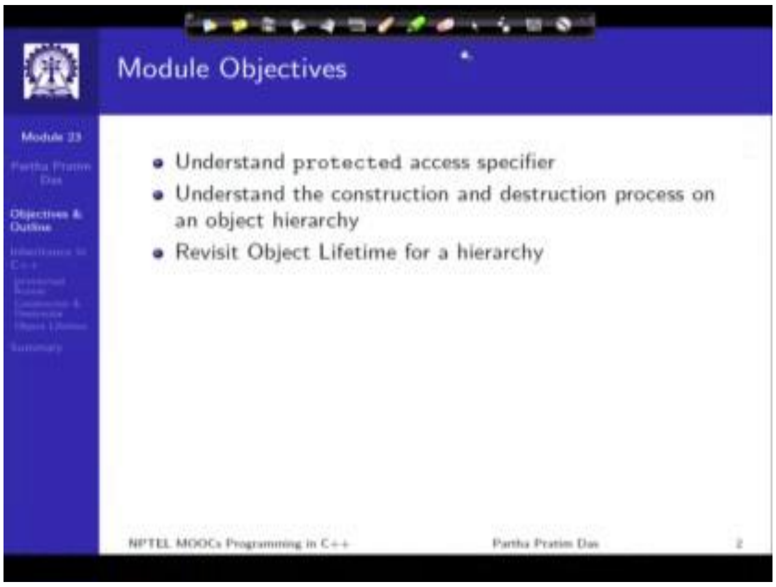
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 38
Inheritance: Part III

Welcome to Module 23 of Programming in C++. We are discussing about Inheritance in C++.

(Refer Slide Time: 00:26)



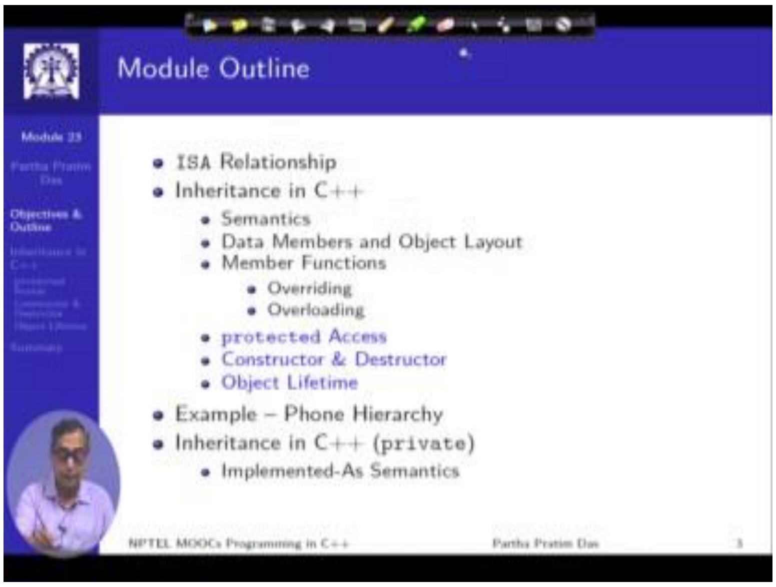
Module Objectives

- Understand `protected` access specifier
- Understand the construction and destruction process on an object hierarchy
- Revisit Object Lifetime for a hierarchy

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

We have talked about the basic requirement of inheritance and we have touched upon; we have discussed the very core two concepts of the affects of inheritance between two classes, that is how does the data members are inherited and abjectly out done and how does the member functions can be inherited with overriding and over loading playing a role. In the current module, we will try to talk about the protected access specifiers, the semantics of the access specifier and what happens to the construction destruction process and relate that to the issue of object lifetime in the context of specialization scenarios.

(Refer Slide Time: 01:24)



Module Outline

Module 23

Partha Pratim Das

Objectives & Outline

Inheritance in C++

protected Access

Constructor & Destructor

Object Lifetime

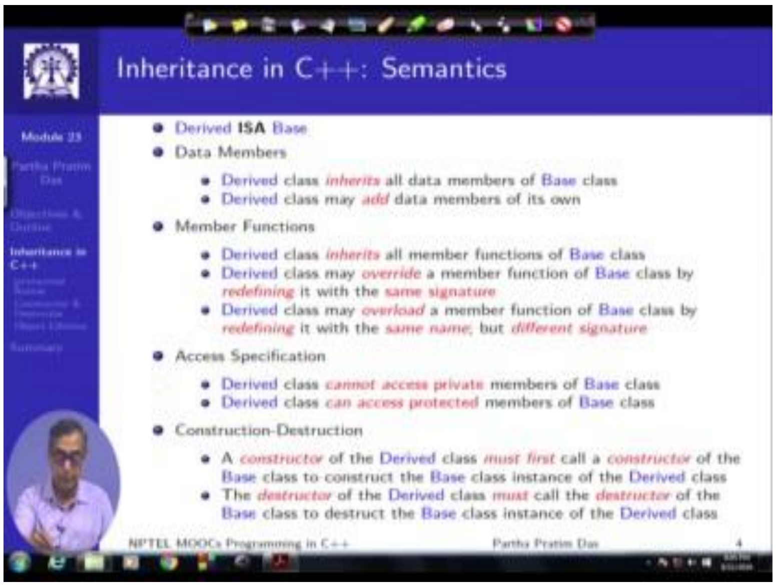
Summary

- 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

The outline as I have already explained is for the whole of this inheritance and the blue part relates to what we will be discussing in this current module.

(Refer Slide Time: 01:41)



Inheritance in C++: Semantics

Module 23

Partha Pratim Das

Objectives & Outline

Inheritance in C++

protected Access

Constructor & Destructor

Object Lifetime

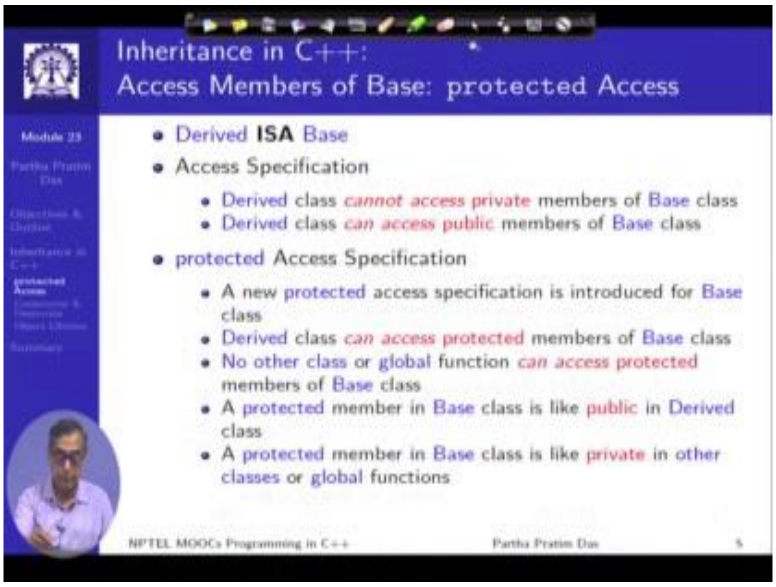
Summary

- 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

This is for your quick reference that, these are the broad semantic requirements and these two we have already done and this is the one that we will be covering in now.

(Refer Slide Time: 02:01)



Inheritance in C++: Access Members of Base: protected Access

- Derived **ISA** Base
- Access Specification
  - Derived class *cannot access* private members of Base class
  - Derived class *can access* public members of Base class
- protected Access Specification
  - A new protected access specification is introduced for Base class
  - Derived class *can access* protected members of Base class
  - No other class or global function *can access* protected members of Base class
  - A protected member in Base class is like public in Derived class
  - A protected member in Base class is like private in other classes or global functions

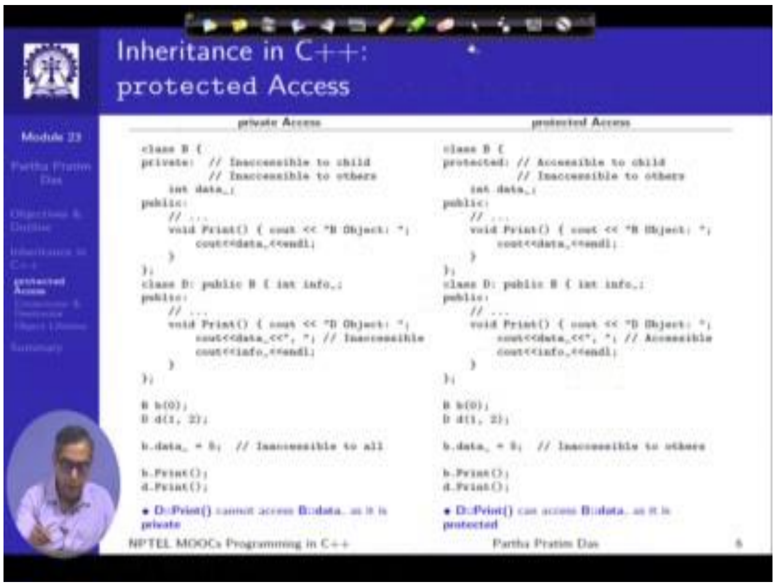
NPTEL MOOCs Programming in C++ Partha Pratim Das 5

Now, so what is the access specification issue? The access specification issue is as we know there are two kinds of access specifiers; the private and the public. So, for any class we have private members data as well as functions; member functions and we have public. So, this means the private members are available only to the class and the public members are available to all other classes, global functions and so on. So, with that if we have done a (Refer Time: 02:47) specialization, then the derived class, certainly cannot access the private members of the base class because the derived class is, by definition in another class. So, it is outside of the base class, it cannot access the private members. So, it can access only the public members.

Now, as you will find that the whole advantage that we want to take by modeling inheritance, by modeling the generalization specialization hierarchy, by introducing features of overriding member functions and so on will be lost, if the derived class cannot access any part of the base class, which the base class protects from everyone else. So, that has given rise to defining a new kind of access specifier known as protected.

So, the protected has a special semantics that a derived class can access the protected members of the base class. But, no other global function, no other class or global function can access the protected members. So, what it means that, if I look at a protected member of a base class, then for the derived class it is like a public access and for other classes and global functions it is like a private access. So, we are kind of trying to; the attempt has been to create a third kind of visibility or access specification beyond private and public, so that private is totally inside the class, public is totally for everybody and protected is something that is for the class, but for some of the classes with home, this class has a very close relationship of specialization generalization or parent child kind of.

(Refer Slide Time: 05:17)



Inheritance in C++: protected Access

Module 23

Partha Pratim Das

private Access

```
class B {
private:   // Inaccessible to child
           // Inaccessible to others
    int data_;
public:
    // ...
    void Print() { cout << "B Object: ";
        cout<<data_<<endl;
    }
};
class D: public B { int info_;
public:
    // ...
    void Print() { cout << "D Object: ";
        cout<<data_<<", "; // Inaccessible
        cout<<info_<<endl;
    }
};

B b(0);
D d(1, 2);

b.data_ = 5;  // Inaccessible to all

b.Print();
d.Print();
```

- D::Print() cannot access B::data_ as it is private

protected Access

```
class B {
protected: // Accessible to child
           // Inaccessible to others
    int data_;
public:
    // ...
    void Print() { cout << "B Object: ";
        cout<<data_<<endl;
    }
};
class D: public B { int info_;
public:
    // ...
    void Print() { cout << "D Object: ";
        cout<<data_<<", "; // Accessible
        cout<<info_<<endl;
    }
};

B b(0);
D d(1, 2);

b.data_ = 5;  // Inaccessible to others

b.Print();
d.Print();
```

- D::Print() can access B::data_ as it is protected

NPTEL MOOCs Programming in C++ Partha Pratim Das 6

So, let us look into example. Again, please look at the left side, do not look at the right. So, we have a class b, which has a private data member and a public method. The public method simply prints the data member. We have a derived Class D of B, which again has its own private data which is the additional data and it has a public method print. So, having understood the inheritance of member functions, we will realize that this is a case of overriding member function that is occurring here because both these print functions have the same signature.

So, D as it is inherits this, but then it overwrites and declares it again and possibly gives it a different semantics, whatever is the body here and the body here is different and this is what we have. And then what we want to do is to instantiate these object and invoke print from both these objects that is the basic target. Now, let us look into the access issues. This is; that the data member of Class B is private. Therefore it is inaccessible to all including the child class and including the derived class, it is not accessible. So, if I want to write a print function in Class D which prints the whole of Class D, how will the class D look like? Class D will look like a layout of this is.

We have already discussed this, there is a base part which has this data 1, there is derived part which is info and that is the class. So, if I want to print this class, then I actually have an int B colon colon data member in this class which is this member. So, to be able to print this, I must print this as well as the info which is the member of this class. The question is that how do I access this data? This data is a private member of this class B. So, if I make it explicit when I talk about this data, I am actually, if I fully qualify applying the name space rules, then I am actually talking about this element B colon colon data which has been inherited by the inheritance rule of specialization.

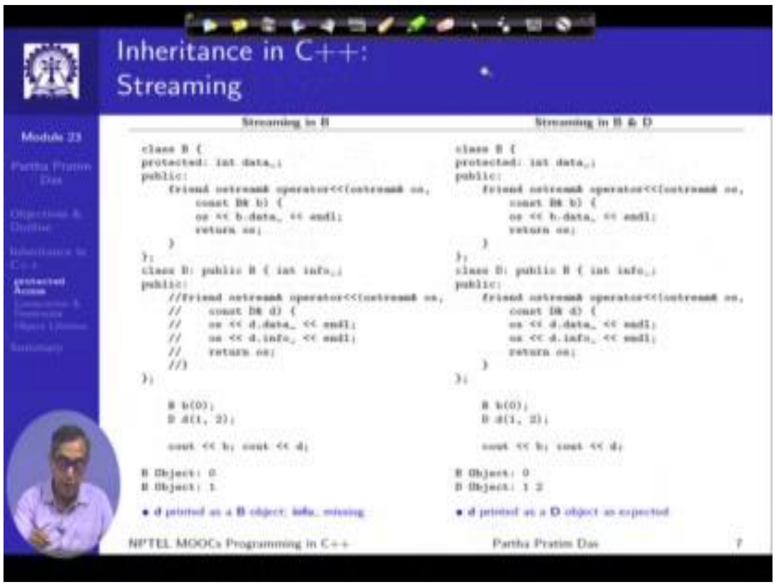
So, accessing this data is basically accessing this data which by rule is not accessible to anyone else. Therefore, this print function cannot compile, it cannot work. Certainly, if outside if I just write b dot data, it will not compile, it is not accessible for the outsiders. So, this is the core problem that the protected access specifier is trying to address. Now, if D cannot access this, then there is no fun of having this data member inherited in this part of this class. So, that will mean that every time I may inherit, the base class will have an instance as a part of this object but, that cannot be read or written by the derived class.

So, it is kind a dead part in the object layout. Now for that, if we make this public, if you move this here, then naturally this problem will get solved. Print will be able to compile this but then it breaks the encapsulation because anyone else will also be able to come and read and change the data. So, we need to provide a specification which is in between these two extremes and, that is where we now introduce a protected specifier. Earlier this was private, now this is protected, so, int data is protected. So, nothing else changes in this code, when now in the print member function of class D, we try to refer to data which again is B colon colon data, is this member.

This access is allowed because being protected is accessible to the child. But just look at this, this has not violated. the restriction on the other use this is written some were else in some other global function or in some other class, but that will not that access will not work because wherever I written this is not a part of a derive class of B. So, had we made this public then this would have work, but this would also have become accessible, but making this protected we had made sure that this will work, but this will continue to be a compilation error this access will not be allowable for here. So, that is the basic meaning of the protected access specifier.

Here, I am showing it in terms of data members, exactly the same semantic will work what members function as well. So, any derive class would be able to therefore, access both the public as well as protected members of the base class. So, now, with this we will be able to implement compile and also invoke the both print functions, one for this, this is simply for the B class members and this as we have over written and this is for the D class where the base part will first be printed and then the derive class part will also be printed. So, that is the basic notion of private access specifier and we will often find a good use of this happening.

(Refer Slide Time: 12:52)



Inheritance in C++: Streaming

Module 23

Partha Pratim Das

Objectives & Outline

Inheritance in C++

protected Access

Constructor & Destructor

Object Lifetime

Summary

Streaming in B

```
class B {
protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) {
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    //friend ostream& operator<<(ostream& os,
    //    const D& d) {
    //    os << d.data_ << endl;
    //    os << d.info_ << endl;
    //    return os;
    //}
};

    B b(0);
    D d(1, 2);

    cout << b; cout << d;

B Object: 0
B Object: 1
```

- d printed as a B object: info_ missing

Streaming in B & D

```
class B {
protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) {
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    friend ostream& operator<<(ostream& os,
        const D& d) {
        os << d.data_ << endl;
        os << d.info_ << endl;
        return os;
    }
};

    B b(0);
    D d(1, 2);

    cout << b; cout << d;

B Object: 0
D Object: 1 2
```

- d printed as a D object as expected

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

In this connection, let me show some more examples. So, here I am just showing the use of; further use of the members, here is the protected members. So, if you look into this class then you will find that; let me put it clearly. If you look into this class it has protected data member which is a shielded from outside, but certainly can be used by the derive class and this class has some friend function which is trying to print that class.

Now, interestingly this being a friend function is as it is not available to class D because this function as we know actually is a global function it is just a friend. So, that it can have the visibility, because this a friend function it has the access to private data members of the class B, but it will also that access would also be extended to the protected data members of class B because being a friend this overloaded operator function has complete access within the inside of the class B. Now, if you and assuming that you do not have you are not written say this operator function in class D and you have instantiated the two objects here, naturally B needs one data for this data part, D objects needs two, this will become the data; this is a base class one and this will become the info.

Now, if you try to do this, what will be your expectation? Certainly your expectation would be that if I do c out b this is the invocation of this function. That is the c out b is basically operator output string, first member c out second member b. So, this should invoke that operator and print the B object which does connecting, but the interesting fact is that what happens, if you also write this. What is this? This is operator D, this d is of type class D and not of this type. So, you would expect your first reaction would be there should be a compilation here, but interestingly it is not. It is not a compilation here this will also go through, what it will do is and that is very, very risky and that is the reason I bought up this an example is the moment you start using protected and start using friend function you will have this.

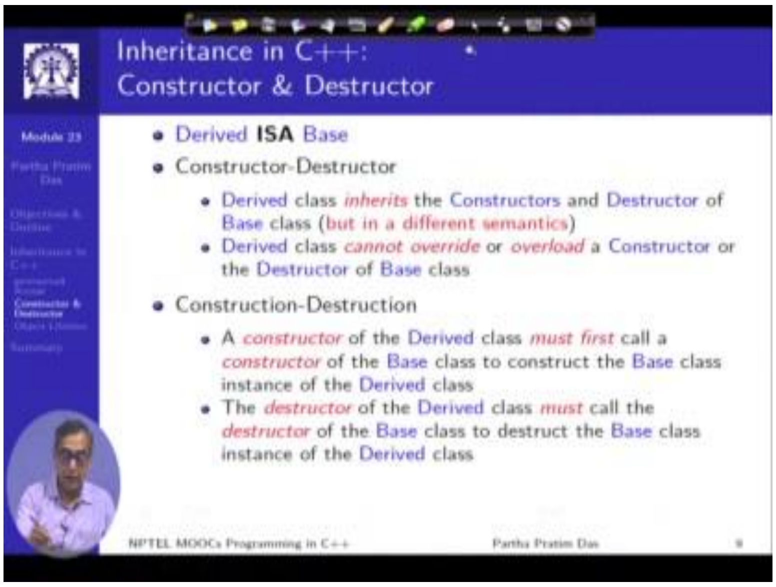
What it will do, it will invoke this function itself, but it needs b, but it has d. How will it work, recall the layout, the layout is d object has two parts; one is the member it has defined which is info, other is an instance of the base class object. So, what is happening here, you require a bass class object at this point and you actually have a derive class object in hand, but a derive class object also has the base class part. So, what the system does, system simply ignores the additional part. It simply takes that as if this is the base class object which you wanted to pass and lets this function be called. That is a reason just a look at the result very carefully this gets printed b object value 0 gets printed from this c out, which is correct because I created object b with the value of data which is 0; clear enough.

This also prints; that is the b object and a value is 1 which is basically, this value 1 is basically the value of the B colon colon data inherited member that we are setting at this point and the two which actually got set to info will not certainly not get printed because this function does not even is aware of the existence of this particular data member. This whole phenomenon happens because of the something known as a implicit casting of which we will discuss about much later, but I just wanted to give you this glimpse of this example to sensitize you that in terms of the; in terms of actually over loading friend functions and so on. Since, friend functions take the object as a parameter and not invoke on the object like other member functions do, you do run the risk of implicit type casting and you have to be careful to protect against that.

Naturally, the solution, the problem will become alright if you just have the overloaded operator function here as well as another overloaded operator function here where you can certainly use data because the data is protected member in b. So, d can inherit as well as access this data and in that context for this object if you try to do c out this will go here, this will go here and now this is happening not by overriding this is happening because of the some to overloading because friend functions are global functions, they are not member functions there is not context of over writing here is this two function define in two different scopes are being resolved by the difference in their signature and both of them will correctly now you will have a d object which prints 1 and 2.

So, that is the basic kind of behavior that you will expect when you start exposing the inner of the object in terms of the protected data members or protected member functions.

(Refer Slide Time: 20:38)



Inheritance in C++: Constructor & Destructor

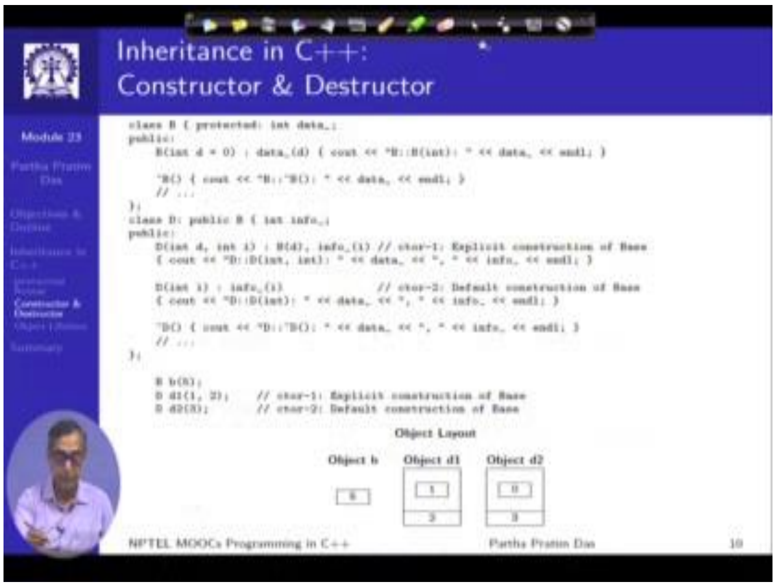
- Derived **ISA** Base
- Constructor-Destructor
    - Derived class *inherits* the Constructors and Destructor of Base class (but in a different semantics)
    - Derived class *cannot override* or *overload* a Constructor or the Destructor of Base class
- Construction-Destruction
    - A *constructor* 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 9

Let us move on and quickly talk on the constructor destructor. Now, when the derive class inherits the constructor and destructor of base class, but you have to very cautious it is with the different semantics. See, if I say, it inherits then the problem is a constructor is a fix name function; a constructor of a class has a same name as a class. So, if I inherit, if the bass class constructor into the derive class then certainly it cannot behave as a constructor. So, the base constructor becomes kind of a member function in the derive class constructor, in the derive class.

Similarly, base class destructor becomes kind of a member function in the derive class and they are, but it is very important that this inheritance of the constructor are possible because again recall that any derive class object has a base class instance. This is an instance of the base. So, if I have to construct the derive class object as a whole certainly I will have to construct the base class instance part of it and that needs the base class constructor to be available. Similarly, if I will construct then I need to destruct. So, by symmetry I will require the other.

(Refer Slide Time: 22:12)



Inheritance in C++: Constructor & Destructor

```
class B { protected: int data_;
public:
    B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }

    ~B() { cout << "B::~B(): " << data_ << endl; }
    // ...
};
class D: public B { int info_;
public:
    D(int d, int i) : B(d), info_(i) // ctor-1: Explicit construction of Base
    { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }

    D(int i) : info_(i)              // ctor-2: Default construction of Base
    { cout << "D::D(int): " << data_ << ", " << info_ << endl; }

    ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
    // ...
};

    B b(5);
    D d1(1, 2);    // ctor-1: Explicit construction of Base
    D d2(3);       // ctor-2: Default construction of Base
```

Object Layout

Object b	Object d1	Object d2
5	1	0
	2	3

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

So, this quickly shows what is possible, we have a base class constructor here, which the data pattern have not changed, the base class constructor there is a base class destructor, there is a base. This destructor does not do anything, simply prints a message and base class constructor is overloaded. It is a parameterized constructor and also a default constructor. So, basically there are two constructors here. For the derive class, we have added one data member. So, there are two values to set. So, what we do, we actually have provided two different constructors, one that takes two parameters and the other that takes one parameter.

So, what is interesting and new in this whole scenario is this piece of the code in the initialization list. The derive class object that needs to get constructed will need to have a base part instance of the base object. So, this needs to get constructed. Now, the basic principle is that you cannot proceed with the construction of the derive class components. There could be several components here, until you have finished with the construction of the base instance because base instance in a derive class object is unique. It may have 0 or 1 or any number of data members, but if in the under inheritance you can have only one base class instance in the object. So, that must get constructed.

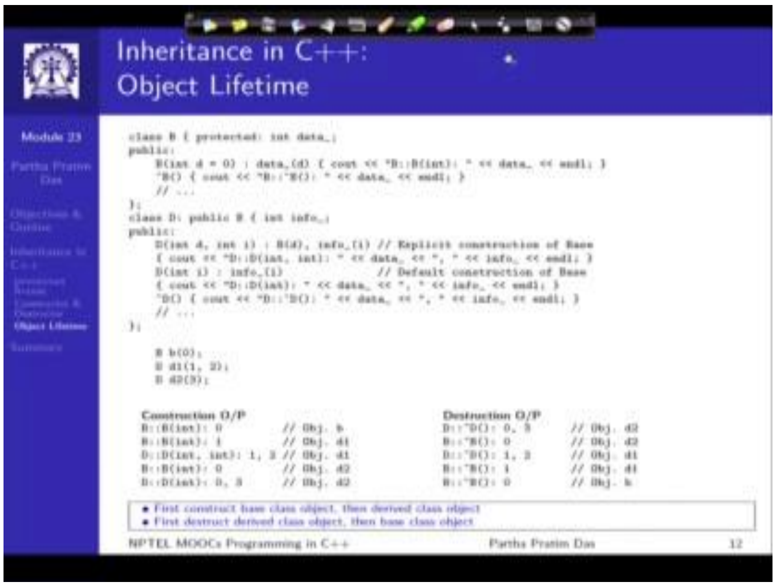
Here, what the base class construct, sorry, the derive class constructor is doing? The derive class constructor is actually calling the base class constructor, this is the name. So, this function is inherited here and the derive class constructor is actually calling this function by name passing the parameter that its needs which it once set in the data. So, the first this will happen this b data will happen and this will get set and then other initializations of its own members will happens.

What happens in this case? In this case there is no base class constructor being called that does not mean that the base class constructor will not need to be called. I have not put it explicitly here, but still at this point the base class constructor will get called because unless the base class part of the object has been constructed, the derive class object cannot get constructed. So, this will; like the compiler provides the free functions constructor destructor all of that. Similarly, this is where the compiler provides a free invocation of the base class a constructor.

Now, naturally if the compiler provides a free invocation then a compiler cannot set a parameter to the base class constructor. So, this is possible that is this kind of a constructor for derive class can be written provided the base class has a default constructor, which does not need a parameter. So, very simply if I just erase this default parameter in the base class constructor then this will become compilation error because it will say for this overloaded constructor of d, there is no default base class constructor, there is no base class constructor called. So, with this we can construct the objects, for example, if we construct is the d 1 object with 1, 2 then certainly this construct is getting used 1 get sets to d, then a base class construction happens that get set to here. So, if we do this then the eventual result will be that will 1 here and 2 here for the object d 1.

In case of object d 2 we are just passing one parameter. So, by overloading this means that this constructor will get invoke, which means that the base class constructor will get invoke with the default as a default constructor with the default value. So, this will become 0. So, in this case this part will become 0 and 3 will go to i which will come here which will go set here. So, that is all, the object will get constructed. So, here I have shown this layout. So, you could a take different examples and try your layout and get convinced that how this constructors work between themselves and similarly that base class will need to have destructor, the derive class will need to have destructor.

(Refer Slide Time: 27:44)



Inheritance in C++: Object Lifetime

Module 23

Partha Pratim Das

Objectives & Outline

Inheritance in C++

protected Access

Constructor & Destructor

Object Lifetime

Summary

```
class B { protected: int data_;
public:
    B(int d = 0) : data_(d) { cout << "B::B(int): " << data_ << endl; }
    ~B() { cout << "B::~B(): " << data_ << endl; }
    // ...
};
class D: public B { int info_;
public:
    D(int d, int i) : B(d), info_(i) // Explicit construction of Base
    { cout << "D::D(int, int): " << data_ << ", " << info_ << endl; }
    D(int i) : info_(i)              // Default construction of Base
    { cout << "D::D(int): " << data_ << ", " << info_ << endl; }
    ~D() { cout << "D::~D(): " << data_ << ", " << info_ << endl; }
    // ...
};

    B b(0);
    D d1(1, 3);
    D d2(3);
```

```
Construction O/P                            Destruction O/P
B::B(int): 0          // Obj. b             D::~D(): 0, 3      // Obj. d2
B::B(int): 1          // Obj. d1            B::~B(): 0         // Obj. d2
D::D(int, int): 1, 3 // Obj. d1             D::~D(): 1, 3      // Obj. d1
B::B(int): 0          // Obj. d2            B::~B(): 1         // Obj. d1
D::D(int): 0, 3       // Obj. d2            B::~B(): 0         // Obj. b
```

- First construct base class object, then derived class object
- First destruct derived class object, then base class object

NPTEL MOOCs Programming in C++ Partha Pratim Das 12

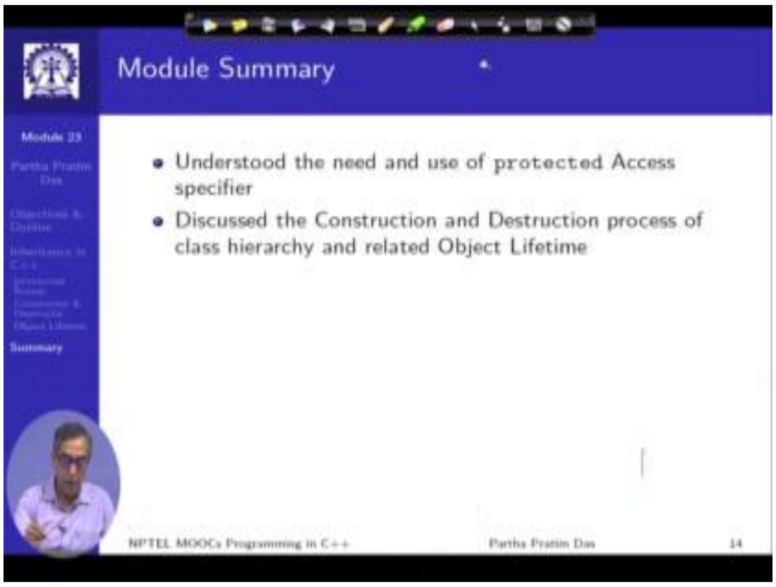
Now, with that if we just try to trace this messages that we have put and in the next slide we will try to do that. Let us say, if you just trace these messages and see then will be able to find out what is lifetime of the objects now. So, let us just try to do this, this is the same construction. So, naturally when this is encountered what has to happen? The base class constructor will have to get called for object being, that is the first thing that happens this message gets printed. Then this construction has to happen, the object d 1 of class D has to; if this has to happen then this constructor gets called, which in turn means that this constructor gets called. So, the next one you see is the construction of b object with the value 1 this is what here for the d 1 object that I have constructed.

Once that has been done then the construction of the d 1 object by the constructor of class D will get completed. So, I will get the next message, these two is the basically the construction of d 1. Next is; so, we have seen up to this point, next is the construction of object d 2 which invokes this constructor, which invokes the default constructor. So, I will again have constructor b called for object d 2, but the parameter value is 0. Once that has been done then the rest of the construction of this will continue and that will print the next message. You can see that the lifetime of an object continues to be same only with the exception that lifetime of the base object will start before the lifetime of the derive object can actually start. So, that will have to get constructed totally in full.

So, which means by symmetry of object lifetime that you have already seen, certainly if we look in to the remaining part of the messages as to what happens at the time of destruction that is when this object go out of scope at this point, certainly this was last object constructed. So, that with the first object you get destructed and what will happen this was last construction message. So, if you now look at the first destruction message is for the object d 2, that is this message. So, this destruction message is basically this one and only when the control reaches at this point, you first clean up whatever additional members that derive class object has and only when you have reached the end having done this, then this destructor of the base class gets called, when you clean up the base class.

So, you are first constructing the base class instance then constructing the remaining data members of the derive class. On the wrap-up time first clean up the non base data members of the derive class object and then clean up the base class part. So, if you look into try to trace for d 1, will see the same symmetric behavior and I would leave it to you to complete. So, this gives you the clear picture in terms of how the layout of the objects will be managed.

(Refer Slide Time: 31:25)


Module Summary

- Understood the need and use of protected Access specifier
- Discussed the Construction and Destruction process of class hierarchy and related Object Lifetime

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

So, to summarize on this module we have understood and really analyzed the need and use of protected access specifiers for specifying access to different base class members, and we have seen how the construction and destruction process happens between a derive class object and a base class object and really under try to follow trace the lifetime of the derive class object with respect to the base class object.