannot be inherited static member function or friend function cannot be inherited, which means that in simple terms it means that anything that can be inherited must have this pointer. So, it is only non-static member functions which can be inherited by this process by the derived class from the base class.

(Refer Slide Time: 10:40)

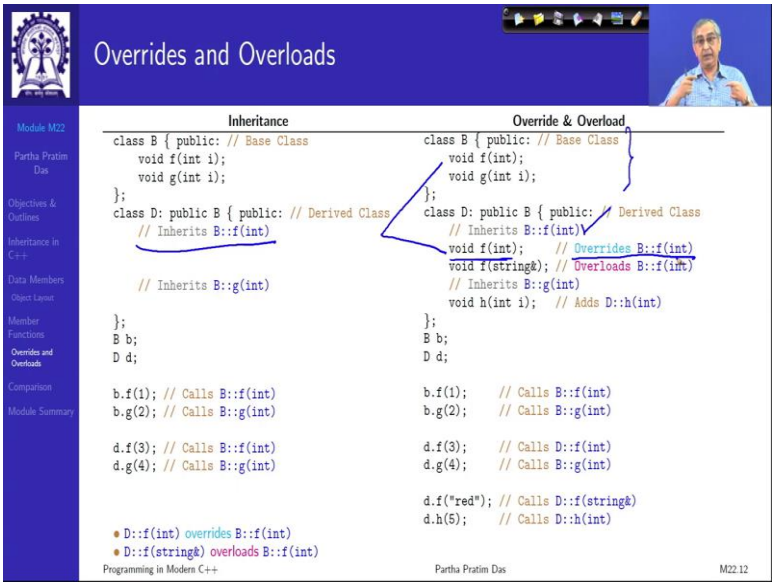


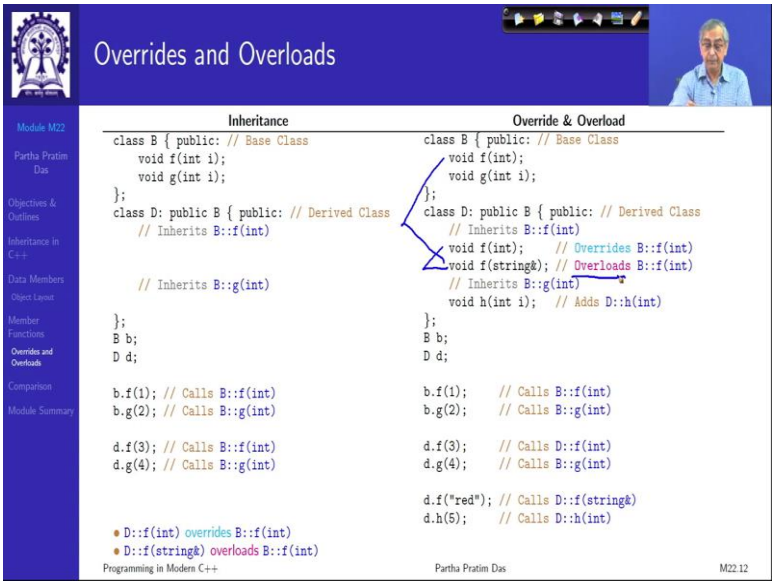
So, let me with a simple example illustrate what is the difference between the override and the overload the two terms. So, the first here is an inheritance. There is a class B. And for simplicity, I have kept everything in public. Because otherwise, the derived class will not be able to use it because we have not studied protected access yet. So, everything is public. In the base class, you have two functions f() and g().

So, in the derived class, you have inherited them, commented out, I have not written anything. So, if I construct two objects b and d of the respective base and derive classes, then what is the behaviour I will get, if I do b.f(1), that is a simple case, like we were doing, b is accessing function f() with parameter 1. So, it calls the function f() of b. If I call b.g(2) this function it is b::g(), as we had.

The interesting thing is if I do this with the d, object of the derived class also these two functions will be called because they are inherited here. So, they have not been written you do not see any function, but actually this function is available here, this function is available here. So, you get to call b : f and b : g by the object d of the derived class D.

(Refer Slide Time: 12:25)





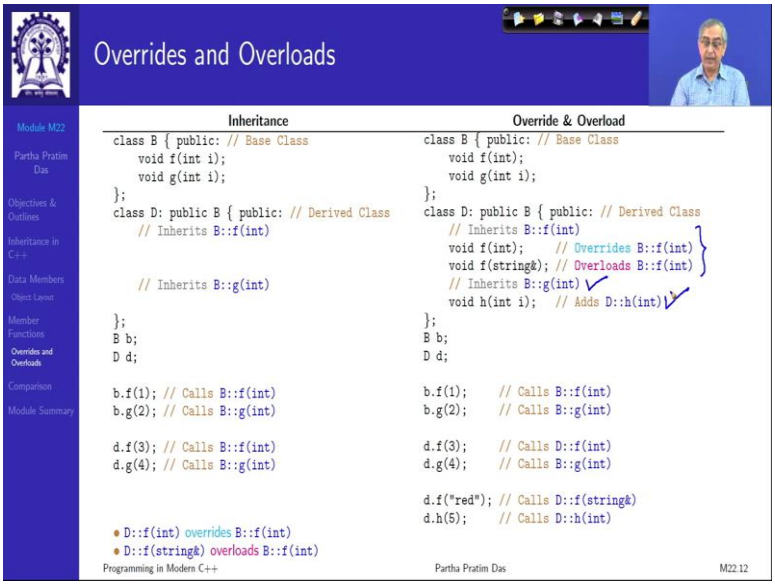
This is the simple inheritance mechanism of member functions. Now, how do I control this, I again have the same base class I have not done anything, I have inherited B::f, so it is available. What I do is, I have again written the same signature, mind you, between this and this, between this and this, the signature is same here, I did not write it, but here I have written it again.

When I write it again it means that well, I have inherited f() which takes int and returns void, but I want to give it a different definition, I want to give it a different implementation and that is what is called overriding. It is not overloading because function signature is not different, what is different is one function belongs to class B the other function belongs to class D.

And the way you let the compiler know is by including an identical signature in the derived class. What I have further done is I have given another function f() here, another member function f() here, which takes the parameter of reference to string. Now, this is not same signature as the original f() function. So, between the original f() function and this or between the overridden f() function and this the signature is different.

So, this actually is the overload of the B::f. So, here I have done two things together one is I have overridden by giving the same function signature, but intending a different body which I will write. And by giving a function with the same name, but a different signature, for which again, I will give a body which overloads the original member function.

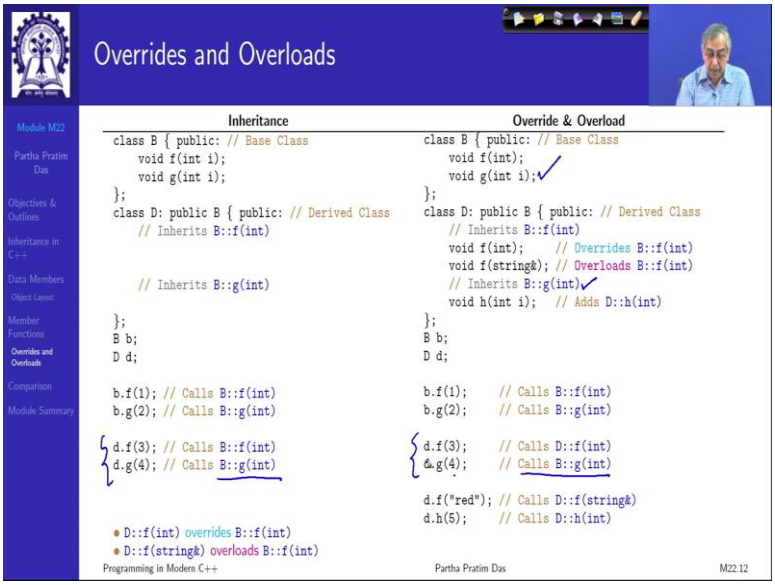
(Refer Slide Time: 14:41)



So, this is what I have done here with g() I have not done anything. So, g() I have just inherited, f() I have inherited and overridden, f() I have inherited and overloaded and then

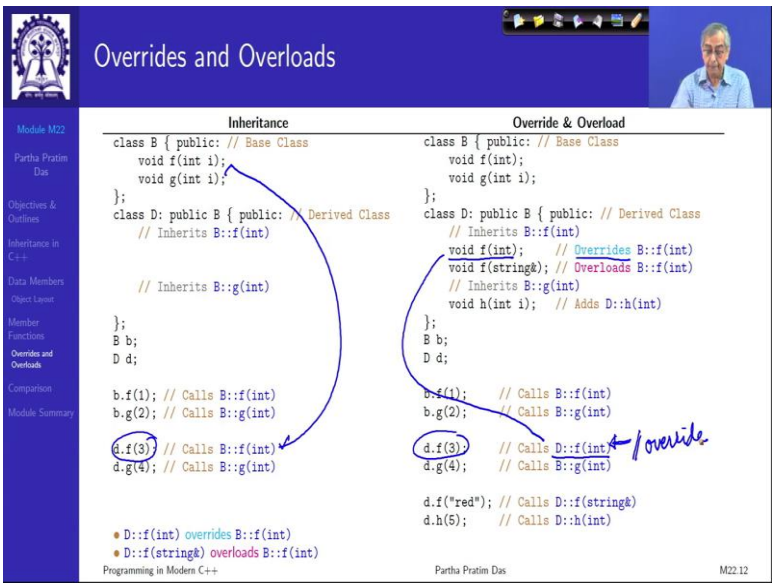
finally, h() is a new function that I have defined only for D, it does not have a correspondent in the base class. So, these are the four possibilities I can inherit, I can inherit an override, I can inherit an overload, or I can give a new function. So, these are the four cases available.

(Refer Slide Time: 15:18)



Again, I have two objects created, if I make the original call as before, naturally, I will have the same function calls, there is no difference because base class I have not touched. Now, what happens if I call make these two calls again in this context. Now, g() I have not touched, I have just inherited. So, g will call B::g, that is this function in any case.

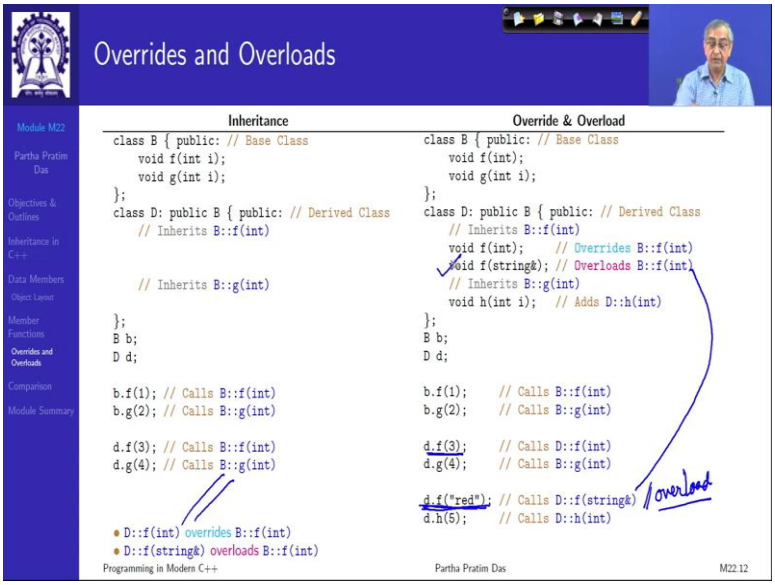
(Refer Slide Time: 15:49)



But what happens to this particular call. Earlier it was calling B::f, because I just inherited did not do anything. Now, I have inherited and I have put it here again to override. So, when it is

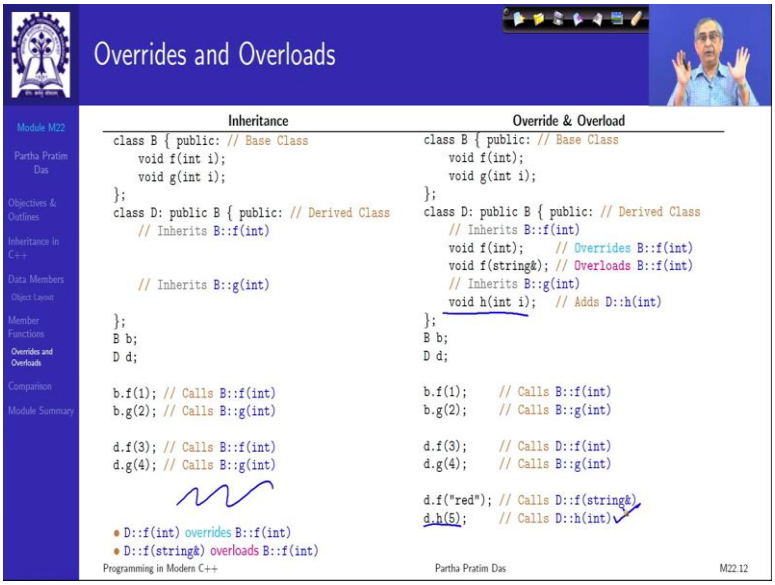
override (overridden), then it is the most recently overridden function is what will get called. So, this calls D::f that is this function. Whereas, this calls this function that is overriding. So, this is a case of override. I hope you are getting it cleared. So, just studied parallelly.

(Refer Slide Time: 16:37)



Now, I have also called D::f with const char* parameter, which is basically which can basically become a reference to string, a temporary object of string will get created, which I could not have done here, because that kind of a member function did not exist. So, this is, naturally this is going to call the function in D because in B no such function exists. And this is the case of overload because a different signature different from what B had defined, a different signature is being used. So, by overload I will be able to call this function.

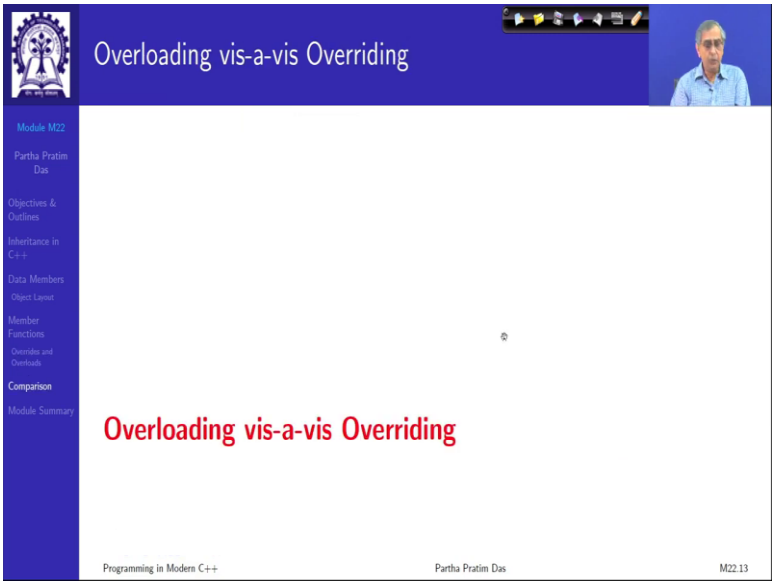
(Refer Slide Time: 17:25)

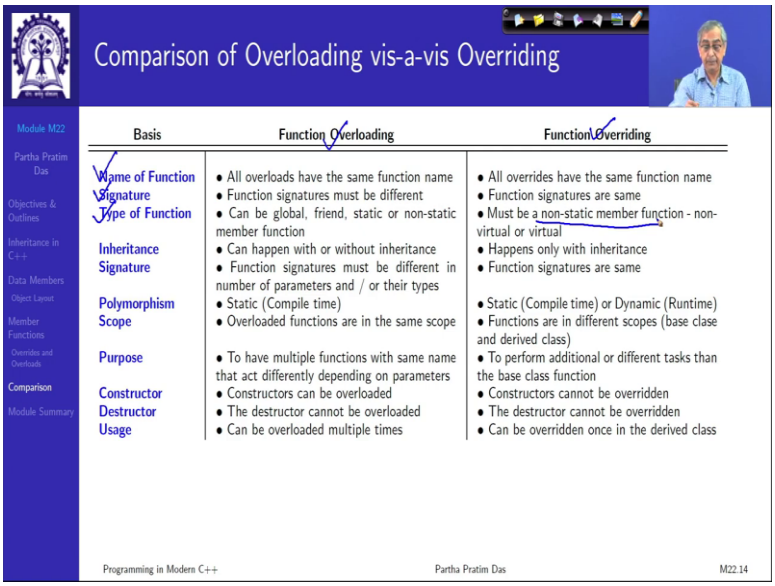


Finally, h() is again a sample case, h() was not here. So, there is no question here, but when I do d.h() a new function member function has been added. So, D::h will be called. So, I think this slide will be a very crisp summary of what happens in member function inheritance, four possible cases that you can have and what is the behaviour if you keep this ready with you handy with you all the time till it gets settled in your mind it will be very useful.

Just to remind you again you inherit all member functions from the base class, you can keep the signature same and still put that signature in your derived class, you are overriding you have to give a new body, you can keep the function name same give a different signature and give that body you are overloading the member function or you can add new member functions. So, this is the whole scenario of member function semantics in inheritance.

(Refer Slide Time: 18:37)

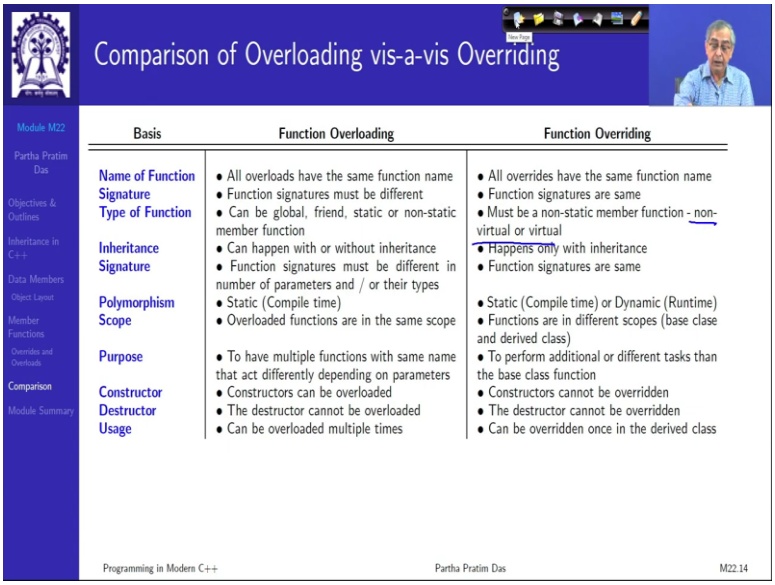




So, quickly we can take a comparative look also. This is just summarising what I have been saying the name of the function. Certainly, whether you over, this is comparing overloading and overriding. Overloads always have the same function name, overrides also have the same function name otherwise, the context is not arranged.

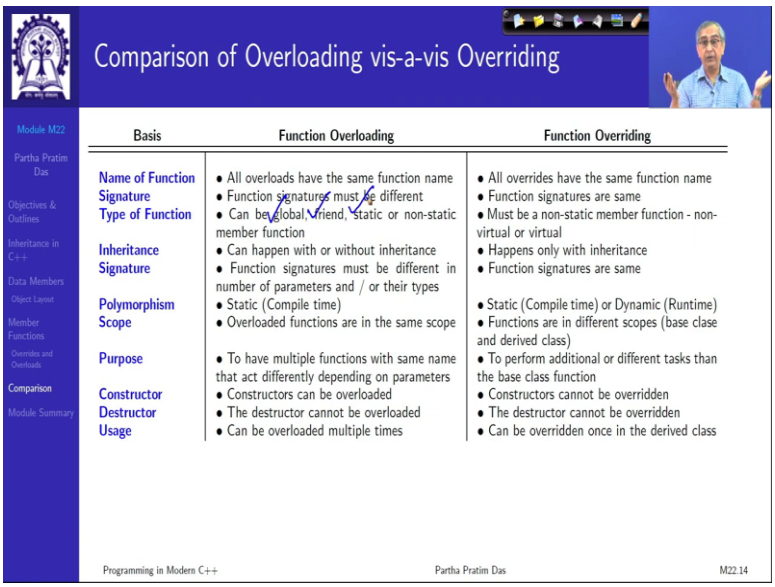
Signature is for overload the signatures must be different, for override the signatures must be the same. What type of function can be overloaded? We have seen global functions can be overloaded, friend functions can be overloaded, static member functions can be overloaded, non-static member functions can be overloaded. All of these can be overloaded. But overloading as I have already mentioned can be done for only non-static member functions.

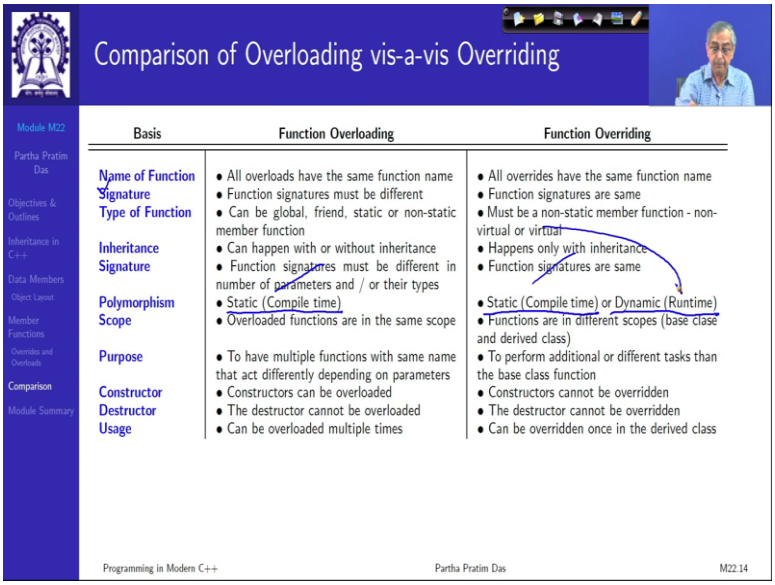
(Refer Slide Time: 19:25)



You may find that I have written some terms after this which you are not familiar with. Do not worry about that. We will learn about it terms of dynamic polymorphism. And at that time, I will clarify what they mean. And you will understand the extensional behaviour of override in that context.

(Refer Slide Time: 19:45)





Then, what is the role of inheritance? Certainly, function overloading has nothing to do with inheritance it can be for example, in these cases it works without any inheritance, here it can work without or with inheritance, but overriding works only in the context of inheritance. Signatures, I think the signature line has been repeated twice. So, you can just ignore this, we have already talked about it, I will remove it from the slide subsequently.

There is a polymorphism, we have talked about polymorphism in the context of overloading saying, when you have the same function giving you two or more different behaviour, you have multiple morphed behaviour. So, which you say is polymorphism. Now, in terms of overloading it is necessarily at compile time. In terms of overriding, it could be compiled time, like the example that I have just shown is compiled time.

But what is interesting more is when you start having overriding in the dynamic context, which is what will happen when we talk about virtual, so that will come in the future. Scope, obviously, overloaded functions are always in the same scope, but the overriding is happening in two different scopes. So, you have a member function in the base class scope, and you have a member function as a derived class scope.

So, it is happening across scopes, overloading will always happen in the same scope. The purpose is clear to have multiple behaviour for overloading by the same function name, but for overriding the main purpose is to perform a different task than what you have got. So, you get a behaviour, but you are trying to modify that behaviour, it is only applicable in that context.

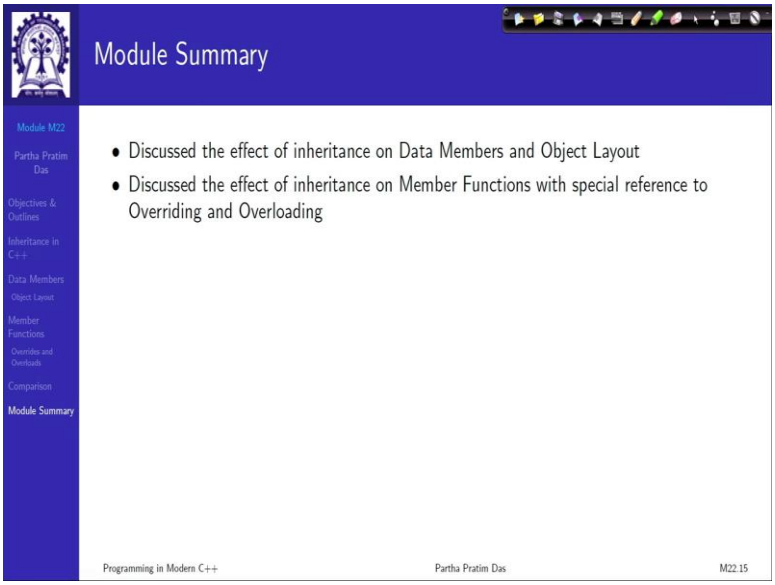
Naturally, constructors can be overloaded, constructors cannot be overridden, you can even understand for in multiple ways one is the constructor in the base, in the context of the derived class. The constructor of the base class is cannot be conceived because constructor of the base class is the name of the base class. And so, it has no role in that way.

Besides, so, we had already mentioned that the constructor actually is a static class, the static function. Because it does not when the constructor is invoked at that time the object has not been constructed. So, this pointer does not exist. So, constructor cannot be overridden.

The destructor as we have seen cannot be overloaded because destructor has to be unique, it cannot be overridden also, for the simple reason that the destructor of the base class and the destructor of the derived class like constructor have different names. And naturally, overloading can be done multiple times, but overriding can be done only once in a particular derived class.

So, these are the relationship between function overloading and overriding. This is a very, very important topic, it is a very, very important behaviour that will drive the way we deal with different semantics of inheritance all across and therefore, we will you will have to be very careful to understand it, practice it. And as I said, always refer to the slide of base and derived example, with four types of member functions in the derived class.

(Refer Slide Time: 23:44)



So, to summarise, we have dealt with the first part of the semantics of inheritance in C++, we have shown how data members are inherited and the fact that all data members or for that matter the entire instance of a base class is a part of the instance of the derived class and in

the object layout we do not know when they will, where they will occur, C++ standard has not specified but it will be a part.

And interestingly it could be that there are some private members in the base class data members in the base class which the derived class instance will contain but will not be able to access and we have seen the effect of inheritance on member functions, you inherit all member functions, you can override them keeping the signature same giving a new body you can overload them by giving new different signatures or you can add new member functions.

So, equipped with this. We will close this module and we will move on to the next to explore further under semantics. Thank you very much for your attention. And we meet in the next module.