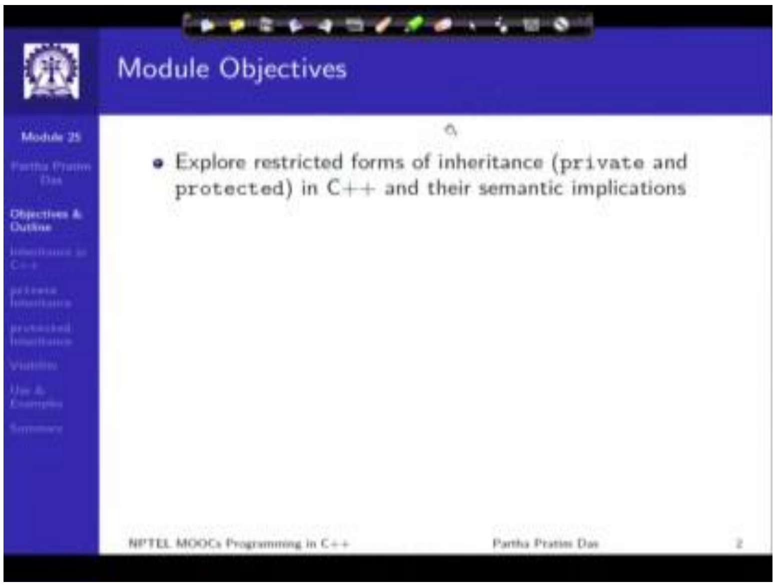
Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 40
Inheritance: Part V

Welcome to Module 25 of programming in C++. We have been discussing Inheritance and with the introduction of all major features of inheritance for generalization specialization or inheritance for ISA hierarchy in C++, we have introduced all the features they are semantics. And in the last module, we took the example of different kinds of phones, a hierarchy of phones; and illustrated how we can start actually designing a set of classes with data members and member functions to realize a hierarchy.

(Refer Slide Time: 01:05)

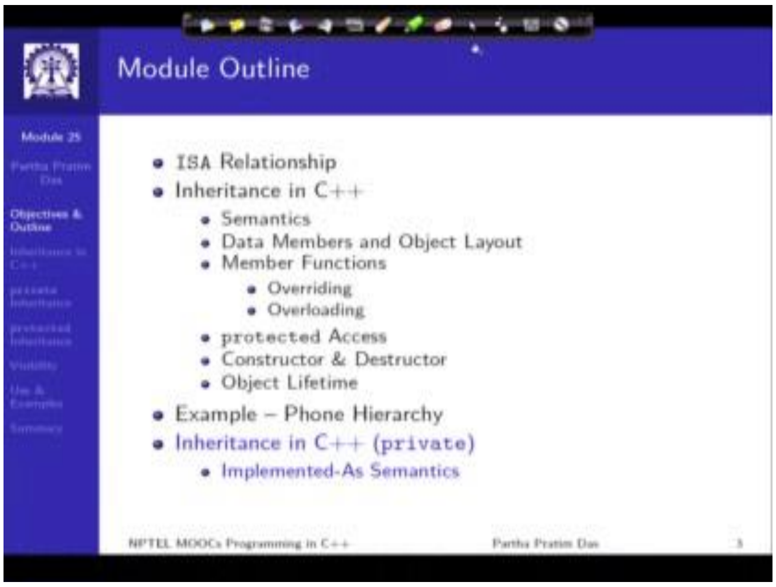


Module Objectives

- Explore restricted forms of inheritance (`private` and `protected`) in C++ and their semantic implications

Now in this context, in this basic series of discussions on inheritance, this will be the last module where we explore some more restricted form of inheritance that C++ offered. They distinguish themselves, there are actually of two types private and protected. They distinguish themselves from the inheritance that we have discussed earlier, because they no more they are called inheritance in C++, but they are no more represent the generalization specialization hierarchy of the object oriented sense.

(Refer Slide Time: 01:41)



Module Outline

Module 25

Partha Pratim Das

Objectives & Outline

Inheritance in C++

private Inheritance

protected Inheritance

Visibility

Use & Examples

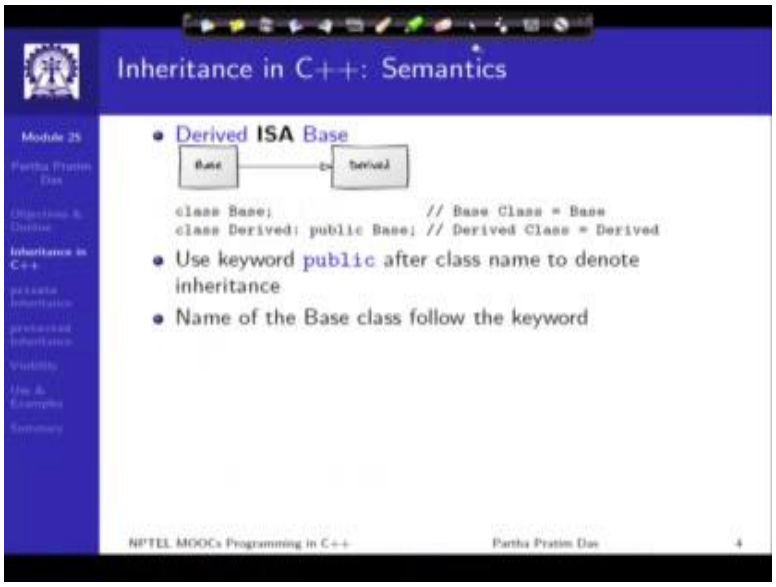
Summary

- 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

So, they are more used for implementation then for actually modeling or the interface or at the design stage. So, in terms of this outline, we are at the discussion of such restricted inheritance.

(Refer Slide Time: 01:54)



Inheritance in C++: Semantics

Module 25

Partha Pratim Das

Objectives & Outline

Inheritance in C++

private Inheritance

protected Inheritance

Visibility

Use & Examples

Summary

- Derived **ISA** Base

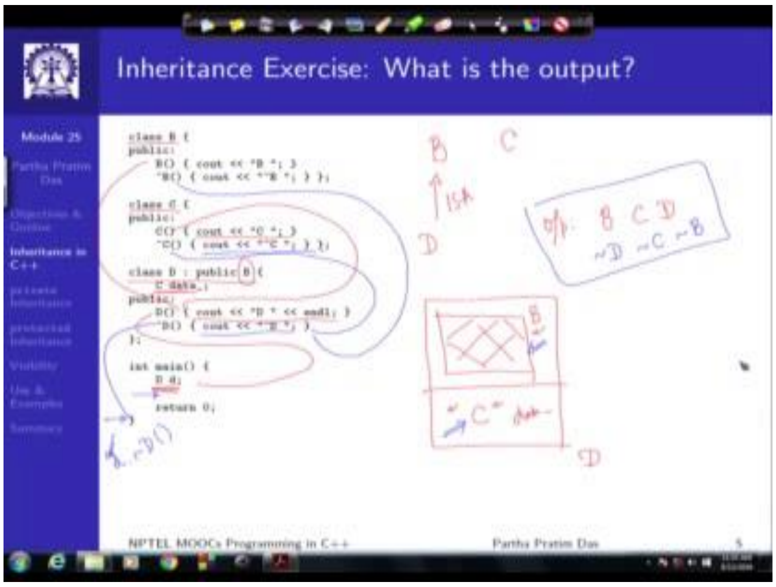
```
class Base;                    // Base Class = Base
class Derived: public Base;    // Derived Class = Derived
```

- Use keyword public after class name to denote inheritance
- Name of the Base class follow the keyword

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, this is the basic inheritance semantics which we have seen and I would like to draw your attention to this public keyword which defines the typical O A D ISA relationship in terms of an inheritance in C++ classes.

(Refer Slide Time: 02:10)



Inheritance Exercise: What is the output?

```
class B {
public:
    B() { cout << "B "; }
    ~B() { cout << "~B "; } };

class C {
public:
    C() { cout << "C "; }
    ~C() { cout << "~C "; } };

class D : public B {
    C data_;
public:
    D() { cout << "D " << endl; }
    ~D() { cout << "~D "; }
};

int main() {
    D d;

    return 0;
}
```

So, just to recapitulate. If I have a set of classes and we want to know how these classes or objects of these classes will get constructed. So, if you have given such a situation, then you have class B, you have class C, you have class D, which is a specialization of class B, and then you are constructing the object of class D. The question is what is the order in which the C outs happen which will tell you what is the life cycle or lifetime of the D object and the other embedded objects in this.

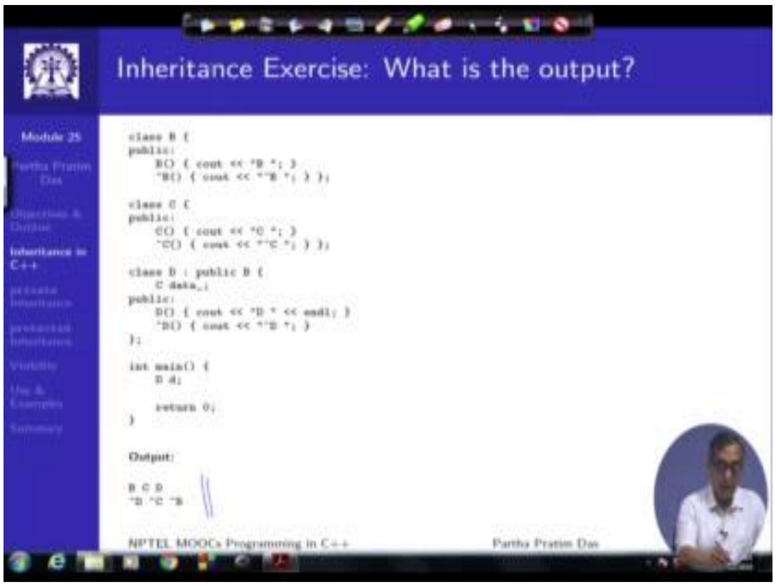
So, whenever you encounter such situations you should quickly try to draw the basic diagram of the relationships. So, here we know that there is a B and C is an independent class. So, this is the basic relationship of the classes. In terms of a D object what do we expect, so this is a D object; in terms of that we have a C data and it is inherited from B. Since it is inherited from B, it will have a base class instance of type B; and as a member, it will have a C class object. So, this is what a basic structure of the layout is. So, once you have been able to draw this, you will be easily able to figure out as to what happens when I try to instantiate an object. Certainly the instantiation of the object, we will start by invoking the constructor of D.

And then what is the first thing that needs to be done D ISA B, so the base class instance of that will have to be constructed. So, this means that the first thing that this intern will call is the constructor of B. And once a constructor of B is called it is executed. So, in the output, if I write the output here in the output you see an output of B. Once this base class instance has been constructed, then the next thing is to construct the data members one by one, so there is one data member. Now this will have to get constructed. So, after this D will give the next call to the constructor of C, because this data underscore member will have to be constructed. So, as it gives call to the constructor of C will have an output C.

Now once this has been done, so the base part instance has been created, the data member has been created, now the actual constructor of D will execute, so you will have an output D. So, after all these has happened you are now at this position where your whole D object has been constructed. At a later point of time, when you reach the end of the scope, naturally it will have to get destructed. So, if it has to get destructed what will get called first, so this is what you cannot see actually what the compiler has put here invisibly is a call to the destructor of D. So, at this point, you will have a call to the destructor of D. And as I said the destruction order is just the opposite. So, the destruction of D whatever actions are required that will be taken first. So, we will first have the output of tilde D. When you reach the end of this, what was constructed just before this was the C the data member. So, this will lead to the call of the destructor of C, which will destroy C will in the output you see this.

And at the end of this scope, it has finished. So, it comes back here again and now you have to destroy the base part instance of the D object. So, you again call the next one which is the destructor of the base class which will print B. And then the function will actually end. So, if you have different hierarchies, it is a good exercise to just put the different print messages in the constructor destructor and trust them to be able to understand how really, what really is a lifetime that happens. So, this was just kind of a recap on what we did.

(Refer Slide Time: 07:13)



Inheritance Exercise: What is the output?

```
class B {
public:
    B() { cout << "B "; }
    ~B() { cout << "~B "; } };

class C {
public:
    C() { cout << "C "; }
    ~C() { cout << "~C "; } };

class D : public B {
    C data_;
public:
    D() { cout << "D " << endl; }
    ~D() { cout << "~D "; }
};

int main() {
    D d;

    return 0;
}
```

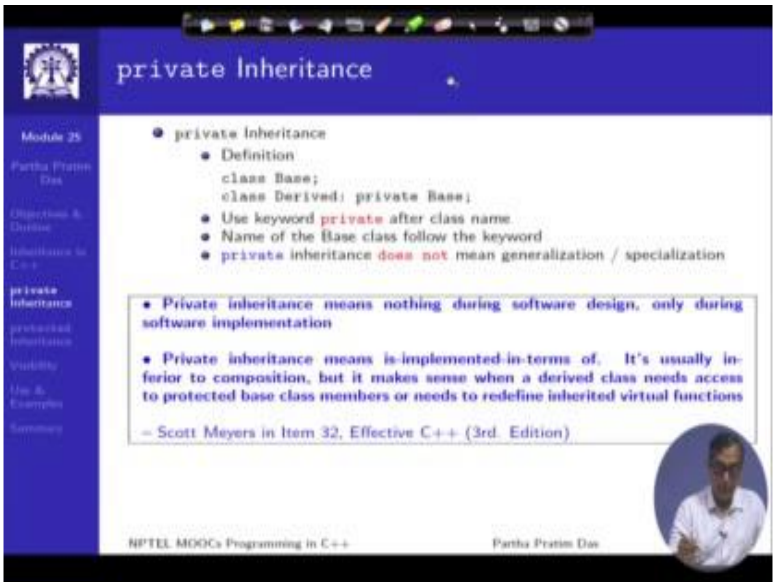
Output:

```
B C D
~D ~C ~B
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, I just worked out another example, so that you can understand it better. In the next slide, I have actually since I wanted to work out. So, I output was not given in the earlier slide, now it is given here. So, you can practice at a later point of time

(Refer Slide Time: 07:27)



private Inheritance

- private Inheritance
  - Definition

    ```
    class Base;
    class Derived: private Base;
    ```
  - Use keyword private after class name
  - Name of the Base class follow the keyword
  - private inheritance does not mean generalization / specialization

• Private inheritance means nothing during software design, only during software implementation

• Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions

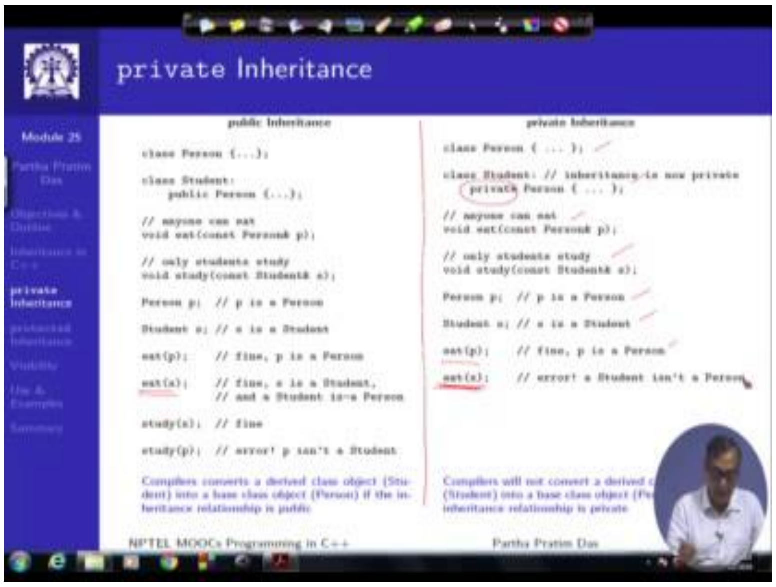
– Scott Meyers in Item 32, Effective C++ (3rd. Edition)

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now coming to private the restricted forms of inheritance let me first discuss about private inheritance. Private inheritance looks pretty much like the inheritance that we have seen which by contrast will say is the public inheritance or just simply inheritance. The only thing different is the key word that we use, you said this is the private inheritance so you write the keyword private. And private inheritance as we understand is not of meant for representing the hierarchy in the design, rather it is used for kind of saying that some part of the object is implemented as some other base class object.

So, it is more like a kind of a candidate alternative for composition, which will see some are later in this presentation in this discussion and it is not to be confused with the basic is a hierarchy that we are modeling to the public inheritance. So, that is why one of the very famous authors' comments that private inheritance means nothing during software design, because during the design time you just know the ISA hierarchy is you do not know the internals of the implementation. So, only when you get to know the internals of the implementation on you are working on that that is where you look into this private inheritance kind of semantics.

(Refer Slide Time: 08:54)



private Inheritance

public Inheritance

```
class Person {...};

class Student:
    public Person {...};

// anyone can eat
void eat(const Person& p);

// only students study
void study(const Student& s);

Person p;  // p is a Person

Student s; // s is a Student

eat(p);    // fine, p is a Person

eat(s);    // fine, s is a Student,
           // and a Student is-a Person

study(s);  // fine

study(p);  // error! p isn't a Student
```

Compilers converts a derived class object (Student) into a base class object (Person) if the inheritance relationship is public

private Inheritance

```
class Person { ... };

class Student: // inheritance is now private
    private Person { ... };

// anyone can eat
void eat(const Person& p);

// only students study
void study(const Student& s);

Person p;  // p is a Person

Student s; // s is a Student

eat(p);    // fine, p is a Person

eat(s);    // error! a Student isn't a Person
```

Compilers will not convert a derived c (Student) into a base class object (Pe inheritance relationship is private

NPTEL MOOCs Programming in C++

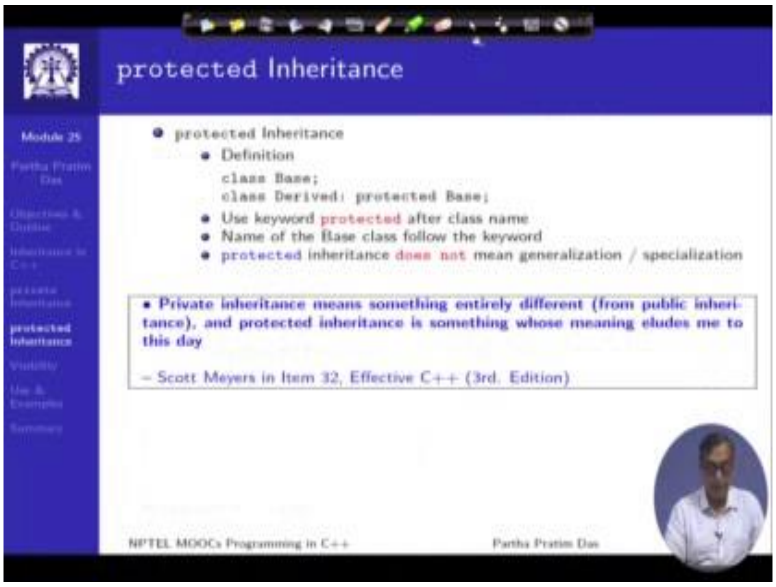
Partha Pratim Das

So, if we try to kind of compare between the public and private inheritance, I am just using a simple example here this is again from Scott Meyer's book. I have a base class person, I have a derived class student. And certain I have two different functions eat and study. Certainly eat applies to persons, because everybody needs to eat; and study applies only to students because those who study are called students right. So, there could be persons who do not study. So, if P is a person and S is a student then eat will work on P, interestingly it will also work on S and that is a basic property of public inheritance. We had seen this during the presentation, during the discussion on inheritance itself, I had talked about output streaming operator and showed how passing a derive class object to a base class parameter would still work through the process of casting. So, in terms of the meaning, the semantics what is being said that we expect this to work because someone may be a student, but certainly being a person S also is expected to eat. So, this should work fine.

Similarly, study will work for S, but certainly what will not work is study P, because study is something which is associated with the students. The derived class object and you do not expect the base class the generalized object to be able to satisfy that. So that is what public inheritance mean. Now in contrast in private inheritance, if you look at the same thing the same set of classes, the same functions, the same to P and S, eat P will work, but what is see eat S, if you have defined this as a private inheritance. Now in the context of C++ this is an error, this is the compiler does not allow this. Whereas in case of public it does allow it that is a basic semantic difference.

Why is it not allowed, because here the inheritance. So, it clearly goes to show you that here what we write in the notation of inheritance in C++ as private is not semantically meant to specialize the concept. So, this is just something else, this is just saying that student is possibly implemented as a person and so on. So therefore, the functionality which is for the person is cannot be available to the student, it is component functionality not generalization specialization functionality. So, in terms of private inheritance always keep this concept in mind

(Refer Slide Time: 11:47)



protected Inheritance

Module 25

Partha Pratim Das

Objectives & Outline

Inheritance in C++

private Inheritance

protected Inheritance

Visibility

Use & Examples

Summary

- protected Inheritance
  - Definition

    ```
    class Base;
    class Derived: protected Base;
    ```

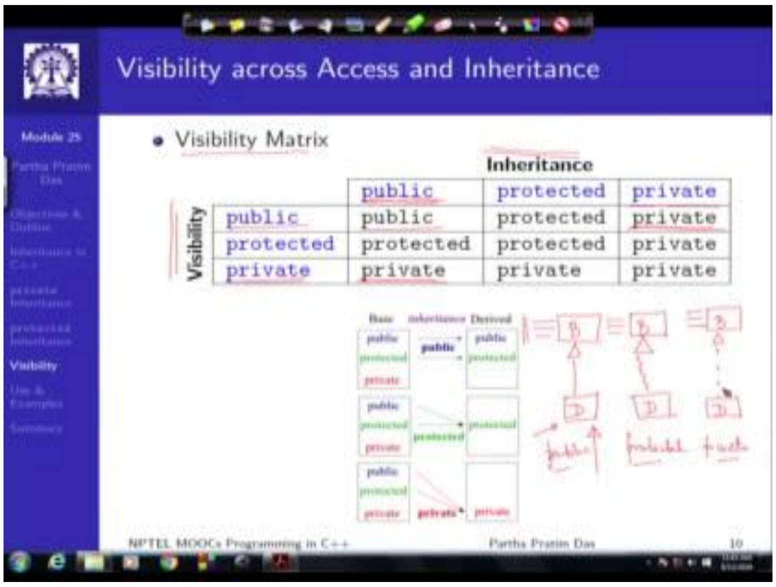
  - Use keyword protected after class name
  - Name of the Base class follow the keyword
  - protected inheritance does not mean generalization / specialization

> • **Private inheritance means something entirely different (from public inheritance), and protected inheritance is something whose meaning eludes me to this day**
>
> – Scott Meyers in Item 32, Effective C++ (3rd. Edition)

NPTEL MOOCs Programming in C++ Partha Pratim Das

By symmetry that the third form of inheritance that C++ provide, where instead of writing public or private we could also write protected at the this point of inheritance structure. And we will see the implications of doing that. I would not discuss much of protected inheritance, because it does not offer very distinctive semantics, and many authors have commented that possibly there is no well-defined design situation, where the protected inheritance may be required. So, it is there more for the completeness purpose of the features.

(Refer Slide Time: 12:33)



Visibility across Access and Inheritance

- Visibility Matrix

Visibility \ Inheritance	public	protected	private
public	public	protected	private
protected	protected	protected	private
private	private	private	private

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

Now, what is important is if we have, so now you have two sets of parameters that control how we get the access, how we get the visibility across the base and the derived class. So, the question is if I just draw, I could have base and specialized as D which is basically public, or I could have base and there is no specific notation available. So, I am just using some wavy notation that this is where the derived class D is derived through a protected manner. And similarly, I have base and let say another this notation also does not exist, this is just I am creating it to represent say this means derived in the private way.

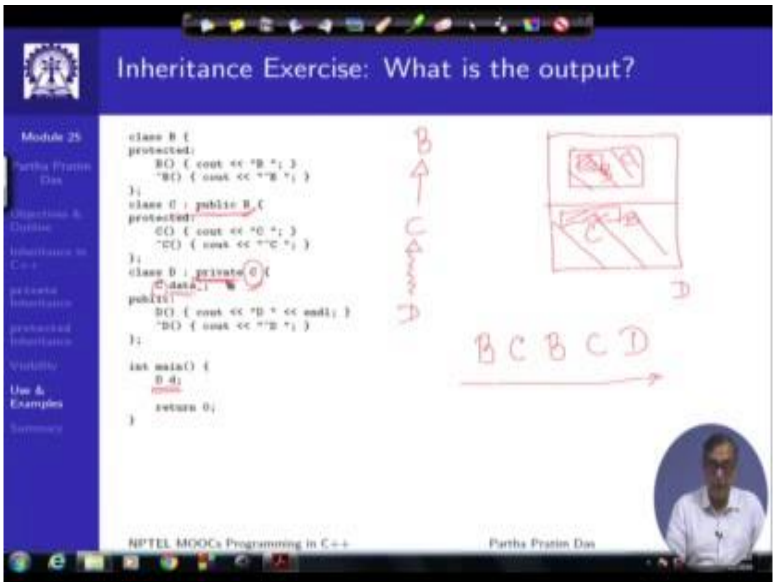
Now the question is between these three types of inheritance, public protected and private, in addition to this the base class has three sets of visibility already available. Some members are public, some members are private, and some members are protected. So, in the derive class, what would happen to the visibility of these members and even downwards when I go further below from the derived class what will happen to the visibility of those (Refer Time: 14:17), so that is defined by this visibility matrix. So, here we say what are the visibilities that is, what are the access specifiers given in the base class. And here and the column we say what is the kind of inheritance public, protected or private that is used, and their combination decide as to what is the resultant visibility of an object. So, if I have a publicly visibled member, and the derived class inherits it using public inheritance then the resultant member would remain to be public, but if it inherits it by say the private inheritance, the resultant member will come private.

So, in other words say if some member has a private visibility, private access specifier, inheritance is public, the resultant visibility in the derived class would become private that is in the derive class this will be treated as if it is a private member and so on. So, it is I mean you may feel that you will have to remember these nine entries, but there is very simple thumb rule to understand this. We now understand that public is a most open visibility that is list encapsulated visibility. Protected is little bit more encapsulated then that and private is the most encapsulated visibility. So, there is the hierarchy between these three.

So, if you look into this, basically a hierarchy is considered on this as well. So, if you are trying to understand, what this entry should be, you should look at what are the visible access specifier and what is the inheritance whichever is more restricted out of these two will be the result. So, public private will be private. Similarly if I have private public, I will have private. If I have protected public, I will have protected. If I have protected private, I will have private because between the protected and private, private is more restrictive. So, this basically the thumb rule is simple that you need to be needed to preserve the encapsulation as much as are possible. So, you become more restrictive and this diagram below clearly visually depicts as how the access specification gets re positioned after the inheritance has taken place.

Certainly, before discussing the restricted types of inheritance, we were not we did not have to discuss this, because if you look into just the first column, since the inheritance is the list restrictive inheritance is the provides the list encapsulation, therefore, the visibility of the data members or the member functions in a derived class is simply same as their visibility or visibility in the base class. It is public is public, protect it is protected, private is private. But as we introduce other forms of inheritance, this as access restriction description is becomes required.

(Refer Slide Time: 17:37)



Inheritance Exercise: What is the output?

```
class B {
protected:
    B() { cout << "B "; }
    ~B() { cout << "~B "; }
};
class C : public B {
protected:
    C() { cout << "C "; }
    ~C() { cout << "~C "; }
};
class D : private C {
    C data;
public:
    D() { cout << "D " << endl; }
    ~D() { cout << "~D "; }
};

int main() {
    D d;

    return 0;
}
```

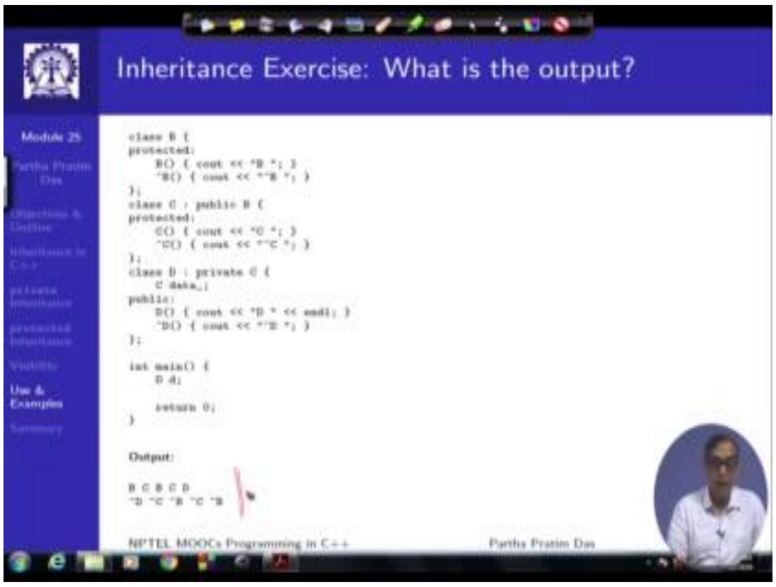
NPTEL MOOCs Programming in C++ Partha Pratim Das

So, with this, we can again go back and try to look at some of the little bit exercise. This is an exercise showing you the similar structure as before, but we have here both public and private inheritance. So, if we want to see again in terms of the construction of D what should happen? Then let me, this is the class B we know, C is a B so this is what we draw here. Then we have we say that D is private C. So, let us just draw it like this. So, this is what we get to see. Then in terms of the object layout we find the D object should have a base part, which is C, the base part is C. Then it has a data part, which is also C, this base part is C as well as the data part is also C, and that is all that we have. We have C is a B, C is a B here, so C again will have a base part which is B and this will have a base part, which is B. We have every C object will have a base class instantiation because is a specialization here.

So, given this, if you now try to trace, how the objects will constructed, certainly as soon as this is encounted the constructor of D is called, so the base has to get instantiated. So, the constructor of C will get called. So, the base of that will have to get constructed. So, first thing that will get constructed is the B object, this B object. Then certainly the construction of the base object completes. So, C will get to that completes the construction of this. Then that data member this data member has to get constructed which also has base part so that will get to constructed then the data member gets constructed, and then finally the D object gets constructed. So, this is the order in which the construction will happen.

So, in terms of private inheritance also the life time issues do not get affected, life time issues remain same. So, we had seen if this kind of an exercising terms of just public inheritance. So, intentionally I have introduce private inheritance here that changes the access specification, what you can access, what you cannot access, but in terms of the dynamics of the object life time things do not really become different, they remain to be same. The full solution is given in the next slide here.

(Refer Slide Time: 20:50)



Inheritance Exercise: What is the output?

```
class B {
protected:
    B() { cout << "B "; }
    ~B() { cout << "~B "; }
};
class C : public B {
protected:
    C() { cout << "C "; }
    ~C() { cout << "~C "; }
};
class D : private C {
    C data_;
public:
    D() { cout << "D " << endl; }
    ~D() { cout << "~D "; }
};

int main() {
    D d;

    return 0;
}
```

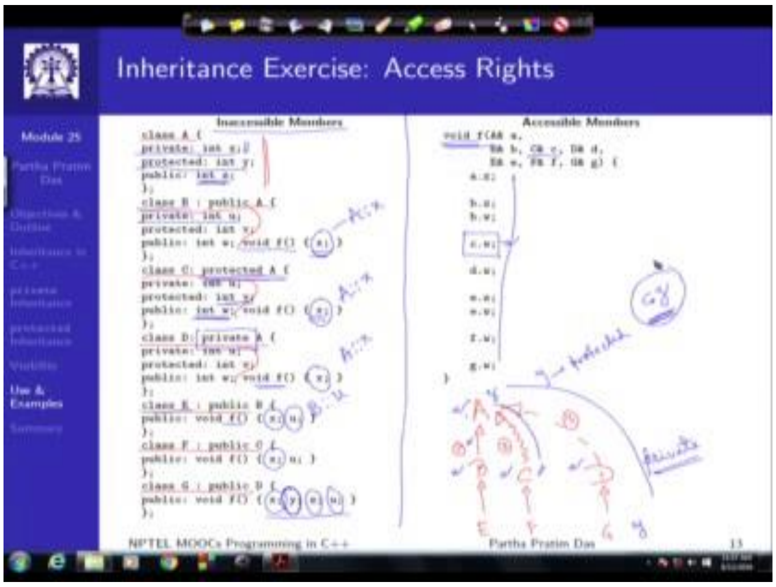
Output:

```
B C B C D
~D ~C ~B ~C ~B
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, at leisure, you can work out and get convenience that this is what is going to happen.

(Refer Slide Time: 21:01)



Inheritance Exercise: Access Rights

Inaccessible Members

```
class A {
private: int x;
protected: int y;
public: int z;
};
class B : public A {
private: int u;
protected: int v;
public: int w; void f() { x; }
};
class C: protected A {
private: int u;
protected: int v;
public: int w; void f() { x; }
};
class D: private A {
private: int u;
protected: int v;
public: int w; void f() { x; }
};
class E : public B {
public: void f() { x; u; }
};
class F : public C {
public: void f() { x; u; }
};
class G : public D {
public: void f() { x; y; z; u; }
};
```

Accessible Members

```
void f(A& a,
       B& b, C& c, D& d,
       E& e, F& f, G& g) {
    a.z;

    b.z;
    b.w;

    c.w;

    d.w;

    e.z;
    e.w;

    f.w;

    g.w;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

To understand the access specification, I have included an example again I will just do couple of steps and then leave it as an exercise for you to work out just first analyze what is there class A, so there is a class A. So, whenever you have this hierarchy related designs or hierarchy related issues, it Is good to just quickly draw a diagram, so that mentally you can see what is really going on. It is more difficult to frequently refer to I am sorry frequently refer to code and talk about this. So, we have B is A, then we have this. So, we say C is A; we also have D is A. This is public type 1, this is protected, and this is private. And then we have these are E is a B public; f is a C, which is public; g is a D which is public. So, this is the basic hierarchy that we have.

So, if you have that then, and we have different members here x, y, z - three members with three kinds of visibility; private, protected and public. And B adds u, v, and w, so does C, so does D, and these data members are also of private, protected and public kind. So, having done that our question is if we now look into the different access of variables in different member functions and this is a global function then what members can be access what members cannot be accessed. So, the way I demonstrate here is, here I only show I have shown on the left side, I have only shown the inaccessible members that is if I have a class f in I am sorry a function f in class B, then x is in accessible to this function. Why is it so? Where is the x coming from, B does not have x, but B is a A has an x. So, this x actually means that it is a colon colon x that has been inherited. And what is the access specifier private. So, certainly the derived class cannot access the private data members of the base class. So, this is inaccessible.

If you look at x in class C, it is again the same logic that this is private in A, therefore, you cannot access it; you cannot access it either in class D, because it is private in A, clear. Now let us say if we look into the class E, and the function f that the class E defines the fact that x will be inaccessible is clear, because x is private here therefore, x is private in B, x is private in C, x is private in D. Because it is private for B, it is private specifier public inheritance, resultant visibility private. For C, it is public sorry private specifier protected inheritance, protected inheritance, resultant is private. In A, this is specifier is private inheritance is private resultant in D, this is private. So, in all of B, C, D, we can reason that x is actually private. So, this access will not be accessible to any of this.

But in E, you find that u is not accessible. Why u is not accessible, what is u you are sitting here. So, where can it get in u form it, it does not have any data member itself. So, it can get the u from the parent class B. So, it B has a U. So, this u is basically B colon colon u which is private member of B therefore, you cannot access it here. You can do similar reasoning for the class F. Let us look at class G, class G cannot access, so many different things for example, class G cannot access y, class G cannot access y. So, where does it get y form, it does not have a y itself. So, where does it I have to get it, it will have to get on this path. So, let us it is parent is D, D does not have any y; u, v and w, it is does not have any y. So, where can it get if from, from the parent A has a y.

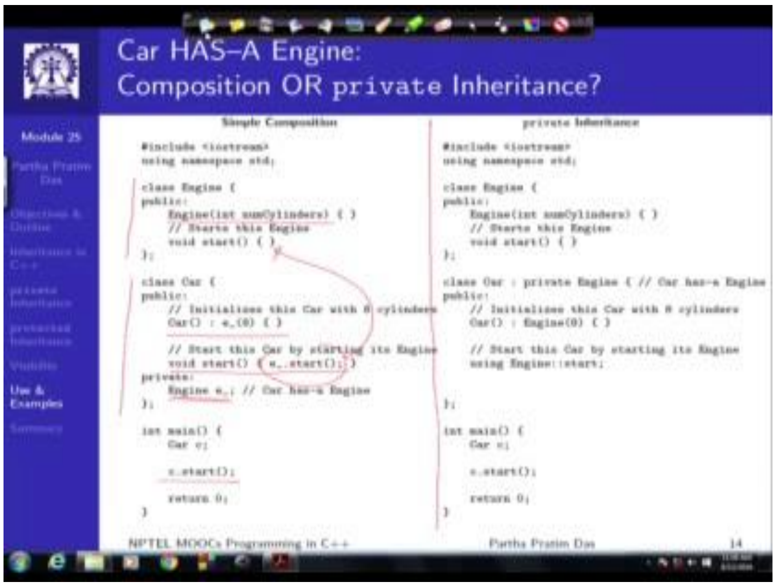
So, you know here, this is protected, this y is protected. So, the child class should be able to access it that is D should be able to access it. Can D access it, if I go to D, if I go to D, the function in D this f function can access y and that is a reason it is not mentioned here that is quiet because you can always access it from the parent, but what happens D has inherited a through a private inheritance mechanism. So, the y which was protected in A now becomes private in D, it is not written, but it is in within the inheritance this is become a private member and certainly no derived class can access a private member, so g cannot access is that is a reason you will not be able to g will not be able to access y.

You can reason through and see the same similar faith for these variables as well.

Going further, you could do, I will not do this now, you can do, you should be able to work this out. This is where we saw over this hierarchy, how does the access of different members actually again I have shown it all for the data members, but the similar reason you will apply for the member functions as well, how will this different forms of inheritance interact with the difference specifier will is clear in terms of the member functions. You could just now again try to solve this for a global function. And all that you will have to follow is a similar reasoning of going over the hierarchy. Wherever if you are trying to say that C dot w is accessible then C is of class C, so it is here. So, you see does C have a w C has a w. And being a global function it has to be able it will be able to access only those members which have public. So, C has a w which is public therefore, this is accessible.

But you think about any of the other variables that any of the other members that is C has none of them are accessible to for example, v is not accessible, because v here is protected; z is not accessible, why z not accessible, because if I talk about C dot z, then there is no z here. So, the z comes from A, which is public, it should have been accessible, but what has happened in the process of inheritance you have inherited it as a protected inheritance; C is inheriting A through the protected manner. So, the resultant visibility of z has become the lesser of public and protector. So, this is become protected member and the protected member is not visible outside the class, since F is a global function here it is not visible to F. So, C dot z is not accessible. So, in this way, I would request that you work out for this whole set and that will clarify all your doubts that you may have.

(Refer Slide Time: 30:02)



Car HAS–A Engine: Composition OR private Inheritance?

Module 25

Partha Pratim Das

Objectives & Outline

Inheritance in C++

private Inheritance

protected Inheritance

Visibility

Use & Examples

Summary

Simple Composition

```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car {
public:
    // Initializes this Car with 8 cylinders
    Car() : e_(8) { }

    // Start this Car by starting its Engine
    void start() { e_.start(); }
private:
    Engine e_; // Car has-a Engine
};

int main() {
    Car c;

    c.start();

    return 0;
}
```

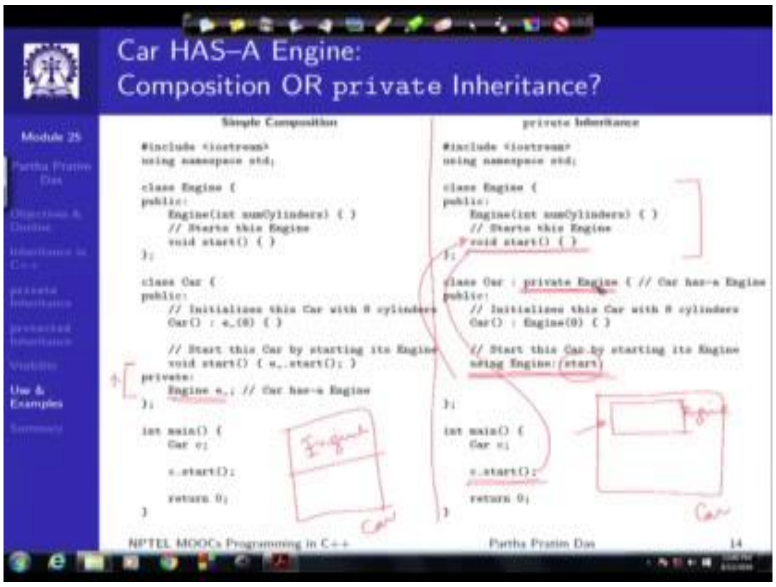
private Inheritance

```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car : private Engine { // Car has-a Engine
public:
    // Initializes this Car with 8 cylinders
    Car() : Engine(8) { }

    // Start this Car by starting its Engine
    using Engine::start;


};

int main() {
    Car c;

    c.start();

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

Now, finally, let me just try to show you that some possibilities of using private inheritance. We take one example again you should look at the left hand side first where I have an engine class, and have a car class, and the basic concept is that the car has a engine. So, if the car has a engine it has component. So, I can model it in this manner. So, that the constructor of the car instantiates the engine which invokes the constructor with eight cylinder value. So, I have a engine instance here. And therefore, and in terms of starting that engine car has provided method start which actually in turn invokes the method start of the engine. And if I do this, the car will get started, sample composition based model.

(Refer Slide Time: 31:03)



Car HAS–A Engine: Composition OR private Inheritance?

Simple Composition

```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car {
public:
    // Initializes this Car with 8 cylinders
    Car() : e_(8) { }

    // Start this Car by starting its Engine
    void start() { e_.start(); }
private:
    Engine e_; // Car has-a Engine
};

int main() {
    Car c;

    c.start();

    return 0;
}
```

private Inheritance

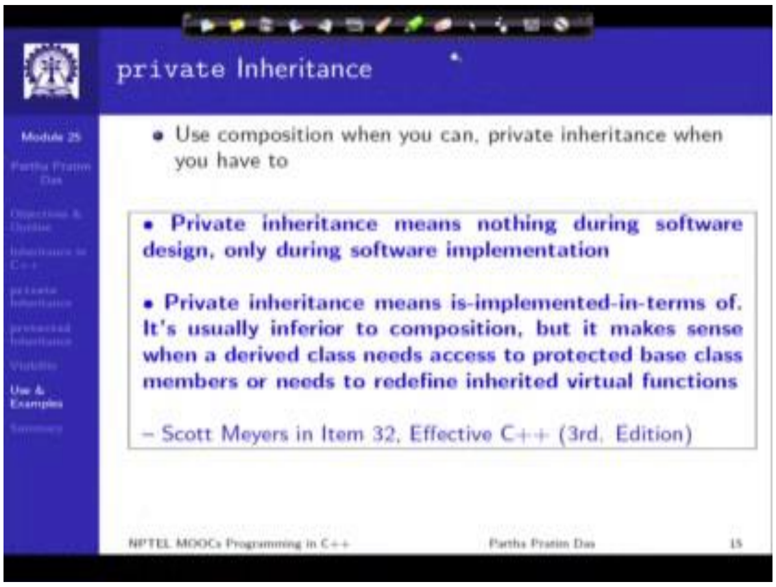
```
#include <iostream>
using namespace std;

class Engine {
public:
    Engine(int numCylinders) { }
    // Starts this Engine
    void start() { }
};

class Car : private Engine { // Car has-a Engine
public:
    // Initializes this Car with 8 cylinders
    Car() : Engine(8) { }

    // Start this Car by starting its Engine
    using Engine::start;
};

int main() {
    Car c;

    c.start();

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 14

Now I show in parallel for comparison that this could also be modeled in terms of private inheritance, I have the same engine class no difference, but what I change is I do not use this private data member instead I inherit from the engine class in a private manner. If I inherit from the engine class is a private manner, what do I have, here I had there are the class and the car class and I had the engine as a data member. Here since I have inherited I will still have the engine as a base class instance, I have the car here and this is a base class instance, so that that way I will be able to have the value, but certainly what I need to do I need to have ability to start the engine. So, that is what I specify here this is just another notation where you using. So, when you say using base class colon-colon method name then it means that whenever I talk of this name, it actually means the base class item, it means that base class function. So, if I do this then I can again instantiate the car and do C dot start; and C dot start through this using will actually call the start of the engine, which is again the engine will get started.

So, in here also the engine is not visible to others, because is a private data member, here also this is not accessible to others, because I have done a private inheritance. So, this is another way that some implementation can be I mean you could implement some kind of a component as private inheritance, and certainly this brings us to question of which one should we use naturally, there is no question of using private inheritance in a usual situation, you should always use composition.

(Refer Slide Time: 33:01)



private Inheritance

Module 25

Partha Pratim Das

Objectives & Outline

Inheritance in C++

private Inheritance

protected Inheritance

Visibility

Use & Examples

Summary

- Use composition when you can, private inheritance when you have to

• Private inheritance means nothing during software design, only during software implementation

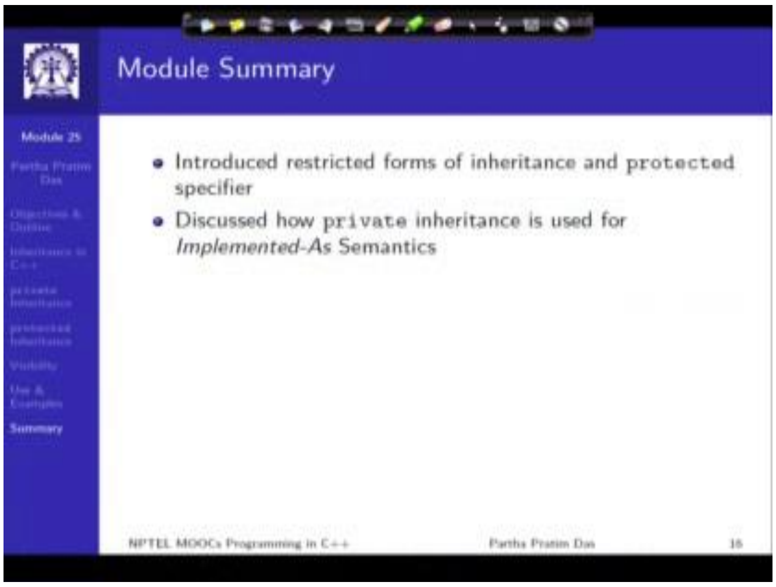
• Private inheritance means is-implemented-in-terms of. It's usually inferior to composition, but it makes sense when a derived class needs access to protected base class members or needs to redefine inherited virtual functions

– Scott Meyers in Item 32, Effective C++ (3rd. Edition)

NPTEL MOOCs Programming in C++ Partha Pratim Das 15

Only very few special situations where composition does not really work you must use private inheritance. And you use them only when those situations are faced. So, while we have while we discuss polymorphism later in this series, then at a suitable point I will show you examples using polymorphism of certain problems, where the private inheritance can give you actually a better solution than using composition. But as a thumb rule I would say rather 99.99 percent of cases, if there is a situation of composition, you should actually use data members and not use private inheritance in that place.

(Refer Slide Time: 33:46)



Module Summary

- Introduced restricted forms of inheritance and `protected` specifier
- Discussed how `private` inheritance is used for *Implemented-As* Semantics

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

To summarize, we have introduced here the basic notions of restrictions on inheritance in terms of private and protected inheritance. And we have discussed how really the protocol is defined through which the three kinds of public protected and private visibility interact with three kinds of inheritance public, protected and private inheritance. And at the end, we have try to show an example illustrating that how private inheritance can be used to realize what is more commonly known as something is implemented as something else or imp or uses a component of something else and, but with the question that we will use it in a very restrictive manner.

So, with this, we will bring close to the series of basic notions of inheritance. And from the next module onwards, we would talk about advanced aspects of inheritance, which lead to what is known as polymorphism that will be the core idea to discuss.