Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology Kharagpur
Lecture 23
Inheritance: Part 3
Constructor and Destructor - Object Lifetime

(Refer Slide Time: 00:33)

Programming in Modern C++

Module M23: Inheritance: Part 3: Constructor & Destructor: Object Lifetime

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Welcome to programming in modern C++. We are in week 5, and I am going to discuss module 23, this is on inheritance.

(Refer Slide Time: 00:39)

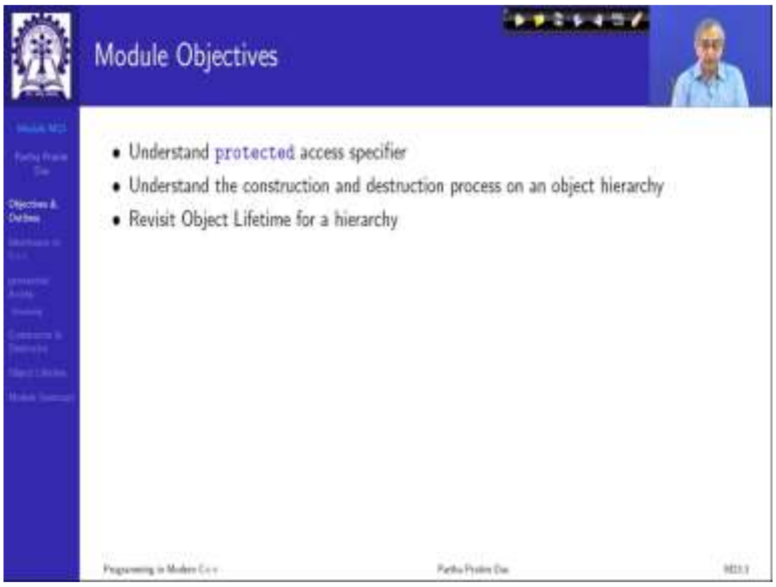
Module Recap

- 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

In the last two modules, we have introduced the concept of ISA relationship in object-oriented analysis and design. And we have introduced the basic semantics of inheritance particularly in terms of the data members of derived class and base class. So, the key take back has been that the derived class has all the data members of the base class, a complete instance of the base class in its object layout.

And we have also discussed about the inheritance of member functions, derived class inherits all member functions, non-static member functions of the base class, it can overwrite a member function, it can provide multiple overloads to a member function or it can add new member functions.

(Refer Slide Time: 01:42)



Module Objectives

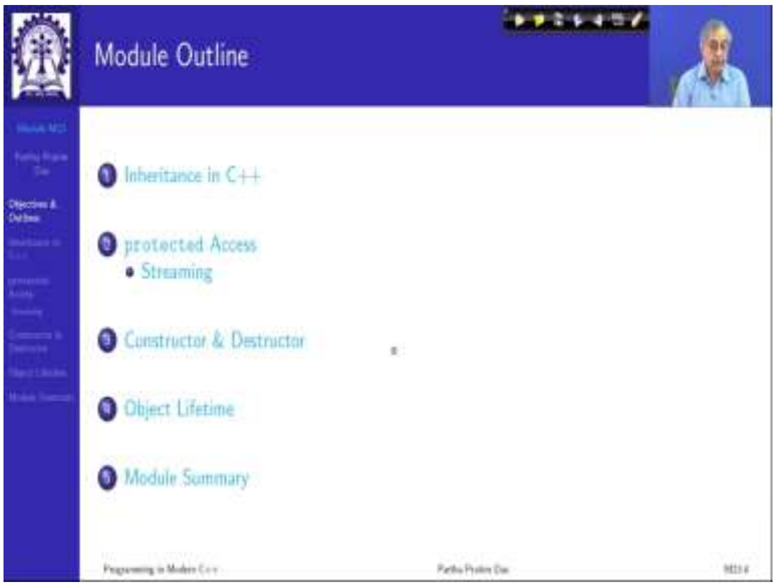
- Understand `protected` access specifier
- Understand the construction and destruction process on an object hierarchy
- Revisit Object Lifetime for a hierarchy

Objectives & Outlines

Programming in Modern C++ Partha Pratim Das M23.3

So, in this module, we will first introduce the concept of protected access specifier to kind of provide a solution that under the semantics of inheritance, the original objective of, the first objective of object-oriented analysis of design that is encapsulation is seeming like breaking down. So, we will have to restore by providing a new access specifier. And then, we will understand the construction and destruction process on a hierarchy and revisit the object lifetime concepts we had discussed earlier.

(Refer Slide Time: 02:31)



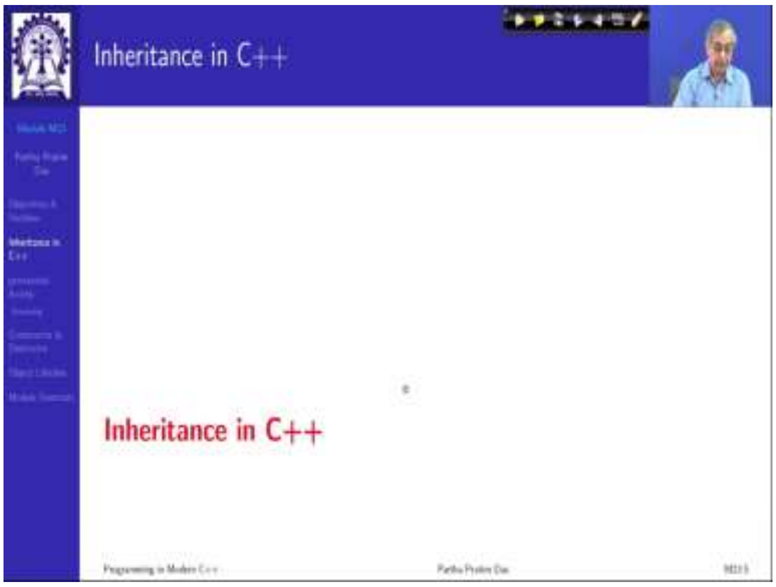
Module Outline

1. Inheritance in C++
2. protected Access
   - Streaming
3. Constructor & Destructor
4. Object Lifetime
5. Module Summary

Programming in Modern C++ Partha Pratim Das M23.4

So, this is the outline it is available on left.

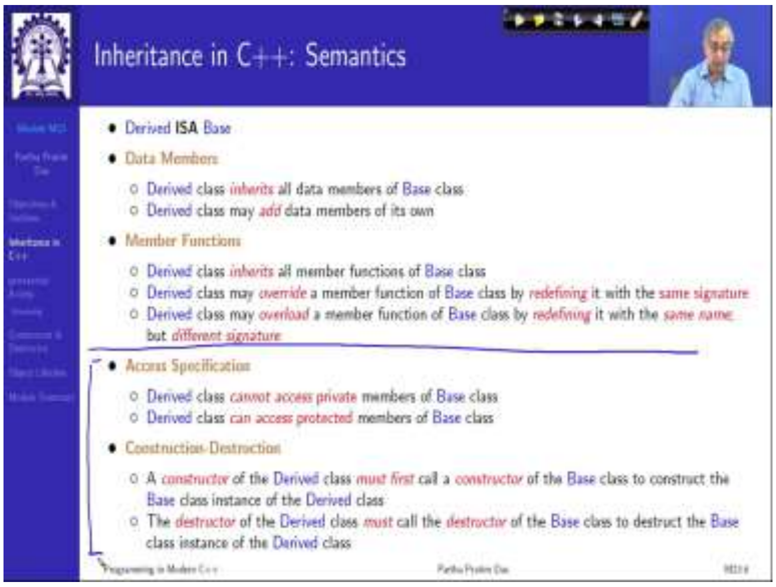
(Refer Slide Time: 02:36)



Inheritance in C++

Inheritance in C++

Programming in Modern C++ Partha Pratim Das M23.5



Inheritance in C++: Semantics

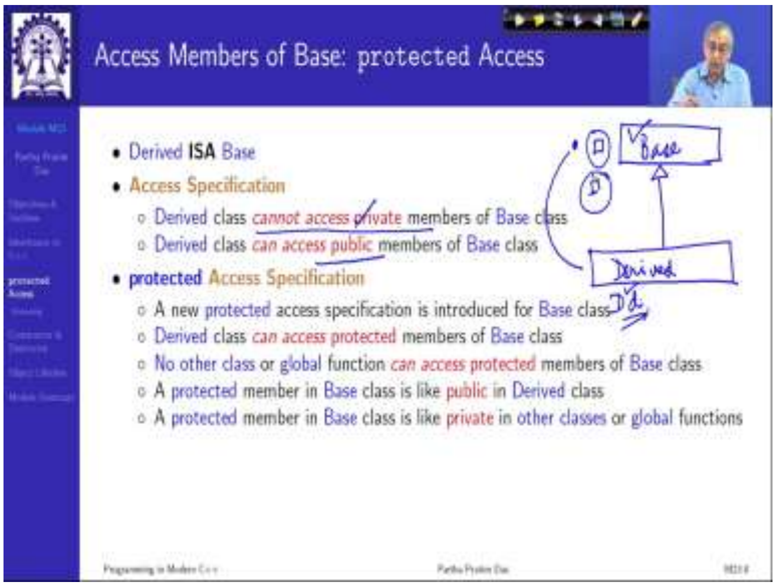
- Derived **ISA** Base
- Data Members
  - Derived class *inherits* all data members of Base class
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits* all member functions of Base class
  - Derived class may *override* a member function of Base class by 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

So, inheritance on C++. So, up to this point we have discussed. Now, we are going to discuss this part. Starting with access specification and then continuing in the object construction.

(Refer Slide Time: 02:56)

protected Access

protected **Access**


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

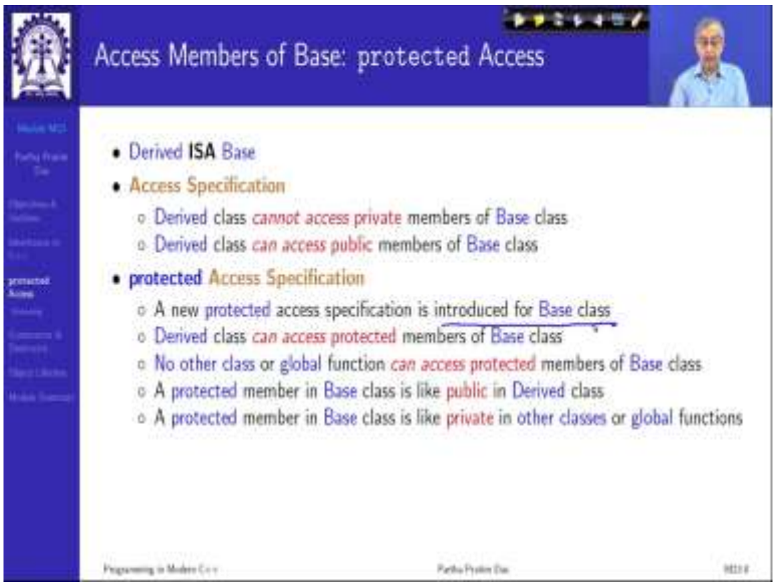
So, what is protected access? Derived is a base in this under the situation the derived class cannot access the private members of the base class. So, let us understand the requirement once more. So, I have the base class, I have the derived class, I have an ISA relationship between them. So, if I have a member here, I will have it in derived that is fine.

But the question is if I have a member which is private then the derived class object of the derived class or the derived class per se, will not be able to access it because it is private. It is encapsulated completely. It was required to encapsulate the original class base. So, we have to stick to it. If it is in public, if that member is in public, then the object of the derived class of the derived class member functions would be able to access it because everyone can access it.

So, this will lead to a new challenge in terms of encapsulation. If I keep the data member only in private, then it is of use by base the derived has to carry the instance but derived cannot do anything cannot exchange information with that cannot do any computation with that. If it is kept as public then the encapsulation is gone.

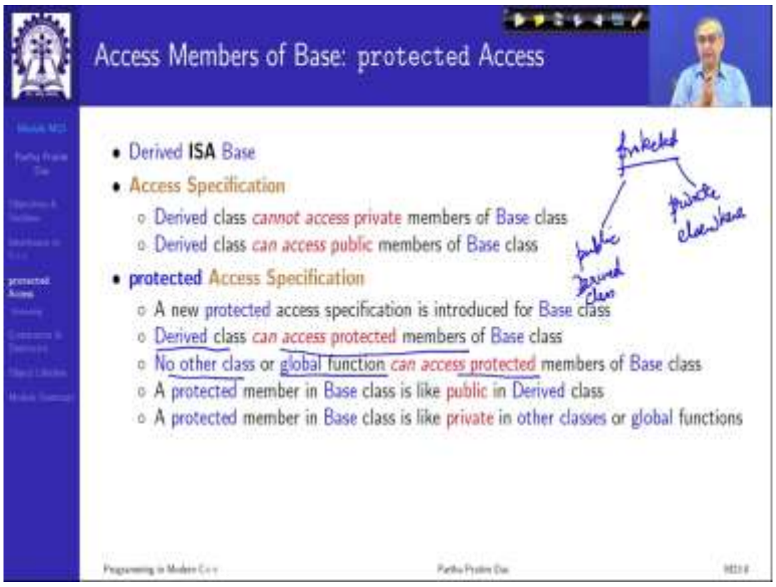
So, you can feel that the what has been provided in terms of private and public is good for a single class only. And using that and gets a idioms and all that you can have a very nice flexible model of information hiding, but that seems to break down when we have this kind of generalized specialized class structure.

(Refer Slide Time: 05:20)



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



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

protected

public
Derived
class

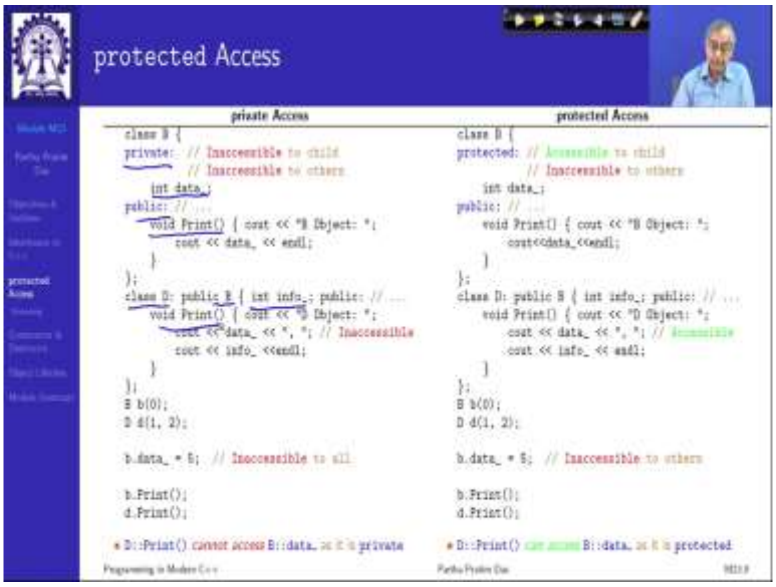
private
elsewhere

So, the protected access specifier is provided to solve this problem. So, it is to be introduced in the base class. Now, it has a very interesting semantics, the semantics is a derived class can access protected members of the base class. No other class or global function can access protected members. So, if I think about protected members then it is like public in the derived class and it is like private elsewhere anywhere else.

So, that is very, very, so this fine grained or finer grained access control access specification restores the information hiding property over object hierarchy. Because now, we can just use public and private within a single class for information hiding within that class on a and symbol of classes, which are related by ISA relationship by generalization specialization.

We can have three types of control to decide what is exclusively before use within the class which will go to private, what is the final interface for others which all will go into public and what is specific at different at stages of this hierarchy of ISA relationship which will go into the protected. So, that is, this is the basic understanding with which the protected access specifier we will have to look at.

(Refer Slide Time: 07:20)



protected Access

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

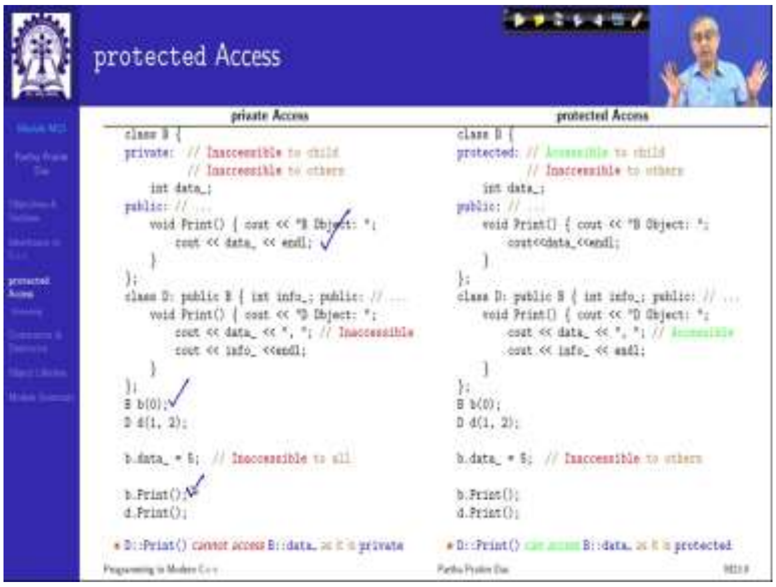
```
class B {
protected: // Accessible to child
           // Inaccessible to others
    int data_;
public: // ...
    void Print() { cout << "B Object: ";
        cout<<data_<<endl;
    }
};
class D: public B { int info_; public: // ...
    void Print() { cout << "D Object: ";
        cout << data_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

Programming in Modern C++ Partha Pratim Das M12.9

So, let me start with an example. Let us think about private access. So, I have private access, I have a data in that, I have a public access, I will print in that. Then class D is derived from class B and it has a private data info and it has, what it has done it is put to print again. So, it has overridden the print function of B, put to public ISA to see.

(Refer Slide Time: 08:05)



protected Access

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

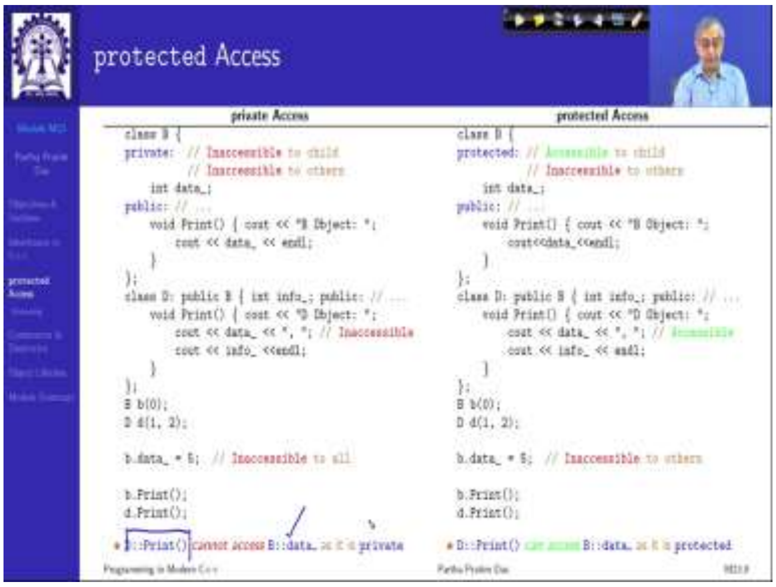
Programming in Modern C++ Partha Pratim Das M12.9

Now, this print will not have any difficulty because it is a single class scenario. So, if I construct an instance of this, I call b.print things are all fine. What happens if I construct an instance of D the derived class object and try to call the derived class function this function. Now, I can see to be able to print a derived class object it has to print both the base part as well as the derived data members that is derived class data member rather, derived class data members which it has added.

So, when it tries to print the base part, it tries to access data which is in the private of the base class B. So, it is inaccessible, it is inaccessible to child, it is inaccessible to D. So, this code simply will not compile. Either B will have to break the encapsulation by providing get/set function, if it does then it is available to others as well not only to class D it will be available to anyone else.

So, this is the problem that happens when I do d.print and this because of this lack of access this code will not compile. And obviously, if I directly want to use b dot data for everybody, this is inaccessible and this will not compile.

(Refer Slide Time: 09:49)



protected Access

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

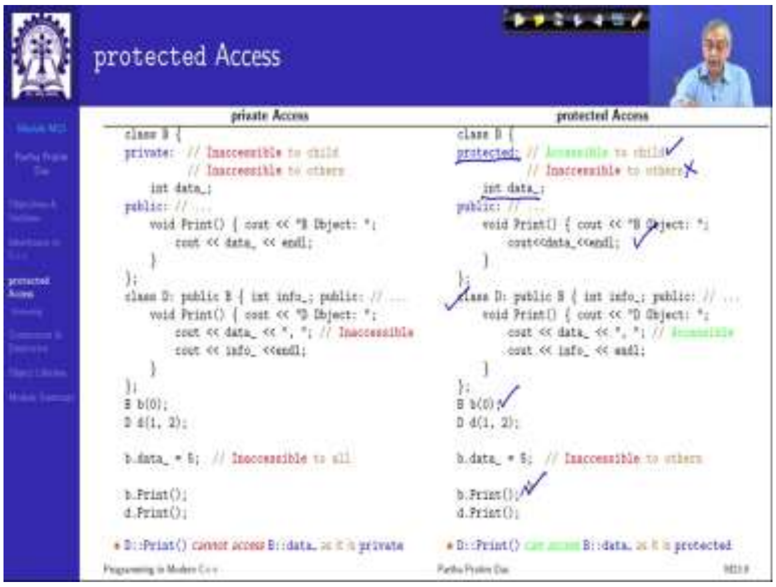
```
class B {
protected: // Accessible to child
           // Inaccessible to others
    int data_;
public: // ...
    void Print() { cout << "B Object: ";
        cout<<data_<<endl;
    }
};
class D: public B { int info_; public: // ...
    void Print() { cout << "D Object: ";
        cout << data_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

Programming in Modern C++ Partha Pratim Das M23.9

Conclusion. D::print this one, D::print cannot access B::colon data as it is private.

(Refer Slide Time: 10:01)



protected Access

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();
```

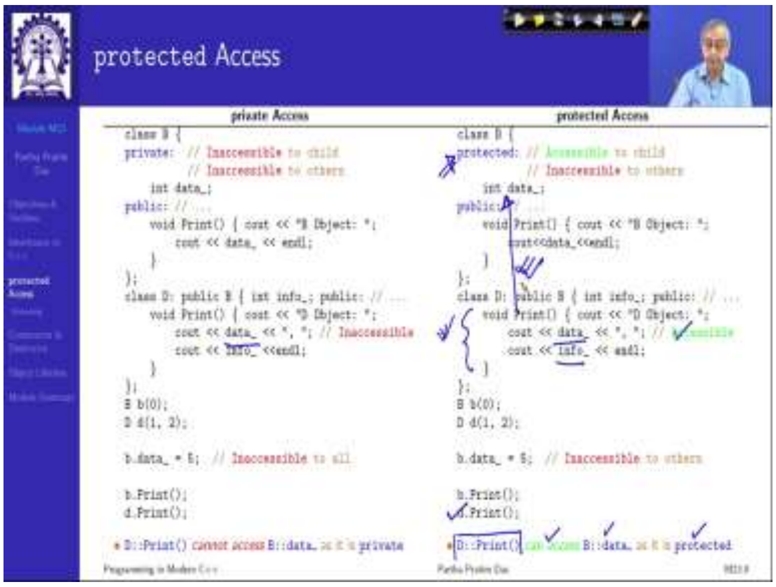
- D::Print() cannot access B::data_ as it is private

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

Programming in Modern C++ Partha Pratim Das M23.9



protected Access

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();
```

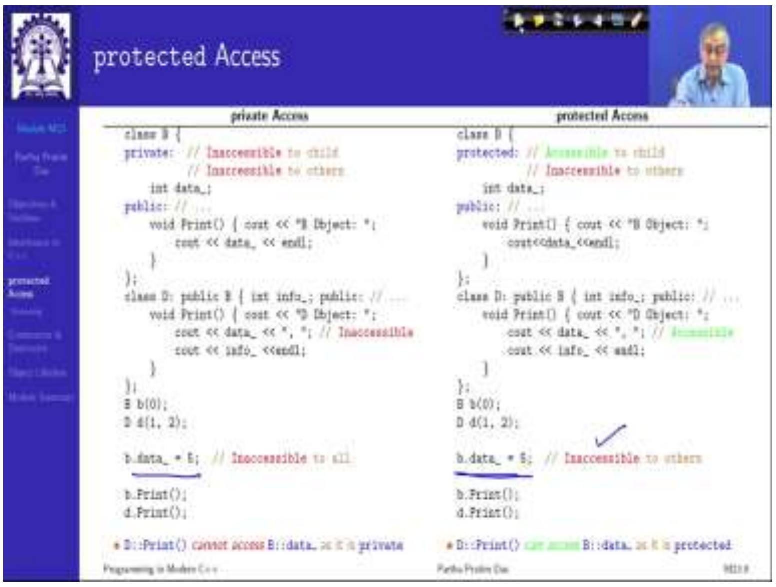
- D::Print() can access B::data_ as it is protected

Programming in Modern C++ Partha Pratim Das M23.9

Let us now try to solve this problem. We put this in protected access specification. This means that it is accessible to child it is accessible to D, but it is not accessible to others like when it was in the public, no it was in the private. So, this print function has no issue object B, b.print these are good as it were.

What happens in the overridden function print in class D is the data of B which was earlier inaccessible is now accessible because it is protected. So, this code will compile perfectly fine. And info of course is his own data, so it can access. So, now this also will work fine. So, D::print can access this data B::data, this can access, this access is possible because it is in the protected access specification.

(Refer Slide Time: 11:23)



protected Access

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();
```

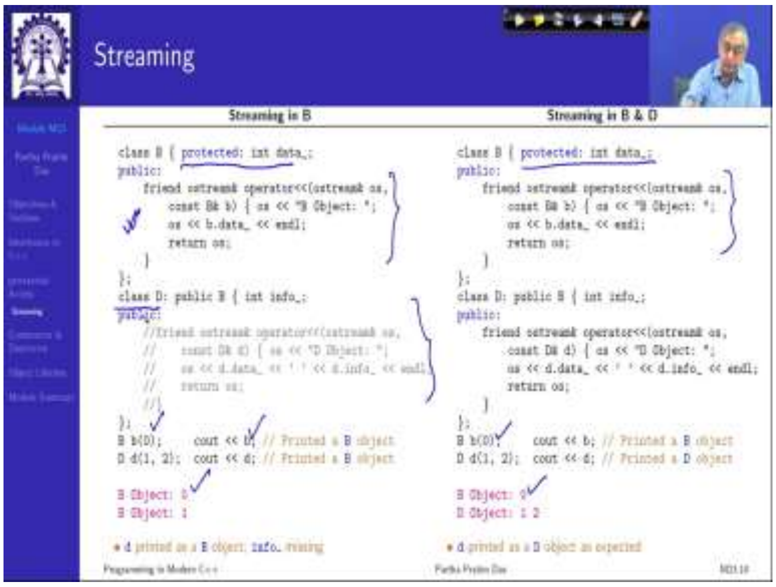
- D::Print() can access B::data_ as it is protected

Programming in Modern C++ Partha Pratim Das M21.8

If you come to b.data, earlier it was private. So, nobody could use it. If you look at b.data again, it is inaccessible to everyone. So, here, what you have achieved is you have not changed your encapsulation to the external world. You have not changed your access within the class. But you have kind of created a privilege, privilege type of rights between the base class and derived class.

They certainly true I am not obviously, if I talk about human inheritance, certainly I will have rights which are special specific to my parents, which others will not have. This is beyond the rights that my parents will have on their own on which even I cannot intrude upon, it is just that concept being put in the semantics of C++. So, this is mine protected access mechanism.

(Refer Slide Time: 12:33)



Streaming

Streaming in B

```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    //friend ostream& operator<<(ostream& os,
    //    const D& d) { os << "D Object: ";
    //    os << d.data_ << ' ' << d.info_ << endl;
    //    return os;
    //}
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a B object

B Object: 0
B Object: 1
```

- d printed as a B object: info_ missing

Streaming in B & D

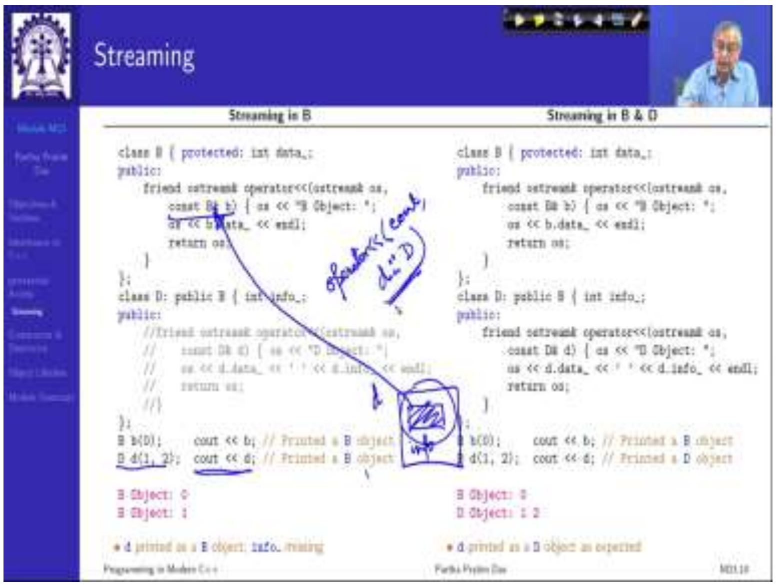
```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    friend ostream& operator<<(ostream& os,
        const D& d) { os << "D Object: ";
        os << d.data_ << ' ' << d.info_ << endl;
        return os;
    }
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a D object

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

- d printed as a D object as expected

Programming in Modern C++ Partha Pratim Das M21.10

We can see some more of the consequences that we have created particularly this is with the protected data in B what if I write a friend function. Earlier what we did was a member function, the friend function will be able to print so, I have b instance I do see out b it will print B object colon 0, same thing would happen here no difference. Now, suppose if I look at the class derived class D. Now, do I need to provide this operator overload again? The answer is yes. Because since this is a friend function, this will not be inherited by the derived class D.

(Refer Slide Time: 13:49)



Streaming

Streaming in B

```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    //friend ostream& operator<<(ostream& os,
    //    const D& d) { os << "D Object: ";
    //    os << d.data_ << ' ' << d.info_ << endl;
    //    return os;
    //}
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a B object

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 Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    friend ostream& operator<<(ostream& os,
        const D& d) { os << "D Object: ";
        os << d.data_ << ' ' << d.info_ << endl;
        return os;
    }
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a D object

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

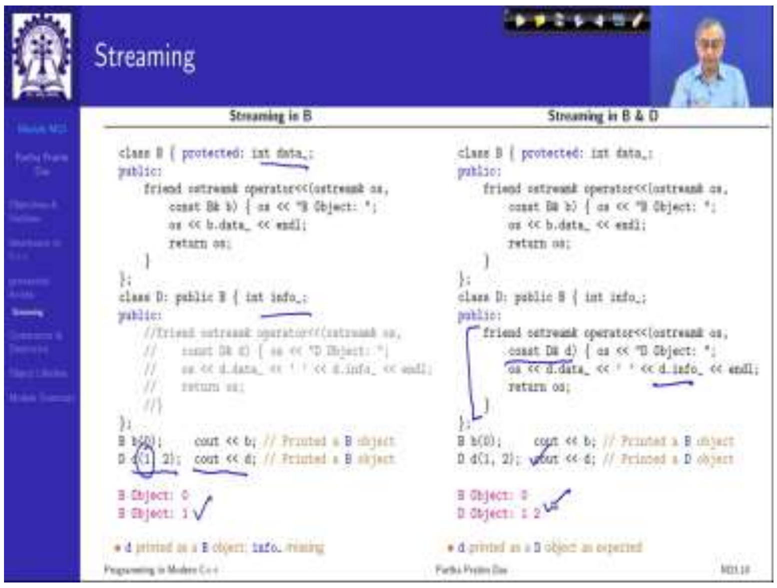
- d printed as a D object as expected

Programming in Modern C++ Partha Pratim Das M21.10

So, if I try to construct a D object and try to do this then what I get is I get a operator with, what do I have cout which is fine, we know string and then I have d which is of type D, but what do I expect is a b. Now here, the compiler does something interesting. The compiler finds figures out that well I need a B object. What you have given me is a D object which is actually bigger.

So, I have this D object and within that I have a base part of the D object. So, and then I have info here. The compiler knows that it has a B object part embedded here. So, what it does simply takes this part and uses it as the B object instead of taking the whole of D, of course, it does not even know the whole of D.

(Refer Slide Time: 15:18)



Streaming

Streaming in B

```
class B { protected: int data_;
public:
    friend ostream& operator<<(ostream& os,
        const B& b) { os << "B Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    //friend ostream& operator<<(ostream& os,
    //    const D& d) { os << "D Object: ";
    //    os << d.data_ << ' ' << d.info_ << endl;
    //    return os;
    //}
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a B object

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 Object: ";
        os << b.data_ << endl;
        return os;
    }
};
class D: public B { int info_;
public:
    friend ostream& operator<<(ostream& os,
        const D& d) { os << "D Object: ";
        os << d.data_ << ' ' << d.info_ << endl;
        return os;
    }
};
B b(0);     cout << b; // Printed a B object
D d(1, 2);  cout << d; // Printed a D object

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

- d printed as a D object as expected

Programming in Modern C++ Partha Pratim Das M21.18

So, what happens as a consequence of that, when you try to do this, you have two data members, one in the base class is data and other in the derived class is info, only the base class part one is printed. But if you provide a separate friend function to overload operator output streaming with a D class parameter and naturally printing the D info then this will print.

So, this is just to show you before we get into the constructor-destructor which are, these just to show you that one is how you carry on the information hiding encapsulation through protected. And when you break that, how does it work with the protected? It works in the same way as it were with the private, there is no special consequences, but you will have to remember the fact that breaking encapsulation by friend is not inherited.

So, it has to be, it is just not going to solve that you have allowed access by giving protected, but you will have to follow your other semantic rules of read I mean providing a separate friend overload for the operator that you want.

(Refer Slide Time: 16:56)

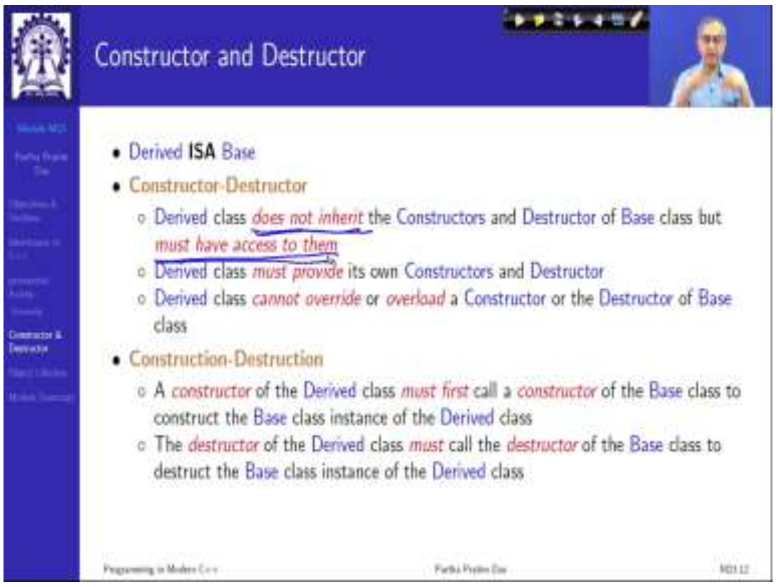
Constructor and Destructor

Constructor and Destructor

Programming in Modern C++ Partha Pratim Das M23.12

So, let me move on to the core of the object cite, the constructor and destructor.

(Refer Slide Time: 17:00)



Constructor and Destructor

- Derived **ISA** Base
- Constructor-Destructor
  - Derived class *does not inherit* the Constructors and Destructor of Base class but *must have access to them*
  - Derived class *must provide* its own Constructors and Destructor
  - 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

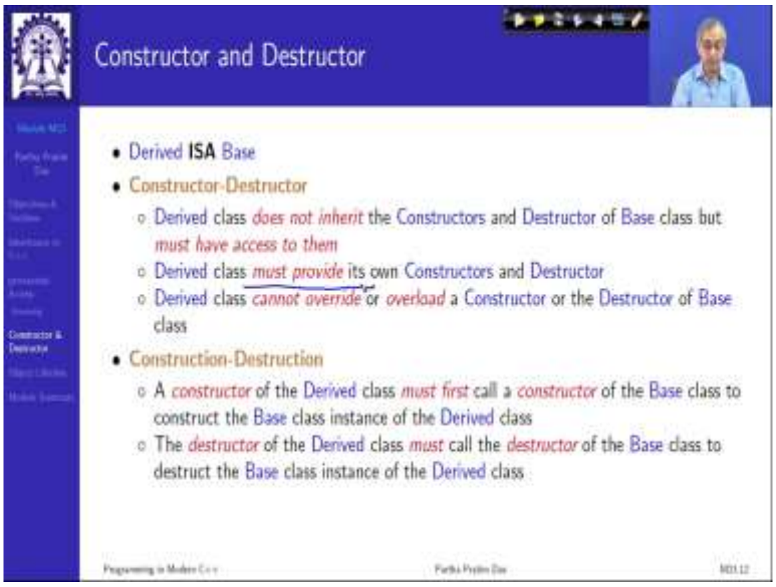
Programming in Modern C++ Partha Pratim Das M23.12

Now, what are the things that I have? I will say that the derived class does not inherit the constructor or destructor of the base class. Please do not get me wrong and maybe some literature would say that it does inherit, but inheritance having mere access is not inheritance, you may inherit but may not have access you may not inherit but may have access. So, you have to distinguish between them.

In case of constructor destructor, you do not inherit them because they do not become your member functions. First of all, constructor is static. So, it cannot be. But more importantly both constructor and destructor named by the class. So, base classes constructor destructor has no semantic meaning in the derived class. So, you do not inherit them, but you must have access to them. Why do you have access to them?

Because in object layout in the derived class, a base class part a base class instance is getting created. Naturally the base class does not, the derived class does not know how to create the base class object. So, it has to use a constructor for them, same for the destructor. So, this is the this is a very one thing.

(Refer Slide Time: 18:19)



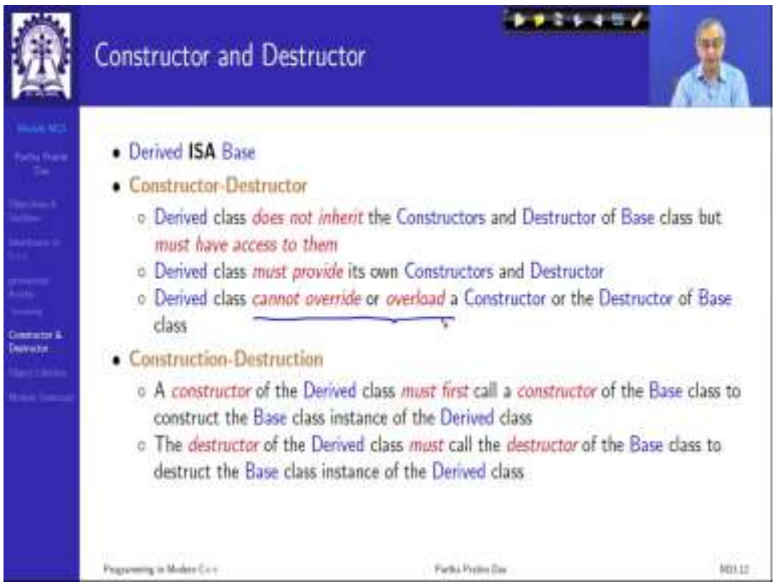
Constructor and Destructor

- Derived **ISA** Base
- Constructor-Destructor
  - Derived class *does not inherit* the Constructors and Destructor of Base class but *must have access to them*
  - Derived class *must provide* its own Constructors and Destructor
  - 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

Programming in Modern C++ Partha Pratim Das M21.12

In addition, naturally, the base derived class must provide its own constructor and destructor. There is no moving away from that.

(Refer Slide Time: 18:29)



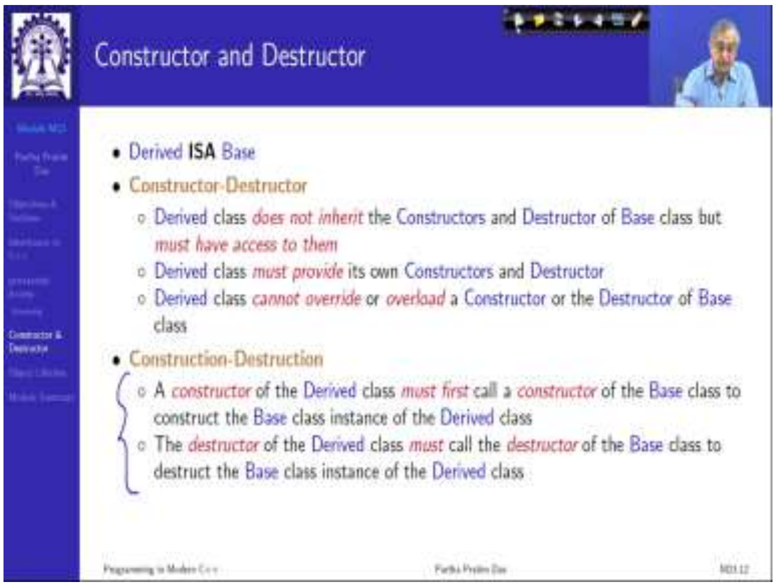
Constructor and Destructor

- Derived **ISA** Base
- Constructor-Destructor
  - Derived class *does not inherit* the Constructors and Destructor of Base class but *must have access to them*
  - Derived class *must provide* its own Constructors and Destructor
  - 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

Programming in Modern C++ Partha Pratim Das M21.12

And as we have noted, derived class cannot overwrite or overload the constructor of the district. It cannot overwrite neither cannot overload constructor or destructor of the base class.

(Refer Slide Time: 18:48)



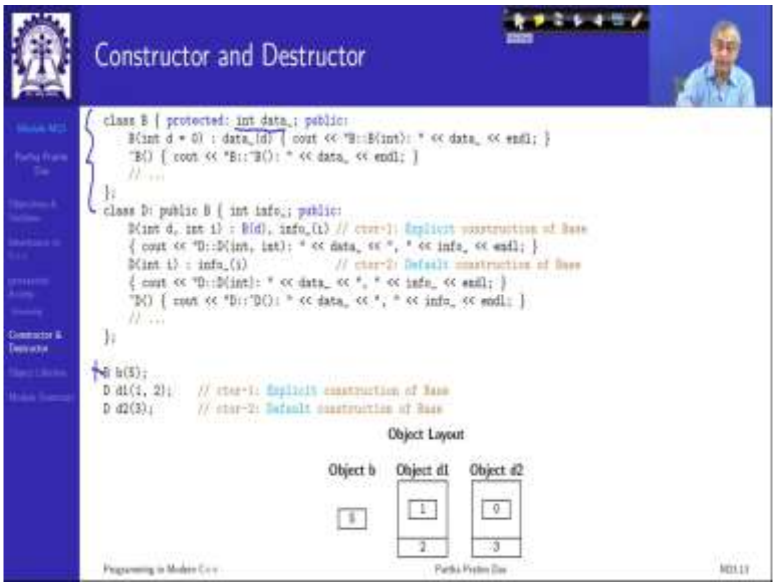
Constructor and Destructor

- Derived **ISA** Base
- Constructor-Destructor
  - Derived class *does not inherit* the Constructors and Destructor of Base class but *must have access to them*
  - Derived class *must provide* its own Constructors and Destructor
  - 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

Programming in Modern C++ Partha Pratim Das M03.12

And then we will have to work on the actual sequence of doing things. Certainly, base class object instances a part of the derived class object layout. So, when you go to construct a derived class object, you must first have to construct the base class part of it and then only the direct transport can be constructed, because the derived class part may use some of the values that the base class part has, some of the data members that base class has. Similarly, the destruction has to go in the reverse order.

(Refer Slide Time: 19:24)



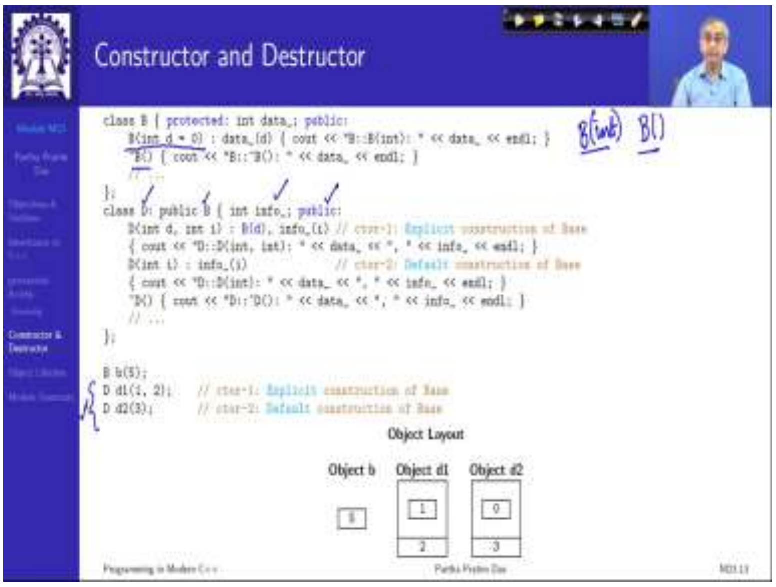
Constructor and Destructor

```
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
```

Programming in Modern C++ Partha Pratim Das M03.13

So, let me illustrate this through a simple example. There is a base class, one data which I have kept in protected, just for because it is only for this and the constructor destructor is put in public, I could have put them also in protected. If I put them in protected, then I will not be able to construct a B object as I am doing here outside, so I have kept it in the public.

(Refer Slide Time: 19:58)

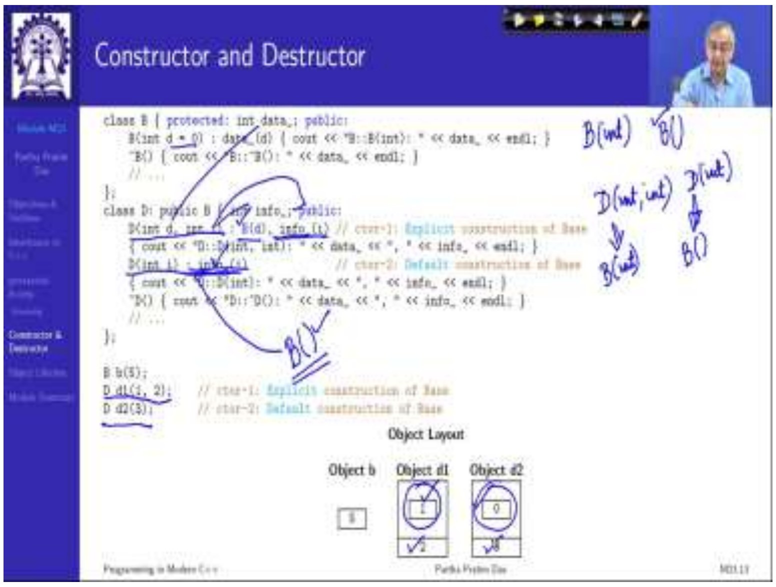


Constructor and Destructor

```
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
```

Now what is the constructor? The constructor is, B is a constructor with a default parameter. So, basically, I have two constructors. One is B int. And another is just B. One is default and other is B int, and I have the destructor, simple scenario. In the class D, derived class of B, I have kept the info in private because it is just single inheritance I am showing. Constructor destructor are public, so that I can use them.

(Refer Slide Time: 20:39)



Constructor and 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
```

Now, let us look at the constructor. So, again, I will right here be int two constructors I have and B nothing. I have a constructor, which takes two parameters. So, I have D, int, int. The meaning is, meaning I am taking is this int is meant to be the data of the base class. And this i should be the info, I mean, the info of the derived class, that is whatever. So, what I have to do, if I am doing it for say, d1, which is 1, 2 I have called, so D is 1.

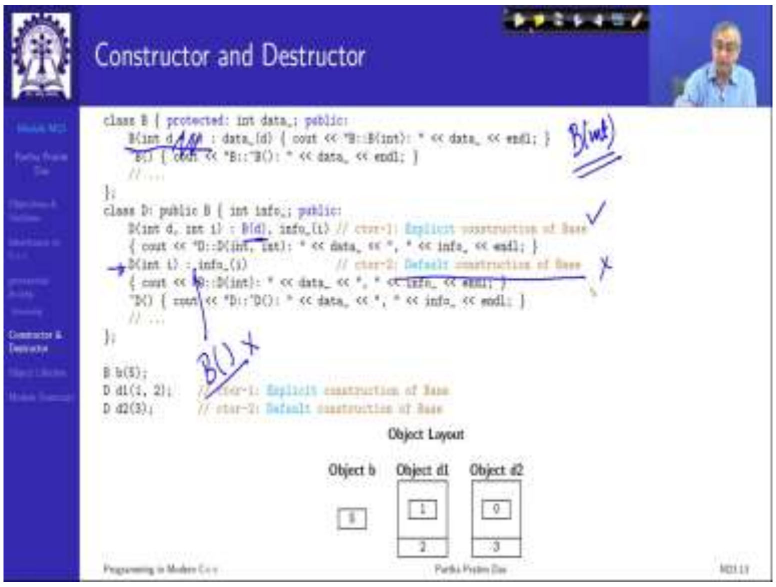
So, I have to first construct this part of the object, the instance, which is embedded, so I call B(d). So, from here, I am calling B int to construct this. So, in terms of initialization, now, it is not enough just to initialize the data, data members of the class, I have to initialize a base class part of that. So, that is the first thing I have to do.

Again, it does not matter in which order I write, wherever I write this, this is always going to be the first on the initializer list to be done. Because without that, I cannot proceed. And then I do the rest, which does this part. I have a second constructor given which takes only one parameter. So, that is D int. What it does? Just look at it is not, it also needs to construct the base class instance, but it is not calling the base class constructor.

So, it is like D d2. If it is doing like this, what it means that the default constructor of the base class must be there and will be invoked. So, first, this is invoked, though it is not written, but it has to construct. So, if base class it finds that base class has a default constructor, so it constructs with the default constructor, which sets it to 0, so, you get this part of the object d2 specify set.

And which means that this second constructor of the derived class is using the default constructor of the base class. So, this gets constructed and then on the initializer info is set to i which is 3. So, this part is created. So, this is the basic mechanism that has to happen.

(Refer Slide Time: 23:50)



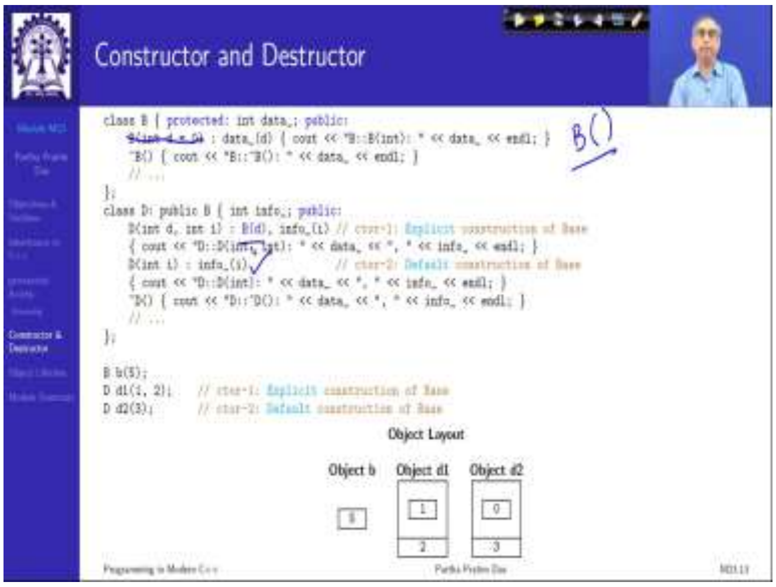
Constructor and Destructor

```
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
```

So, in construction, it will always be the constructor of the base class which will be invoked and then the rest of the initialization of the data members of the derived class in the order they are listed, we have understood these rules well then those will be covered one after the other. It is important to see that suppose here I had given default constructor in the process. What if I did not write this, did not default this parameter?

So, what will mean that I have only one constructor which has B int. In that case, this constructor which requires B default, because I have not invoked it, does not get an appropriate constructor in the base class not there. So, this constructor with the default constructor of base expected will simply not compile because compiler does not know which constructor to call, it needs a parameter. So, you have not given that which you have given here. So, this is okay but this will not be okay.

(Refer Slide Time: 25:23)



Constructor and 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
```

B()

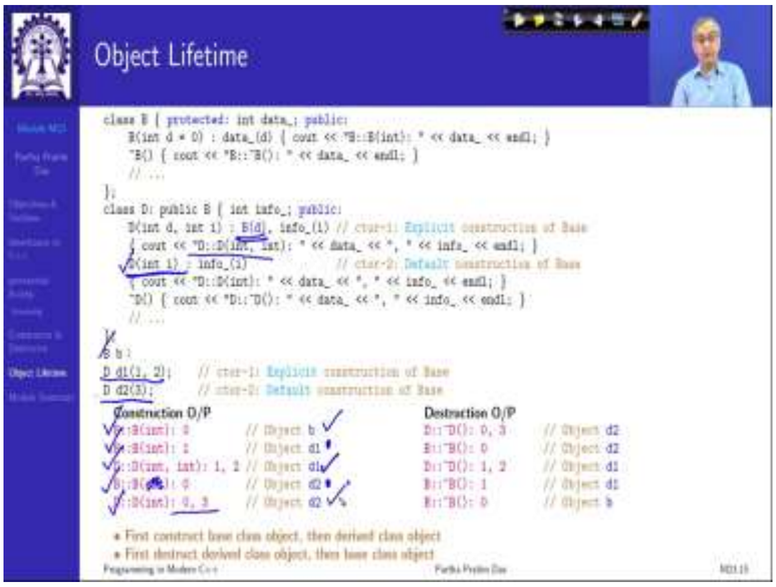
Object Layout

Similarly, of course, the other possible case is much easier to see that instead of writing this, if I had just written the default, then obviously this will compile, this will not compile because of the parameter mismatch. So, this is the basic construction story in the in terms of specialization of a derived class from a base class. The destruction does not need any specification because you know, what you have, and you just destroy things in the reverse order of how you have created.

(Refer Slide Time: 26:05)

Object Lifetime

Object Lifetime



Object Lifetime

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

Construction O/P

```
B::B(int): 0           // Object b
B::B(int): 1           // Object d1
D::D(int, int): 1, 2   // Object d1
B::B(int): 0           // Object d2
D::D(int): 0, 3        // Object d2
```

Destruction O/P

```
D::~D(): 0, 3   // Object d2
B::~B(): 0      // Object d2
D::~D(): 1, 2   // Object d1
B::~B(): 1      // Object d1
B::~B(): 0      // Object b
```

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

Programming in Modern C++ Partha Pratim Das M21.15

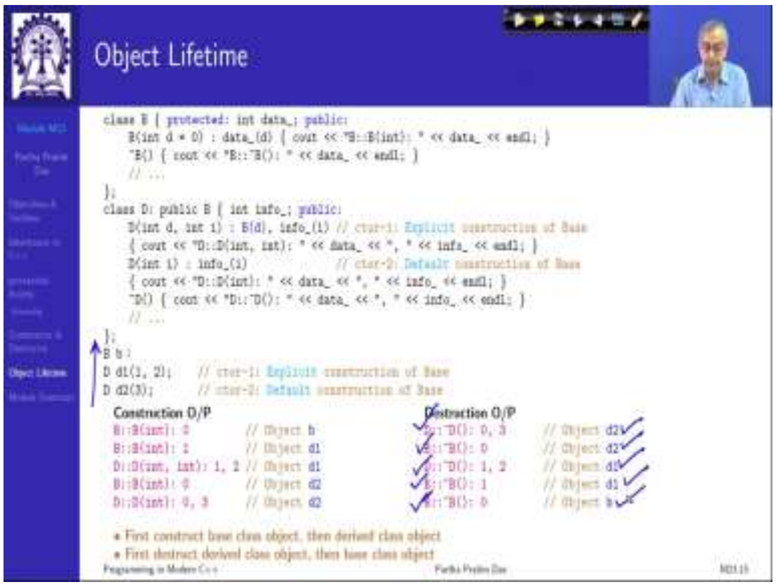
So, this naturally impacts your lifetime. So, I will end with a lifetime example of the same design. So, you have base class with this pair of default and non-default constructor and two constructors for the derived class and these instances. So, what will happen when I, if I, so in what you see in terms of magenta is a trace of the output that you are getting. So, first this has to be constructed.

So, the base class constructor with int value 0 will get called object B, then this will have to get constructed which means that the base class constructor with the value d, which is 1 has to first happen. So, the base class constructor is called with value 1, then only the remaining part of the derived class construction can happen the, derived class construction can happen. So, now you have the derived class construction with one two.

So, this was here the object d1 is ready, here it was being worked on, it is partially ready, then you have d2. So, this will use this constructor, which means it will use the default constructor of the base class. So, again, you have the base class constructor. This is a typo; this should be removed.

This is a base class default constructor, which naturally takes the value 0, partially d2 is constructed, then this constructor of derived class D goes on and you have the object d2 constructed. So, this is all, on in, this is how the lifetime of these objects one after the other are studying. So, it is objects as well as the base part of the embedded of the derived class object.

(Refer Slide Time: 28:38)



Object Lifetime

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

```
Construction O/P                               Destruction O/P
B::B(int): 0            // Object b            D::~D(): 0, 3     // Object d2
B::B(int): 1            // Object d1           B::~B(): 0        // Object d2
D::D(int, int): 1, 2    // Object d1           D::~D(): 1, 2     // Object d1
B::B(int): 0            // Object d2           B::~B(): 1        // Object d1
D::D(int): 0, 3         // Object d2           B::~B(): 0        // Object b
```

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

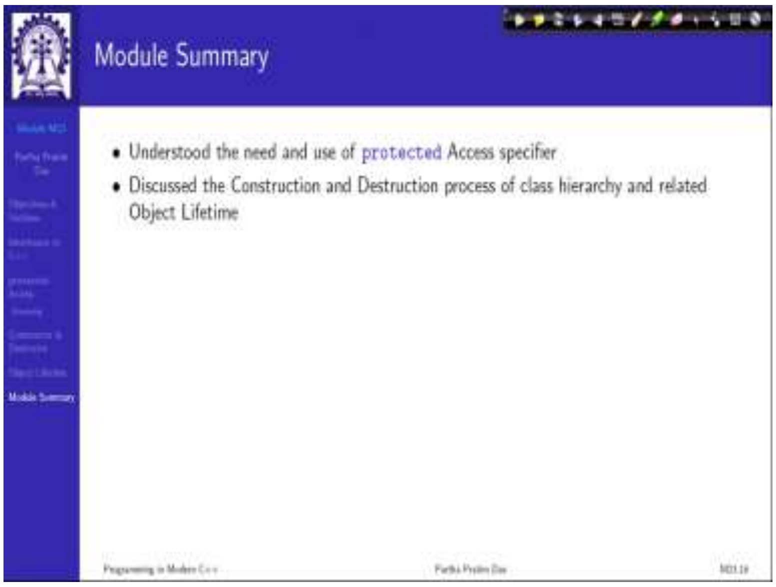
Programming in Modern C++ Partha Pratim Das M21.15

So, you can easily see that the way it will happen is it will the destruction will happen in the reversal. So, first, you will have the destructor called for the last object d2 created. Now what you will have to do, it has to destroy the base class. So, it will call the destructor of the base class for the value 0. And then the task gets completed. Next is destruction for d1 which calls the destructor for class B with having the value 1.

So, the destruction of d1 gets completed. And finally, the destruction for the base class object B which is with value 0, and object B gets destructed. And that completes the lifetime scenario of all of these objects. And I have purposefully taken a very simple whenever there is a key idea to retain or to use in several times in your programming, in your assignments, examination, I tried to put a key small example which captures the entire thing in one slide.

So, that you can just you know refer to this slide and you have the entire answer to the object lifetime under construction destruction of base class derived class object, everything else is an extension of that. You can have further derivations, you can have hybrid inheritance, you can have multi level inheritance, you can have various context in which these objects are placed, they could be automatic, they could be static, they could be constant, nonconstant, all everything else you can work out based on your earlier knowledge, but this is the core of how objects get constructed and destructed if they are on a hierarchy.

(Refer Slide Time: 30:33)



Module Summary

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

So, this brings us to the end of this module where we have understood the use of protected access specifier and in particular, how does it help in providing the preservation of information hiding in a proper manner on the hierarchy, and then we have discussed the very critical construction destruction process of class hierarchy and the related object lifetime.

I hope you have enjoyed this. And this we are going through a very, very key area of C++ and object-oriented design. So, please make sure that you understand this very well you can start practicing now. And going on to the next module. I will take up a little bigger example of the phone to give you an illustration on how the complete story can be stitched together. Thank you very much.