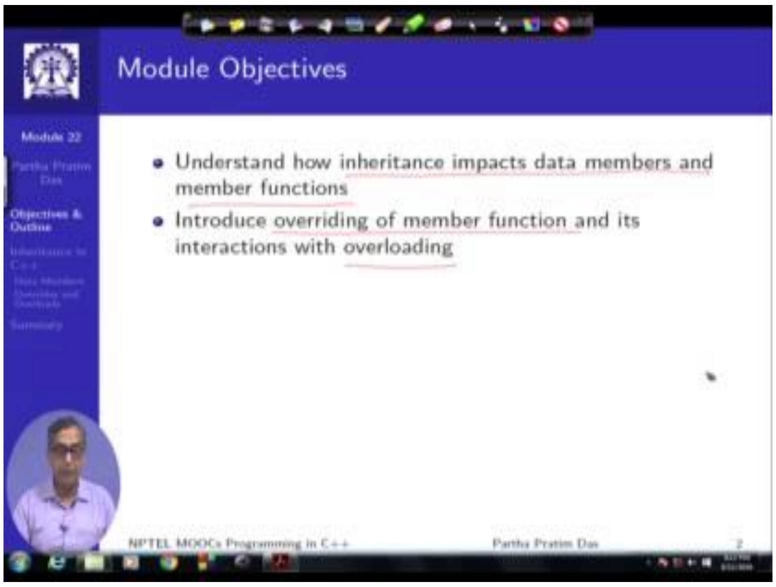
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 37
Inheritance: Part II

Welcome to Module 22 of programming in C++. Since the last module, we have been discussing on inheritance; we have noted that inheritance is a basic mechanism in C++ of modelling, encoding, there is a relationship of object oriented programming, the generalization and specialization relationship.

(Refer Slide Time: 00:33)

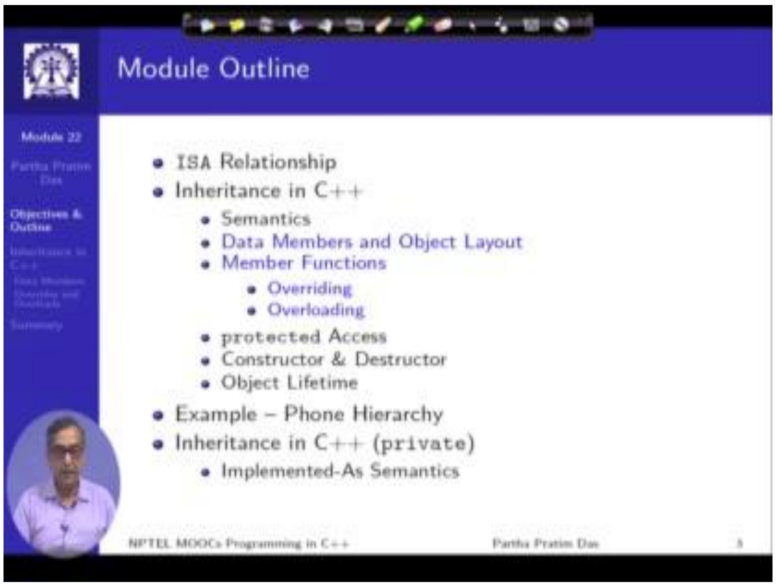


Module Objectives

- Understand how inheritance impacts data members and member functions
- Introduce overriding of member function and its interactions with overloading

And, we have just seen how basic inheritance information can be coded in the language. In this module, we would try to discuss how inheritance impacts data members and member function that is under inheritance, what happens to the data members and member functions. And we will try to understand that what is overriding of member functions, and how does it interact with the concept of overloading that you are already familiar with.

(Refer Slide Time: 01:24)



Module Outline

Module 22

Partha Pratim Das

Objectives & Outline

Inheritance in C++

Data Members

Overrides and Overloads

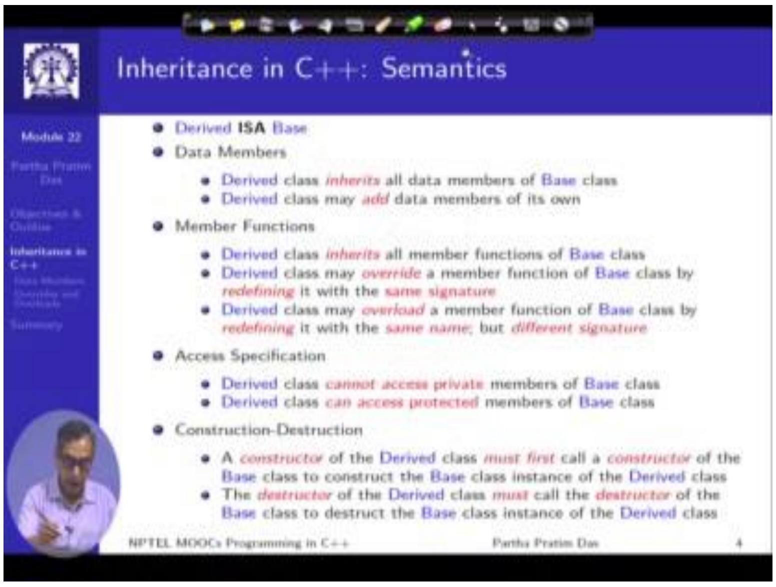
Summary

- ISA Relationship
- Inheritance in C++
  - Semantics
  - Data Members and Object Layout
  - Member Functions
    - Overriding
    - Overloading
  - `protected` Access
  - Constructor & Destructor
  - Object Lifetime
- Example – Phone Hierarchy
- Inheritance in C++ (`private`)
  - Implemented-As Semantics

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

This is the outline and as I had mentioned in the last module, this is the total outline of inheritance; and the blue part is what we will be discussing in this current module.

(Refer Slide Time: 01:41)



Inheritance in C++: Semantics

Module 22

Partha Pratim Das

Objectives & Outline

Inheritance in C++

Data Members

Overrides and Overloads

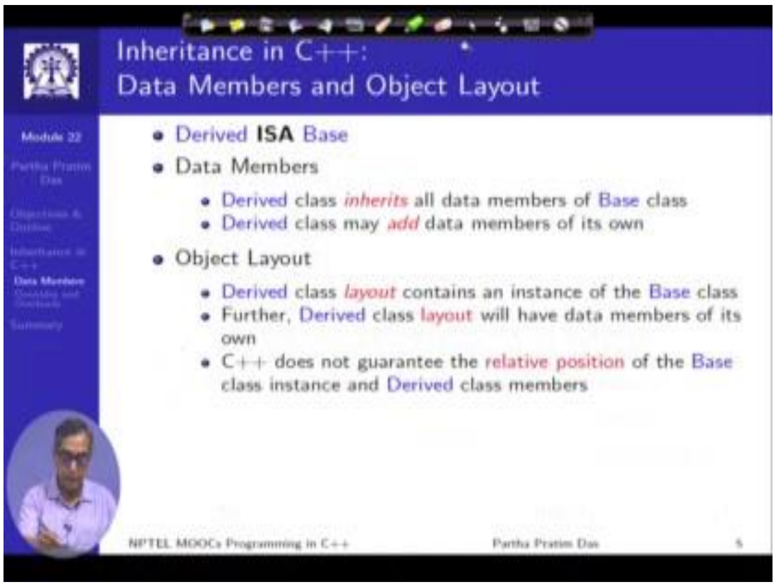
Summary

- Derived **ISA** Base
- Data Members
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the same name; but *different signature*
- Access Specification
  - Derived class cannot access private members of Base class
  - Derived class can access protected members of Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call a *constructor* of the Base class to construct the Base class instance of the Derived class
  - The *destructor* of the Derived class *must* call the *destructor* of the Base class to destruct the Base class instance of the Derived class

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, to recap, this is the inheritance aspects of inheritance semantics. So, we will need to understand about what happens to the data members, member function, access specification, construction, destruction, object lifetime, all these aspects. And here we are discussing starting to discuss about the data members.

(Refer Slide Time: 02:04)



Inheritance in C++: Data Members and Object Layout

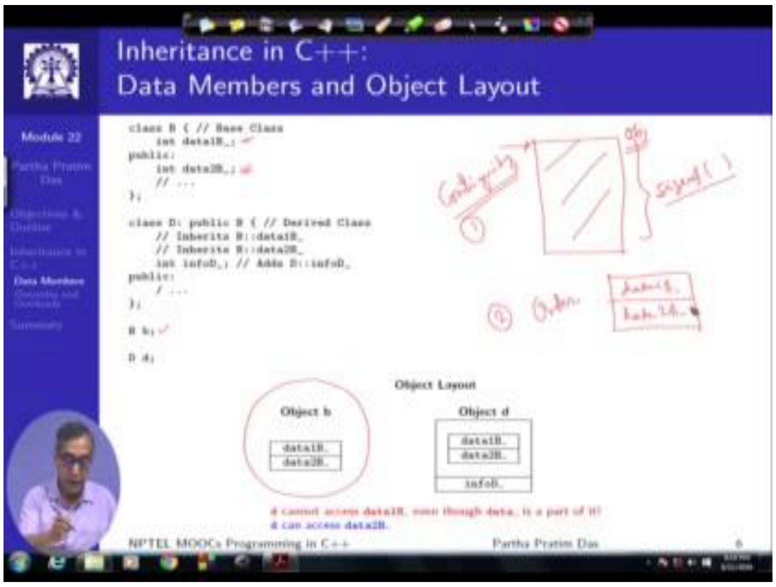
- Derived **ISA** Base
- Data Members
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- Object Layout
  - Derived class *layout* contains an instance of the Base class
  - Further, Derived class layout will have data members of its own
  - C++ does not guarantee the relative position of the Base class instance and Derived class members

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

Now if derived ISA base, ISA basic is a relationship that we are trying to encode. Then we will observe that the derive class will inherit all data members of the base class. This is true in every case of inheritance. Further, the derive class may add more data members of its own it can have some more additional data members. So, if we have that then certainly in terms of the layout of the object, you would recall that we had talked about layout of objects in couple of earlier modules. And if you are not clear about the concepts, I would add that you go back, and revisit those modules before you proceed further. So, the layout is basically the organisation of the different data members within an object of a class in the way that they will get organised in memory.

The important point to be noted is that derived class, the layout of the derive class will contain an instances of the base class. This is something which is new that will come in here. And further naturally, the derive class can have its own members. And unlike the case of sample data members, the relative position of the base class instance, and the derive class members are not guaranteed in C++.

(Refer Slide Time: 04:00)



Inheritance in C++: Data Members and Object Layout

```
class B { // Base Class
    int data1B_;
public:
    int data2B_;
    // ...
};

class D: public B { // Derived Class
    // Inherits B::data1B_
    // Inherits B::data2B_
    int infoD_; // Adds D::infoD_
public:
    // ...
};

B b;

D d;
```

Object Layout

d cannot access data1B_ even though data_ is a part of it!
d can access data2B_

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

Let us, go to an example and start seeing what we mean. So, we start with a base class definition, B is a base class; it has two data members, I have just arbitrary taken two data members, I am not showing the member functions yet, just showing the data members, so data one B, which as I have put is a private member and data two B, which is a public member. Then we have another class D, which is public B; that is it is a specialization. So, if I draw in terms of the UML notation, then this is what we are representing here, D is a B. Now what happens is when we look try to look for the data members of D certainly, what we are saying is guaranty that we will have a data members data one B in the class D as well even though we will not be writing this explicit.

Similarly, we will have a data member data to be in class D by inheritance from class B. So, these are therefore, this is why I am highlighting here these are the data members, which are inherited; they do not have to be there cannot be listed in this class. But the fact that D is a B will inherit this in every instance of D. But I can have some additional members also like the info D that I am putting in here, which is a member of D and was not a member of B; so, this will exist only in this.

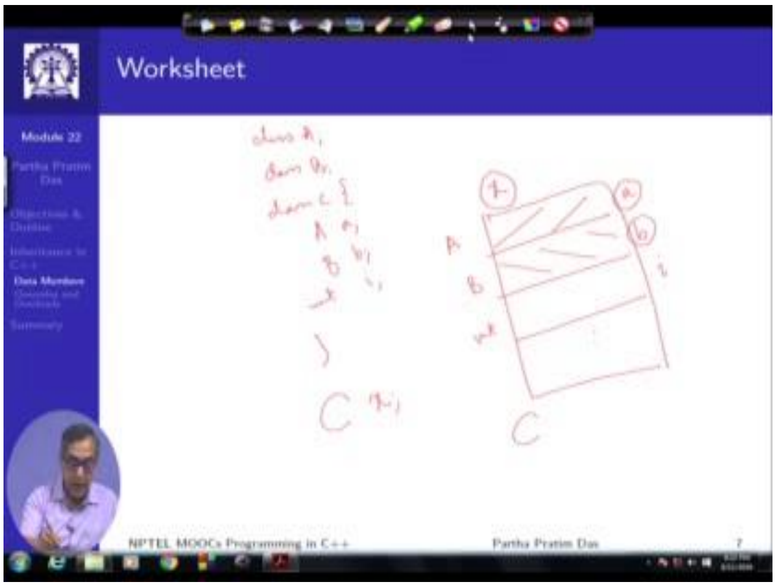
In this context, if I define if I instantiate an object B, then I would certainly have a layout for this, which would look like this is what we have discussed already. And in terms of a normal layout what we have discussed that if a class is instantiated, then this is the object and what is the basic properties of the layout. There are two basic properties that we had discussed one is all data members will be allocated space in the object in the memory in a contiguous manner, that is the whole of a object as a starting address and a size which is a number of bytes of memory that it occupies.

And all of the data members will occur within this space summer. It will never get fragmented that one part of the object is in one part of memory and other part is another part of the memory. So, contiguity is a basic requirement of layout that is we have already seen these, that is one. The second that we had seen is the order that is if we have two data members like this then one will come after. So, they will occur in the order in which they have been put down in the definition of the class this order will be maintained.

This is the basic structure of a layout. So, he given that if I look into the layout of b, this value except something like this. Now let us try to consider, what is the layout of D. Now, in D what happens specifically is certainly D will have its own data member which is which is find easy to understand. But since D is a B it will have an explicit instance of a B object as a part of the D object. So, as if the D object here, would have in one part a B object and when I am saying that D inherits this data member from B it means; that actually if I talk about D dot data one b, then this particular memory will be referred to. And what is this memory? This memory is a base object part of the derived object.

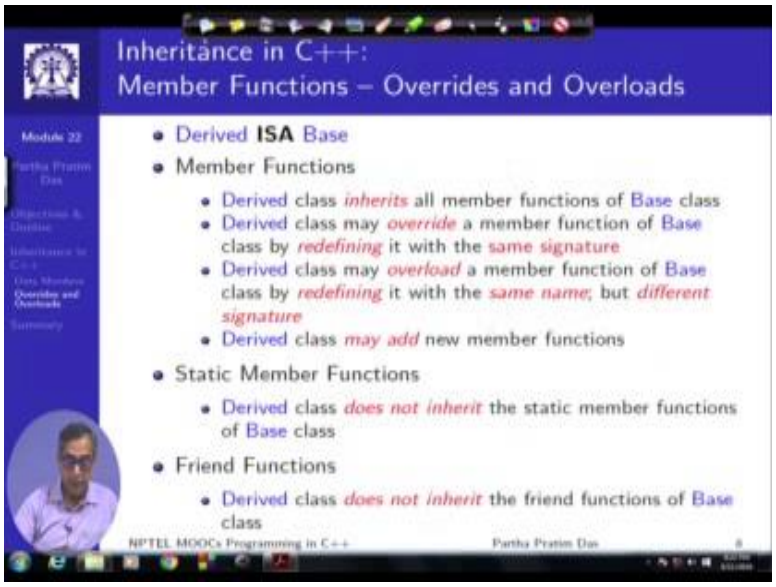
Similarly, if I refer to D dot data two B then I am referring this particular part of the memory, which has been inherited through the mechanism of inheritance and actually form the base part of the object.

(Refer Slide Time: 09:46)



So, you can see that there is something very interesting happening here, we had seen that earlier also we had seen; that lets say if I just take an example of different kinds of data members. Let us say if I just go here and let us let me take that I have a class A, I have a class B, I have a class C, which as an A object as a data member, which as a B object as a data member possibly and int and so on and so forth.

Then when I look into the instance of C, say x is an instance of C when I look at x, then I will have a object A, will have a b object B, will have an int and int i and so on and so forth. So, we have already seen that different objects of different classes could be part of the layout of another class. So, this is the class C and object x of class C, which have instances of different other class and that is the concept of component.



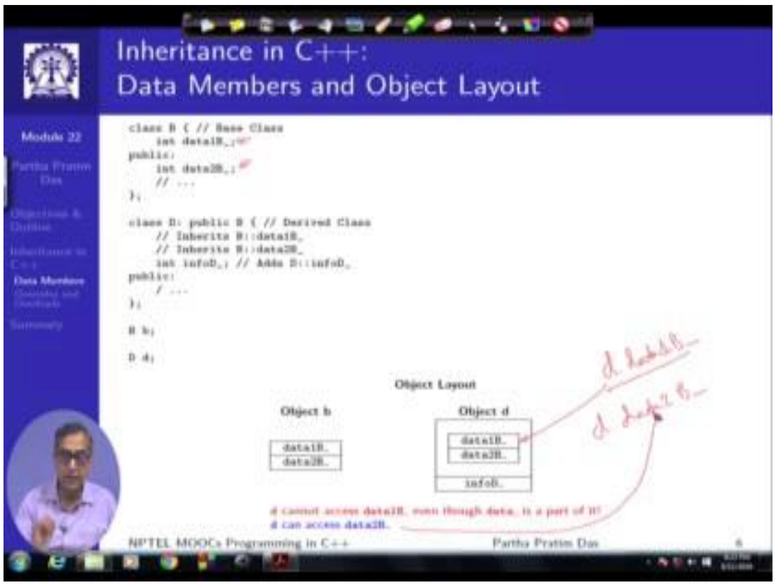
Inheritance in C++: Member Functions – Overrides and Overloads

- Derived **ISA** Base
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*, but *different signature*
  - Derived class *may add* new member functions
- Static Member Functions
  - Derived class *does not inherit* the static member functions of Base class
- Friend Functions
  - Derived class *does not inherit* the friend functions of Base class

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

But here as we note that I am sorry, here as we note that just one second, is change our create problems at times.

(Refer Slide Time: 11:06)



Inheritance in C++: Data Members and Object Layout

```
class B { // Base Class
    int data1B_;
public:
    int data2B_;
    // ...
};

class D: public B { // Derived Class
    // Inherits B::data1B_
    // Inherits B::data2B_
    int infoD_; // Adds D::infoD_
public:
    // ...
};

B b;

D d;
```

Object Layout

Object b
data1B_
data2B_

Object d
data1B_
data2B_
infoD_

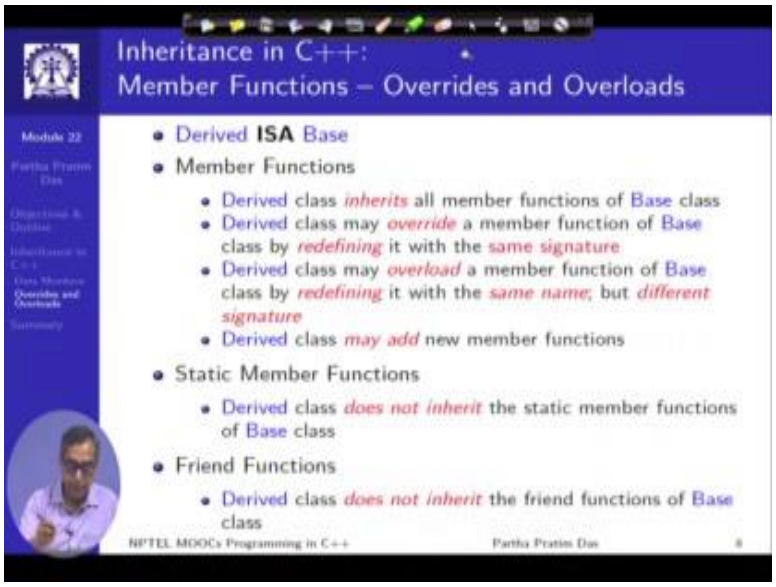
d cannot access data1B_ even though data_ is a part of it!
d can access data2B_

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

Here, as we note that since D is a B it also gets a component, but that component is very unique like I can have any number of data members, I may not have any data members. But when I inherit I will have one unique component coming in the derived class object, which is the base component and that is something that you will have to be very aware of in terms of the object layout.

There some additional noting have made here, which is a immediately not very important, but you can note that since this is a private member of the base class, private member cannot be access from outside. So, I cannot, though I can write d 1 B and that refers to this memory I cannot actually access it, that access is not because is not is a part of it, but the access is restricted. Whereas, if I talk about this other member which is public, I can I writing d dot data 2B, I can also access, that we will discuss about this access issues later on, but I just wanted to highlight this on the passing.

(Refer Slide Time: 12:25)



Inheritance in C++: Member Functions – Overrides and Overloads

Module 22

Partha Pratim Das

Objectives & Outline

Inheritance in C++

Data Members

Overrides and Overloads

Summary

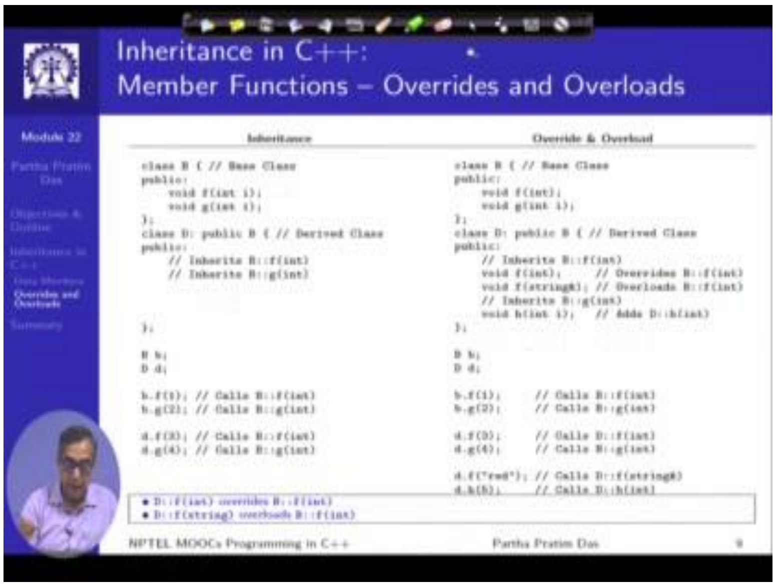
- Derived **ISA** Base
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by *redefining* it with the same signature
  - Derived class may *overload* a member function of Base class by *redefining* it with the *same name*; but *different signature*
  - Derived class *may add* new member functions
- Static Member Functions
  - Derived class *does not inherit* the static member functions of Base class
- Friend Functions
  - Derived class *does not inherit* the friend functions of Base class

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Now, let me move on to checking what happens with the member function with the member functions again it inherits all the member functions, but the important thing is after inheriting it can override a member function or it can overload a member function. So, we need to understand these concepts very, very carefully.

So, I illustrate that with example, but in the passing I would just like you to note, that the static member function or the friend member function cannot be inherited by the inheritance mechanism or in very simple terms. It is only the member functions, nonstatic member functions or only those functions, which have this pointer or which is actually associate with an object instance can only be inherited through this mechanism. So, we will see that for the static and friend function the meaning of inheritance would be quite different.

(Refer Slide Time: 13:38)



Inheritance in C++: Member Functions – Overrides and Overloads

Inheritance

```
class B { // Base Class
public:
    void f(int i);
    void g(int i);
};
class D: public B { // Derived Class
public:
    // Inherits B::f(int)
    // Inherits B::g(int)

};

B b;
D d;

b.f(1); // Calls B::f(int)
b.g(2); // Calls B::g(int)

d.f(3); // Calls B::f(int)
d.g(4); // Calls B::g(int)
```

Override & Overload

```
class B { // Base Class
public:
    void f(int);
    void g(int i);
};
class D: public B { // Derived Class
public:
    // Inherits B::f(int)
    void f(int);      // Overrides B::f(int)
    void f(string&); // Overloads B::f(int)
    // Inherits B::g(int)
    void h(int i);    // Adds D::h(int)
};

B b;
D d;

b.f(1);      // Calls B::f(int)
b.g(2);      // Calls B::g(int)

d.f(3);      // Calls D::f(int)
d.g(4);      // Calls B::g(int)

d.f("red"); // Calls D::f(string&)
d.h(5);      // Calls D::h(int)
```

- D::f(int) overrides B::f(int)
- D::f(string) overloads B::f(int)

NPTEL MOOCs Programming in C++ Partha Pratim Das 9

So, let us go to an example and try to understand what the member function inheritance mean. I would first want you to look at the left part of the code only do not look at this part, do not look at this part, this part as if it is not there. So, I have a class B, I have a class D the base and the derived the relive relationship is the same and for simplicity again in this case I am not looking at the data members I am just looking at the member functions where in public.

Certainly the issue of access is not a problem. Now I have a function f and a function g in class B, but I do not have any function in class D, class d is just an empty class. But still it will inherit the function f, it will inherit the function g and clarify one more notation here is this notation which we have mentioned earlier that a class is a namespace. So, if I am outside of this class then the name of this function is B colon colon f. So, by B colon colon f int I basically mean that this base class function, by this I mean this base class. So, what that I am saying is if you inherit then you will automatically have this base class functions available to you.

So, what does that mean, here again do not look at this, what does it mean that, if I have two instances b and d, and I try to invoke this functions f and g. Naturally if I do this is a straight forward we already understand that b dot f will invoke. This function b dot g will invoke this function there is no surprise in that.

But the interesting thing is I can actually invoke d dot f, which is not there in the definition of the class D of which small d is an object is an instance, but even then I can invoke d dot f, because d inherits from b and b as f function. So, when I invoke d dot f then I am actually invoking the f function of the base class. Similarly, if I do d dot g I am invoking the g function of the base class this is what is the core and flux of inheritance of member functions that you simply inherit all the member functions from your parent.

Now, let us forget about this part and look into the right side. Again, the same base class there is no difference I have not changed it. So, I the derive class it inherits f it inherit g as we have seen. In terms of g we again also I have not done anything, but in terms of f I have done something interesting; what I have done? That the (Refer Time: 16:28) signature or the prototype of f that was included in b as a member function have put that prototype again in d.

Now when I do this I say that I am overriding the earlier function, because the moment I put this I am talking about two functions, this functions is b colon colon f and this function is d colon colon f. When I did not have this function and still wanted to use it I was only meaning the base class function, but now I explicitly have another definition of the same member function by the same signature. So, let me clear up again. So, if I look in here again to instances of objects if I call f and g with b object certainly the effect is the same. If I call f with the d object then what will happen, earlier d did not have an f, but now it as overridden this f. So, if I now called d dot f it no more means this function it does not mean this it means this function, so it has changed into d colon colon f.

So, if you just compare with the previous one d dot f three was the f function of b now it is f function of d and this behaviour of member functions under inheritance is known as overload overriding. In overriding, whenever you talk of overriding it means that you are trying to attach two different definitions to a function by the same name and necessarily the same signature. You remember that for overloading the necessary part is the signature as to be different, signature in the sense of at least one parameter as to be different between two overloaded functions the name function name is same the return type does not matter, but the function parameters must differ some other signature must be different.

For overriding, it is that signature must be same, but they must belong to two different classes, which are related by a generalization, specialization, inheritance, relationship. And then depending on which particular object you are using you will end up calling different versions of this function. That is the reason that when we when you b dot f you call one function you call this function, but when you do d dot f now you call a different function the overridden function. This is the basic mechanism of overriding member functions, which add a lot of new value and semantics to the whole story.

Now at the same time if you look at the other function g in b that also is inherited by d, but that function as not been overridden, that is in d we have not included a fresh signature fresh signature of function g. So, if I in this context if I call d dot g then we will again actually be calling the function in b, because that is a function that is inherited. So, this behaviour d dot g four between the left and the right does not change, because g has not been inherited it has not been overridden after inheritance in the class D. So, this clearly explains as to what is the difference between inheriting impure inheritance and inheritance and overriding of a function. What adds a very nice flavour to the whole thing is I could also overload as function, which the base classes.

Now look into this two and this is this often confuses students and you know. In terms of what I am doing, am I overriding am, I overloading what am I doing. Here the distinguishing features are that the signature of the function is different between these two cases. So, you would do not perceive that it overriding you perceive as if the class D is introducing a new function just incidentally, which as the same name as a function that d already inherits from b.

So, if I invoke something like this d dot f again, but with the parameter which is a string C string, which can be taken as a string type in the stl as well. Then between this d dot f and between this d dot f that is between these two functions, the normal resolution of overloading will work. And depending on the type of the parameter this associates with this function and this associates with this function and this function is necessarily the overload of the other function f, which we inherited and subsequently override.

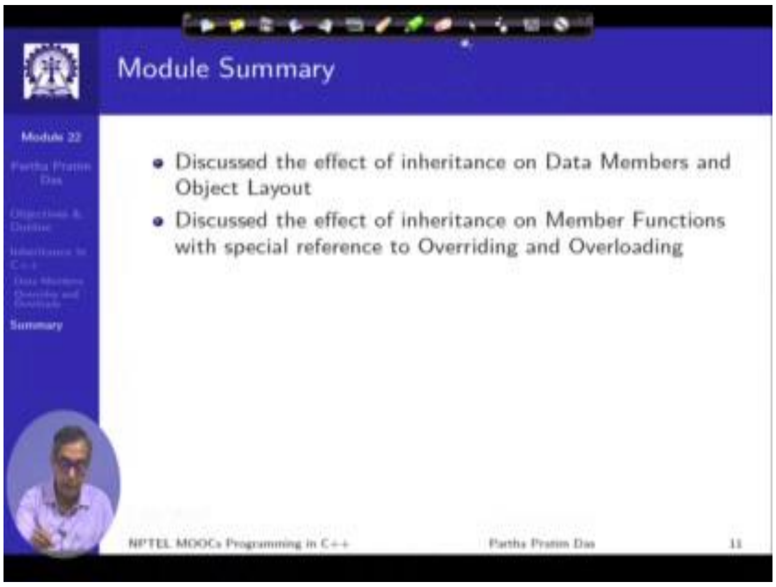
So, overloading in contrast to overriding would necessarily require that one both functions that are overloaded must belong to the same class, either directly or by inheritance and to they must differ in the signature that they have that is at least to one of the parameters have to be different between these two functions. And that is when we will say that we have a situation of overloading that is taking place.

Beyond all this certainly the class can introduce new functions, which means that a function with a new name, which has no name commonality with the member functions that the base class had neither f nor neither f nor g is an another function h, which the class D adds. And therefore, the class D can actually invoke that function on an object instance and certainly this function will get added.

As we see that in terms of member functions there are four possible dynamics that can happen, one is you could just inherit something as in g you could inherit and override something as in this f that is signature is same. But you are redefining, so possible in the implementation of the function that is this function will be implemented as B dot f int when you will add body to this. And the implementation of this function which will be D colon f add body to it these will possibly be different so, but signature is the same we have the second case which is overriding, the third is between two member functions you can continue to have overloading as we had earlier. In the same scope two functions by the same name must have different set of parameters to be overloaded and the forth is we can still add new member function.

So, the class and expand the functionality of the class. And this principles are kind of if my may so are recursive in nature. So, that d could also now become a base class for some another class and if d is a base class for another class, then all of this functions this overridden f, the overloaded f. The invisible g, which has been, which has come here through inheritance and h, all this four functions would be available to be inherited by any class that specialises from the class D.

(Refer Slide Time: 26:40)



Module Summary

- Discussed the effect of inheritance on Data Members and Object Layout
- Discussed the effect of inheritance on Member Functions with special reference to Overriding and Overloading

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

To summarize here, in this module continuing on the discussion on inheritance, we have discussed to very core concepts of inheritance relating to what happens to the data members when one class specialises from another, we have seen that. Under that situation, the derive class object will have an instance of the base class object as a part of it. And we have noted that in terms of the layout. It is not guaranteed as to, whether the base class object will come at the lower address and the data members of the derived class will occur at a higher address or vice versa or some other mix will be done.

And we have also seen that in terms of inheritance, the member functions are inherited from a base class to the derived class, but very interestingly after inheritance the member functions can be overridden. And in that context, the original rules of overloading also would continue to work.