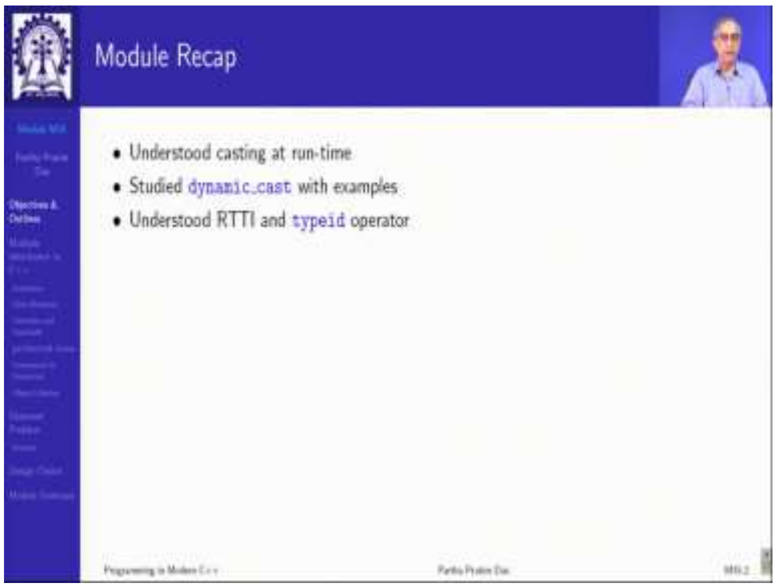
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 35
Multiple Inheritance

(Refer Slide Time: 0:35)



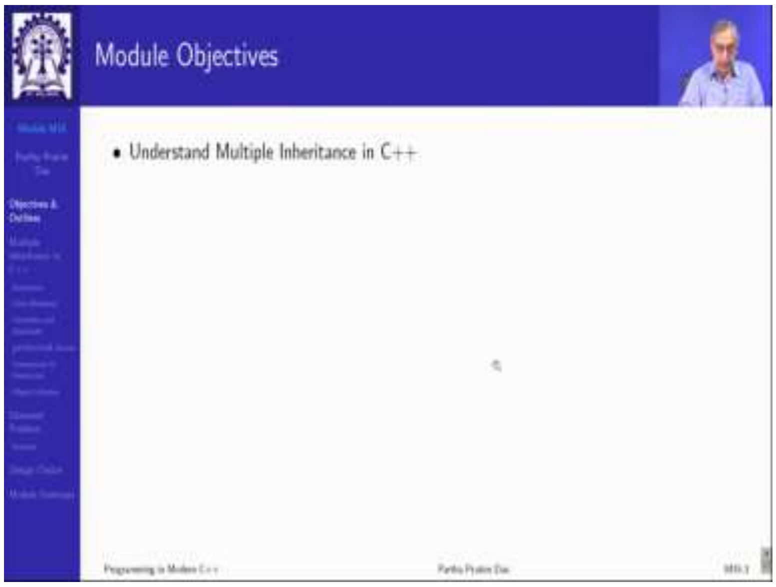
Module Recap

- Understood casting at run-time
- Studied `dynamic_cast` with examples
- Understood RTTI and `typeid` operator

Programming in Modern C++ Partha Pratim Das M35.2

Welcome to Programming in Modern C++ we can we are in week 7 and we are going to discuss module 35. In the last module, we concluded the three-part discussion on typecasting in C and C++. And in the last module, we particularly discuss this very important casting for polymorphic type by dynamic cast, the typeid operator and the required runtime type information system.

(Refer Slide Time: 1:03)



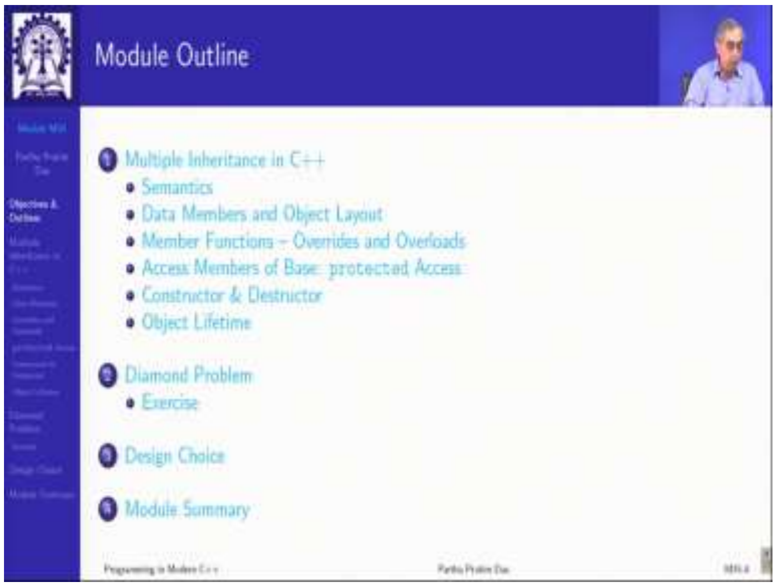
Module Objectives

- Understand Multiple Inheritance in C++

Programming in Modern C++ Partha Pratim Das M35.3

In the present module, we are going to discuss about multiple inheritance. We have mentioned about it from time to time occasionally, but now that we are empowered with the complete knowledge of polymorphism, particularly the runtime type detection, the dynamic typing and so on, on a polymorphic class hierarchy, we are able to handle multiple inheritance.

(Refer Slide Time: 1:38)

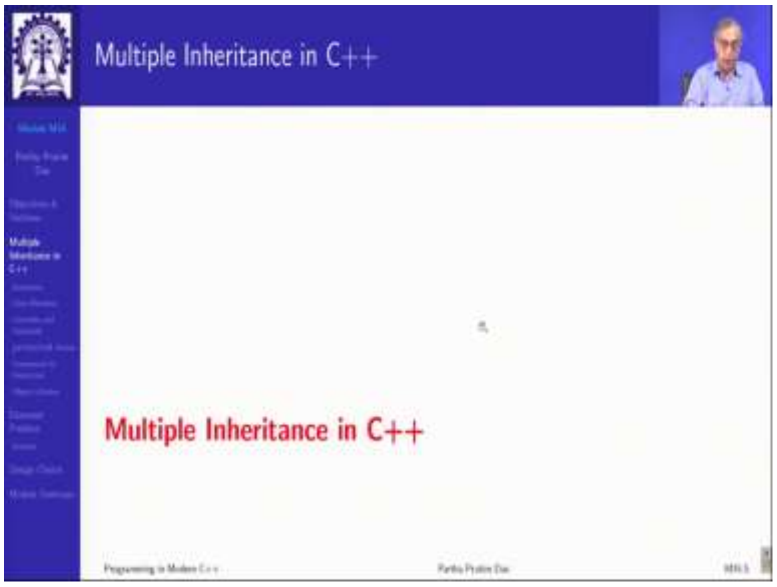


Module Outline

1. Multiple Inheritance in C++
   - Semantics
   - Data Members and Object Layout
   - Member Functions – Overrides and Overloads
   - Access Members of Base: protected Access
   - Constructor & Destructor
   - Object Lifetime
2. Diamond Problem
   - Exercise
3. Design Choice
4. Module Summary

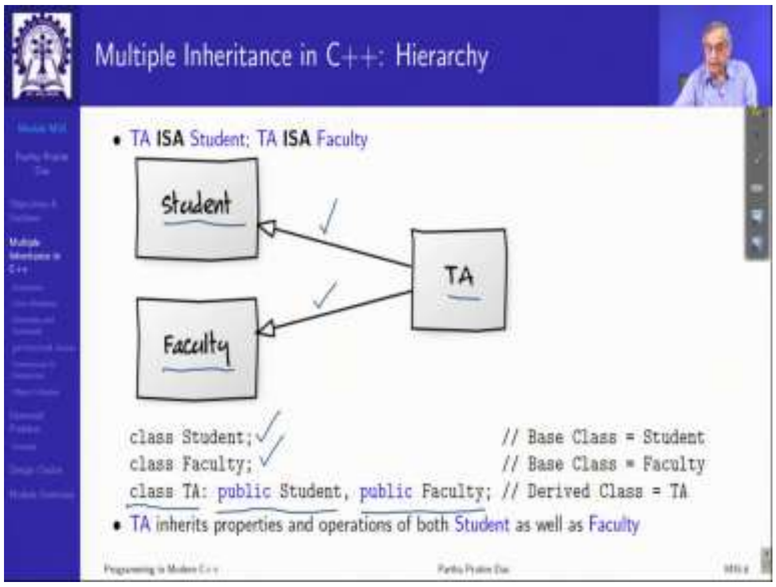
So, we will introduce the notion and see what are the consequences of that and how does C++ empower us to handle those, the outline is given here, which will be on the left all the time.

(Refer Slide Time: 1:48)



Multiple Inheritance in C++

Multiple Inheritance in C++



Multiple Inheritance in C++: Hierarchy

- TA **ISA** Student; TA **ISA** Faculty

```
class Student;                                // Base Class = Student
class Faculty;                                // Base Class = Faculty
class TA: public Student, public Faculty;     // Derived Class = TA
```

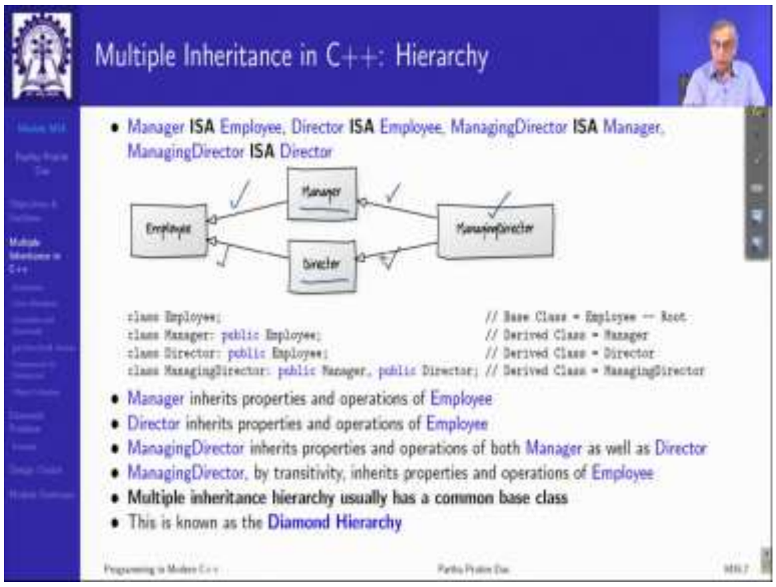
- TA inherits properties and operations of both Student as well as Faculty

So, what is multiple inheritance? As the name suggests, multiple inheritance is occurring when in a ISA hierarchy a class has more than one base classes, more than one classes that it is specializing from. So, consider the case of students, faculty and the teaching assistant, TA stands for teaching assistant, now, what is the role of a teaching assistant? Teaching assistant necessarily is a student. So, that justifies this specialization, the teaching assistant has roll number, has a year of enrollment, course of study and so on.

At the same time, the teaching assistant is evaluating the different assignments in the courses they are taking part in the class tests, maybe setting some of the question papers for students and so on. So, the teaching assistant is actually also playing certain roles that the faculty would normally play. A teaching assistant occasionally may even deliver a few special lectures.

So, when we have to model this, we have a base class Student, a base class Faculty and when you show this inheritance, we will have to put both these base classes on the inheritance line. So, public Student and public Faculty the derived class will inherit from both of them, that is the basic idea of multiple inheritance.

(Refer Slide Time: 3:30)



Multiple Inheritance in C++: Hierarchy

- Manager **ISA** Employee, Director **ISA** Employee, ManagingDirector **ISA** Manager, ManagingDirector **ISA** Director

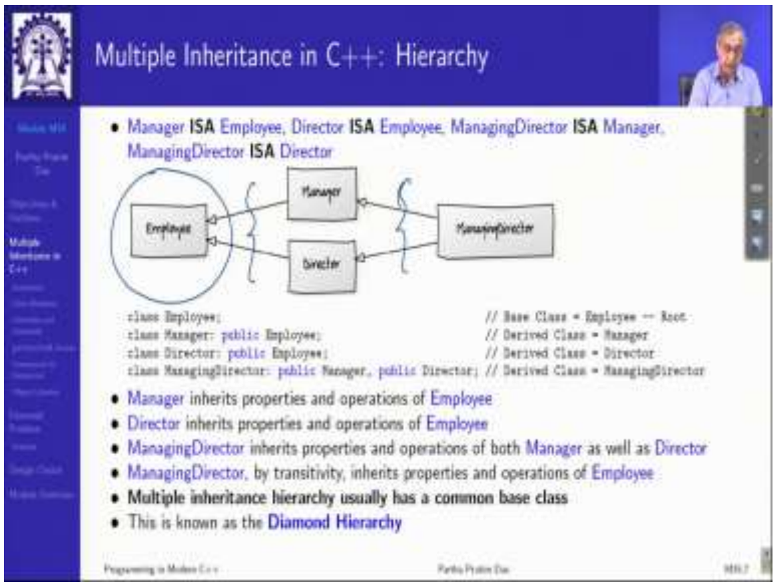
```
class Employee;                                          // Base Class = Employee -- Root
class Manager: public Employee;                          // Derived Class = Manager
class Director: public Employee;                         // Derived Class = Director
class ManagingDirector: public Manager, public Director; // Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee
- Director inherits properties and operations of Employee
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- **Multiple inheritance hierarchy usually has a common base class**
- This is known as the **Diamond Hierarchy**

Programming in Modern C++ Partha Pratim Das M16.7

We can see a refinement of this idea say in the scenario of employees and managers and directors and managing directors. So, let us look at if we have employees who are managers and who are directors, each one of them is an employee in some way, it is obligated to the company get salary and so on so forth. Now, one or more specific persons could be managing director. So, he or she is a manager at the same time is a director, manager in the role that he or she disperse director in the way that sitting in the board of the company deciding on strategic matters and so on, so forth.

(Refer Slide Time: 4:22)



Multiple Inheritance in C++: Hierarchy

- Manager **ISA** Employee, Director **ISA** Employee, ManagingDirector **ISA** Manager, ManagingDirector **ISA** Director

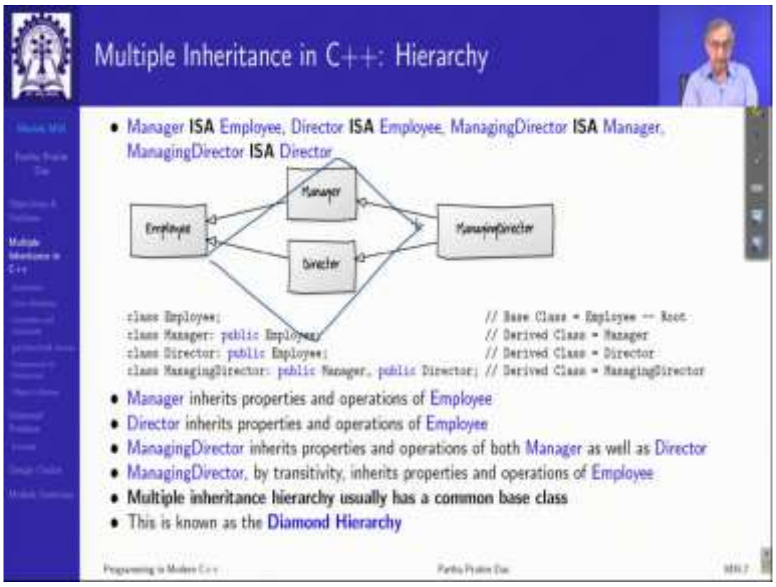
```
class Employee;                                          // Base Class = Employee -- Root
class Manager: public Employee;                          // Derived Class = Manager
class Director: public Employee;                         // Derived Class = Director
class ManagingDirector: public Manager, public Director; // Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee
- Director inherits properties and operations of Employee
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- **Multiple inheritance hierarchy usually has a common base class**
- This is known as the **Diamond Hierarchy**

Programming in Modern C++ Partha Pratim Das M16.7

So, here we see two different things coming out. One is at this point, we have multiple inheritance. Now, when it inherits from two base classes Manager and Director, they in turn, actually inherit from a common base class, that is the whole inheritance hierarchy has one root only, which normally is what we have seen in the case of single multi-level inheritance. But at this point, I have a multiple inheritance.

(Refer Slide Time: 4:54)



Multiple Inheritance in C++: Hierarchy

- Manager **ISA** Employee, Director **ISA** Employee, ManagingDirector **ISA** Manager, ManagingDirector **ISA** Director

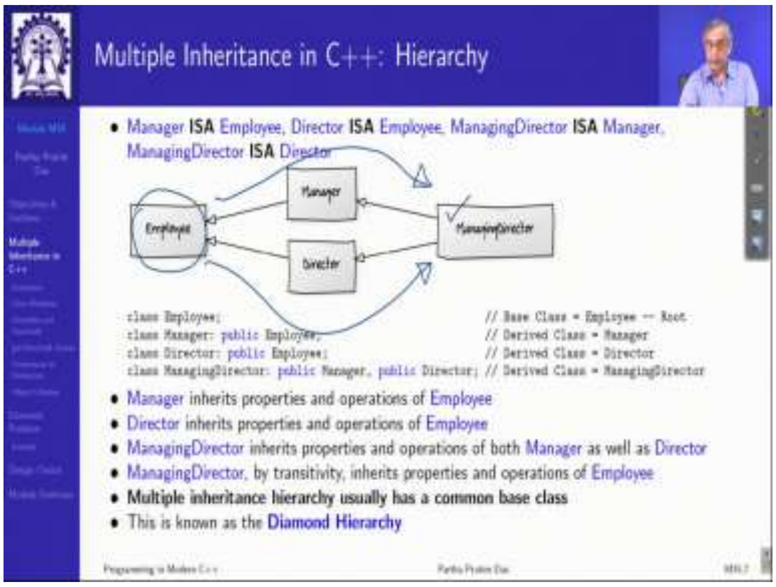
```
class Employee;                                              // Base Class = Employee -- Root
class Manager: public Employee;                              // Derived Class = Manager
class Director: public Employee;                             // Derived Class = Director
class ManagingDirector: public Manager, public Director;     // Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee
- Director inherits properties and operations of Employee
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- **Multiple inheritance hierarchy usually has a common base class**
- This is known as the **Diamond Hierarchy**

Programming in Modern C++ Partha Pratim Das

So, what I get is actually what is known as the diamond hierarchy. So, we get this kind of diamonds structure. So, we said this is a diamond hierarchy where a multiple inheriting from two or more base classes, which in turn has a common base class inheriting from.

(Refer Slide Time: 5:15)



Multiple Inheritance in C++: Hierarchy

- Manager **ISA** Employee, Director **ISA** Employee, ManagingDirector **ISA** Manager, ManagingDirector **ISA** Director

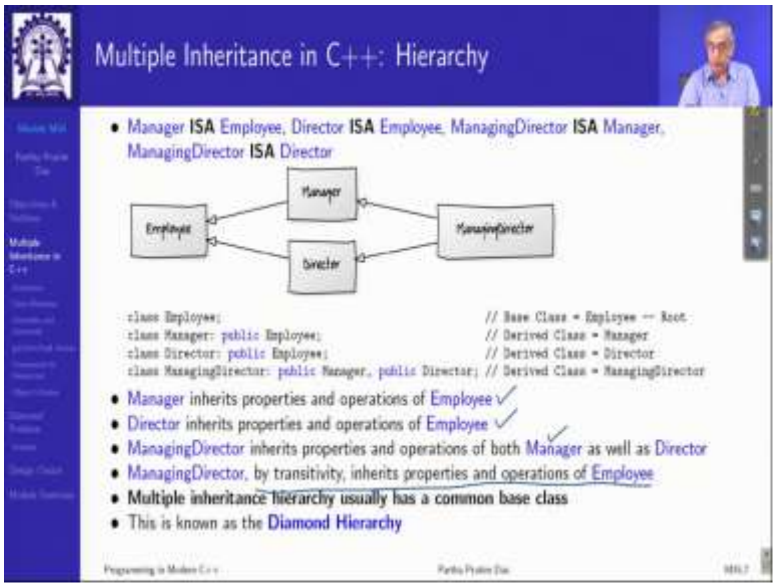
```
class Employee;                                              // Base Class = Employee -- Root
class Manager: public Employee;                              // Derived Class = Manager
class Director: public Employee;                             // Derived Class = Director
class ManagingDirector: public Manager, public Director;     // Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee
- Director inherits properties and operations of Employee
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- **Multiple inheritance hierarchy usually has a common base class**
- This is known as the **Diamond Hierarchy**

Programming in Modern C++ Partha Pratim Das

So, naturally the one question that would come up strongly here is in the Managing Director, what is the identity of the Employee? Is identity flowing through this path of inheritance? Or is the identity flowing through this path of inheritance? We will see what are the consequences of that, but this is just the basic definition of what is the diamond hierarchy or often called the diamond problem also.

(Refer Slide Time: 5:41)



Multiple Inheritance in C++: Hierarchy

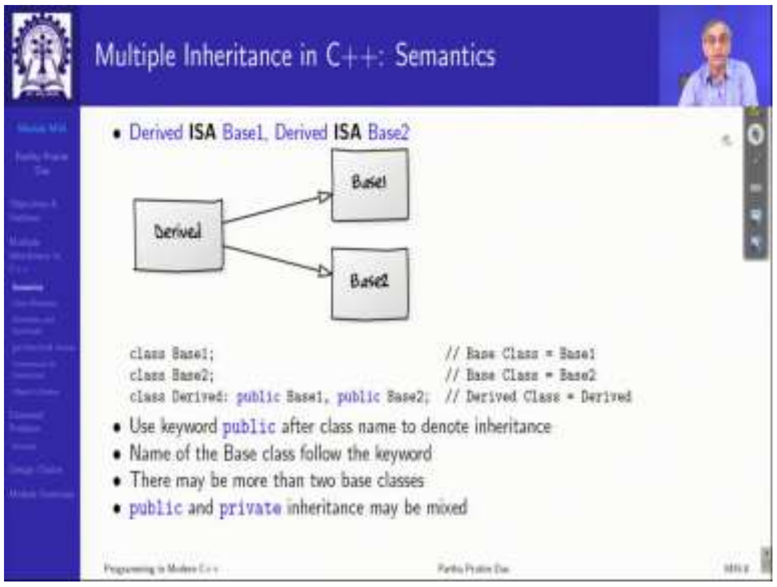
- Manager **ISA** Employee, Director **ISA** Employee, ManagingDirector **ISA** Manager, ManagingDirector **ISA** Director

```
class Employee;                                                   // Base Class = Employee -- Root
class Manager: public Employee;                                   // Derived Class = Manager
class Director: public Employee;                                  // Derived Class = Director
class ManagingDirector: public Manager, public Director;          // Derived Class = ManagingDirector
```

- Manager inherits properties and operations of Employee ✓
- Director inherits properties and operations of Employee ✓
- ManagingDirector inherits properties and operations of both Manager as well as Director
- ManagingDirector, by transitivity, inherits properties and operations of Employee
- **Multiple inheritance hierarchy usually has a common base class**
- This is known as the **Diamond Hierarchy**

So, Manager inherits all properties of Employee, Director does so, Managing Director does for both and Managing Director by transitivity get all the properties and operations of the Employee right. Let us move on.

(Refer Slide Time: 5:59)



Multiple Inheritance in C++: Semantics

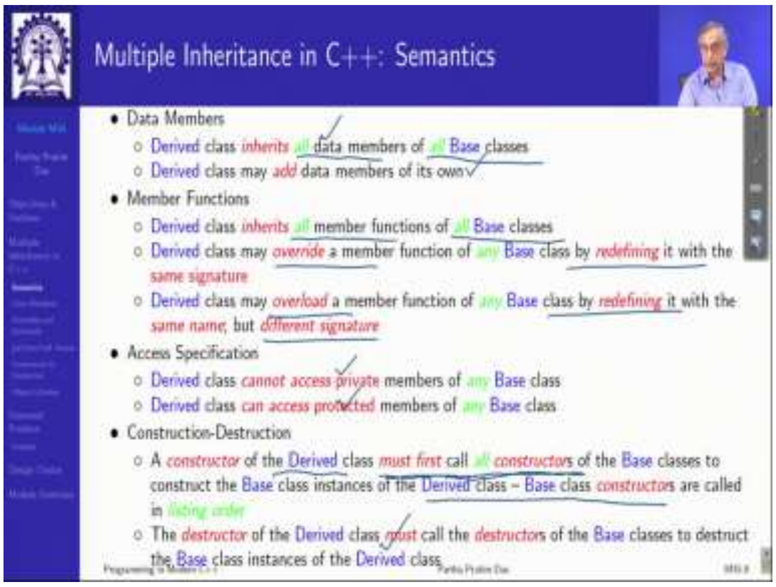
- Derived **ISA** Base1, Derived **ISA** Base2

```
class Base1;                                   // Base Class = Base1
class Base2;                                   // Base Class = Base2
class Derived: public Base1, public Base2;     // Derived Class = Derived
```

- Use keyword `public` after class name to denote inheritance
- Name of the Base class follow the keyword
- There may be more than two base classes
- `public` and `private` inheritance may be mixed

So, let me now with a abstract example, show you the basic rules of multiple inheritance, what is the semantics? So, I have, we have two base classes base 1 and base 2 and the derived class which is inheriting from both of them. Inheritance as we have observed is public and that is what we will continue to use, but it is possible that one or more of these multiple inheritances could be private as well, they may be mixed in in any way. But usually as the inheritance always tell us that it is public inheritance.

(Refer Slide Time: 6:40)



Multiple Inheritance in C++: Semantics

- Data Members
  - Derived class *inherits* all data members of all Base classes
  - Derived class may *add* data members of its own
- Member Functions
  - Derived class *inherits* all member functions of all Base classes
  - Derived class may *override* a member function of any Base class by *redefining* it with the *same signature*
  - Derived class may *overload* a member function of any Base class by *redefining* it with the *same name*, but *different signature*
- Access Specification
  - Derived class *cannot access private* members of any Base class
  - Derived class *can access protected* members of any Base class
- Construction-Destruction
  - A *constructor* of the Derived class *must first* call *all constructors* of the Base classes to construct the Base class instances of the Derived class – Base class constructors are called in *listing order*
  - The *destructor* of the Derived class *must* call the *destructors* of the Base classes to destruct the Base class instances of the Derived class

Programming in Modern C++ Partha Pratim Das M45.8

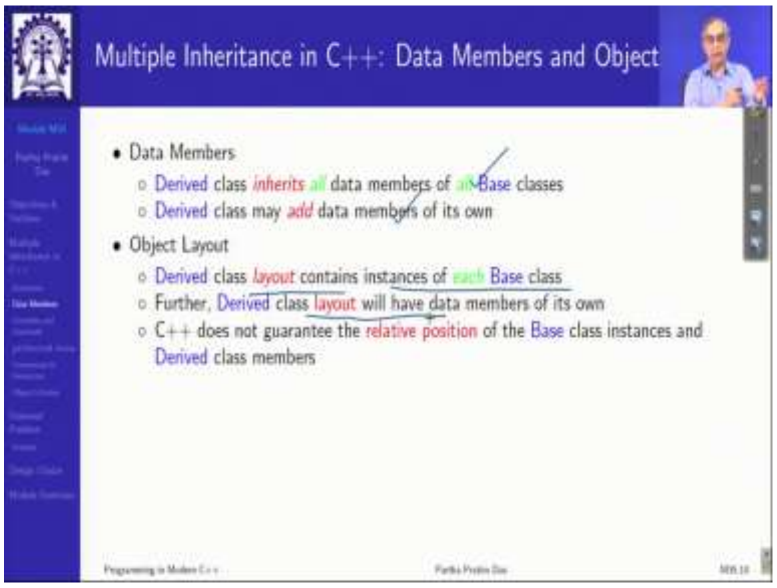
So, the basic rules, naturally these rules are extensions and in some cases modifications of the rules for single inheritance. So, what we have is naturally the derived class inherits all data members. Now, of all base classes because there is more than one base class, and it can add its own, that is that part remains the same quite obvious.

It inherits all member functions of all base classes, it can override a member function by redefining it with the same signature, it can overload the member function of any base class by redefining it with the same name, but different signature. So, all these were already there, now, it can be done for all the base classes of a class, that it is multiple inheriting from.

Access specifications remain the same, that is you the derived class cannot access the members of the base class which are private, but it can access the members of the base class which are protected. For construction, the of the derived class it must first call all constructors of all base classes. Earlier there was only one base class we can construct that, now, it is quite obvious that there are multiple base classes.

So, all of those constructors of the base classes must be called and the base instances have to be created within the object before the base derived class can be constructed. And these constructors will be called in the order in which they are listed on the inheritance listing list that we provide. The destructors as usual will be called all of them will have to be called for the base as well and they will be called by the same rule that the destructor will be that destructor will be called first whose constructor was called last. So, we will proceed in that manner. These are the basic laying down rules.

(Refer Slide Time: 8:47)



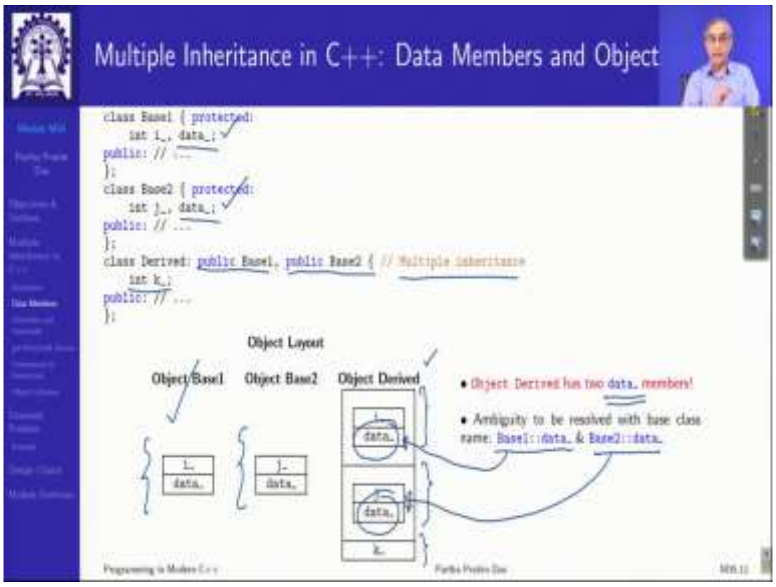
Multiple Inheritance in C++: Data Members and Object

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

Now, let us look each one of these categories of items one by one. So, you first focus on the data members and the object layout, we have studied the object layout for inherited objects for simple inheritance. So, it will inherit all data members of all base classes can add members, we have talked off, the layout contents instances of each base class, there are multiple base class earlier there was one, so, instance of that was there, now all of them we will have to do there. and then it will also have the layout of its own class.

Now, C++ unfortunately does not guarantee the relative position of the base class instances and the derived class member, it does not say that first base class instance, second base class instance and so on then the derived class instance this will be followed, it may be that you know derived class instance might come somewhere in between them it is you know implementation dependent feature. So, always while I mean we should not normally need to look at, look inside the layout, but when you think about it, we should not think about any particular order in which they will happen, only thing is every instance of a base class will be together.

(Refer Slide Time: 10:09)



Multiple Inheritance in C++: Data Members and Object

```
class Base1 { protected:
    int i_, data_;
public: // ...
};
class Base2 { protected:
    int j_, data_;
public: // ...
};
class Derived: public Base1, public Base2 { // Multiple inheritance
    int k_;
public: // ...
};
```

Object Layout

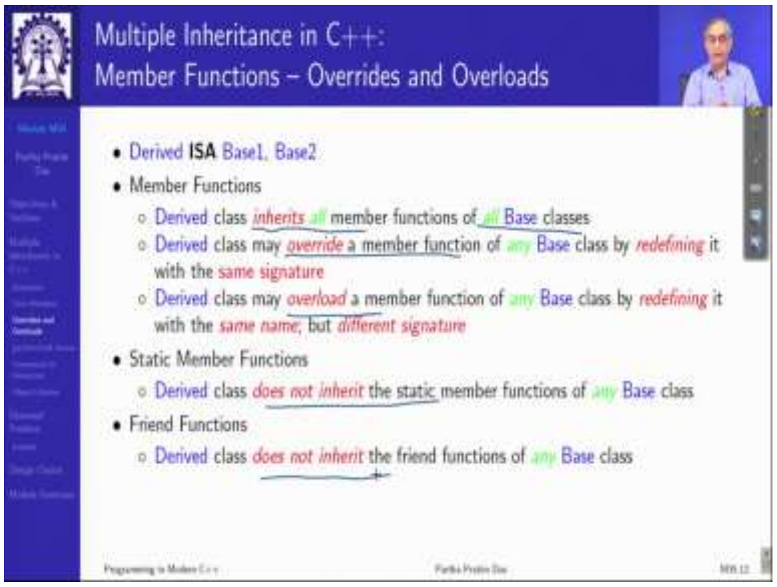
- Object Derived has two data_ members!
- Ambiguity to be resolved with base class name: Base1::data_ & Base2::data_

So, here is an example, I have a base class Base1 which has two data members int data members i_ and data_. Base2 has another base class which has two data members j_ and data_ and Derived class derives from both of them and multiple inheritance and add its own member. So, if I have a base class 1 instance, it will have clearly i and data, if I have base class 2 instance it will have j and data and if we have derived class instance it will have a base 1 instance, it will have a base 2 instance and it will have its own data members.

Now, this relative order is as I said is not known, in general is not specified in general, the compiler will decide that. Now, the interesting factor to note here is there are two data members coming from two base classes, both of whose name is data. So, as a derived class object how do we refer to this data member. So, what the data member has to do? What the derived class has to do? The directors object has to do, is to qualify that which data member it is it wants to access.

So, it cannot just access it as data_ this one it will have to access as Base1::data_ and this as Base2::data_ this is the added complication that comes up because you have multiple base classes and obviously, they can have members which are common in the name but they are actually instance wise they are distinct and both of them will exist in the derived class object.

(Refer Slide Time: 12:07)



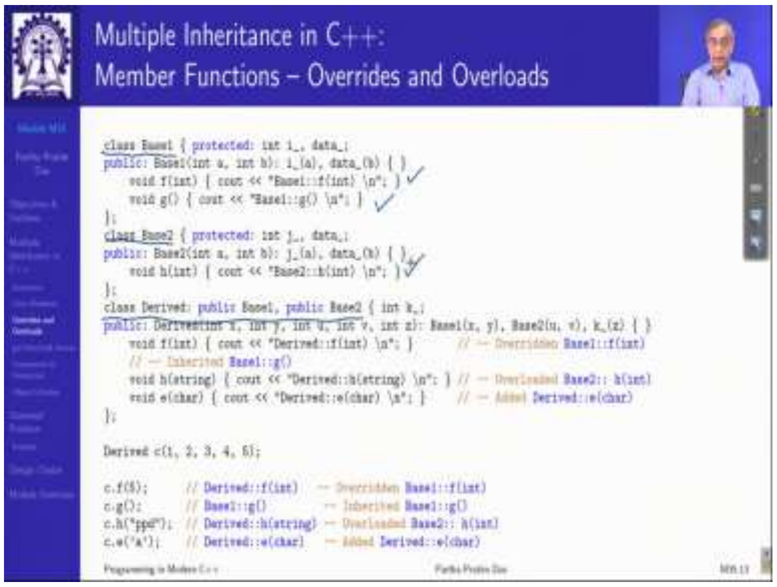
Multiple Inheritance in C++: Member Functions – Overrides and Overloads

- Derived **ISA** Base1, Base2
- Member Functions
  - Derived class inherits all member functions of all Base classes
  - Derived class may override a member function of any Base class by redefining it with the same signature
  - Derived class may overload a member function of any Base class by redefining it with the same name, but different signature
- Static Member Functions
  - Derived class does not inherit the static member functions of any Base class
- Friend Functions
  - Derived class does not inherit the friend functions of any Base class

Programming in Modern C++ Partha Pratim Das M18.12

Let us move on to the member functions. So, how does the override and overloads work? Naturally all member functions of all base classes are inherited, they can be overwritten, they can be overloaded by redefinition. Static member functions are not inherited as you know, so, there is nothing special in terms of multiple inheritance here neither does for the friend functions. So, we will not have any other special discussion about them, because as you know that they have no role in the inheritance process because they are all statically bound.

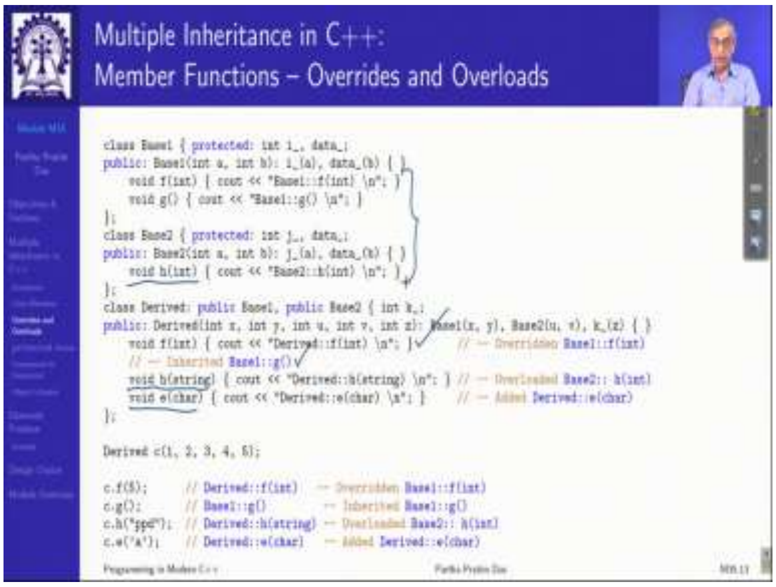
(Refer Slide Time: 12:51)



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

```
class Base1 { protected: int i_, data_;
public: Base1(int a, int b): i_(a), data_(b) { }
    void f(int) { cout << "Base1::f(int) \n"; }
    void g() { cout << "Base1::g() \n"; }
};
class Base2 { protected: int j_, data_;
public: Base2(int a, int b): j_(a), data_(b) { }
    void h(int) { cout << "Base2::h(int) \n"; }
};
class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int u, int v, int z): Base1(x, y), Base2(u, v), k_(z) { }
    void f(int) { cout << "Derived::f(int) \n"; }        // -- Overridden Base1::f(int)
    // -- Inherited Base1::g()
    void h(string) { cout << "Derived::h(string) \n"; } // -- Overloaded Base2:: h(int)
    void e(char) { cout << "Derived::e(char) \n"; }      // -- Added Derived::e(char)
};

Derived c(1, 2, 3, 4, 5);

c.f(5);      // Derived::f(int)    -- Overridden Base1::f(int)
c.g();       // Base1::g()          -- Inherited Base1::g()
c.h("ppd");  // Derived::h(string) -- Overloaded Base2:: h(int)
c.e('a');    // Derived::e(char)   -- Added Derived::e(char)
```

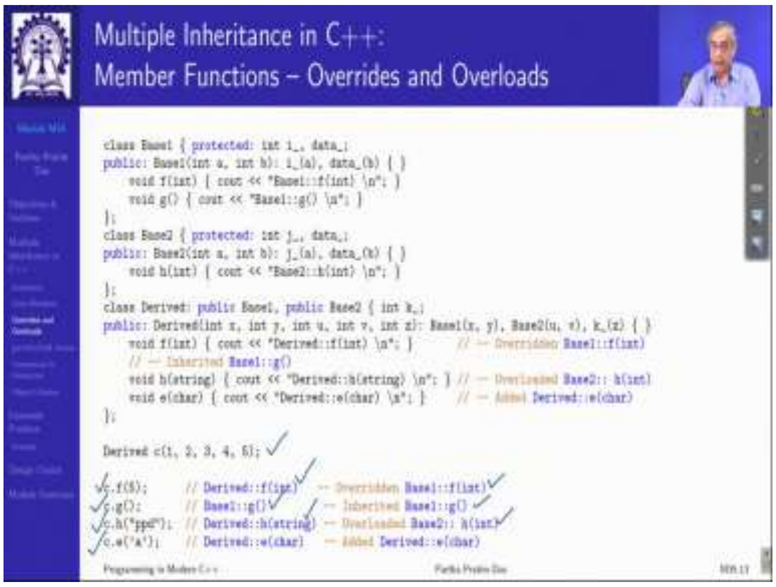
Programming in Modern C++ Partha Pratim Das M18.13



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

```
class Base1 { protected: int i_, data_;
public: Base1(int a, int b): i_(a), data_(b) { }
    void f(int) { cout << "Base1::f(int) \n"; }
    void g() { cout << "Base1::g() \n"; }
};
class Base2 { protected: int j_, data_;
public: Base2(int a, int b): j_(a), data_(b) { }
    void h(int) { cout << "Base2::h(int) \n"; }
};
class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int u, int v, int z): Base1(x, y), Base2(u, v), k_(z) { }
    void f(int) { cout << "Derived::f(int) \n"; }     // -- Overridden Base1::f(int)
    // -- Inherited Base1::g()
    void h(string) { cout << "Derived::h(string) \n"; } // -- Overloaded Base2:: h(int)
    void e(char) { cout << "Derived::e(char) \n"; }    // -- Added Derived::e(char)
};

Derived c(1, 2, 3, 4, 5);

c.f(5);      // Derived::f(int)    -- Overridden Base1::f(int)
c.g();       // Base1::g()         -- Inherited Base1::g()
c.h("ppd");  // Derived::h(string) -- Overloaded Base2:: h(int)
c.e('a');    // Derived::e(char)   -- Added Derived::e(char)
```

Programming in Modern C++ Partha Pratim Das M18.11



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

```
class Base1 { protected: int i_, data_;
public: Base1(int a, int b): i_(a), data_(b) { }
    void f(int) { cout << "Base1::f(int) \n"; }
    void g() { cout << "Base1::g() \n"; }
};
class Base2 { protected: int j_, data_;
public: Base2(int a, int b): j_(a), data_(b) { }
    void h(int) { cout << "Base2::h(int) \n"; }
};
class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int u, int v, int z): Base1(x, y), Base2(u, v), k_(z) { }
    void f(int) { cout << "Derived::f(int) \n"; }     // -- Overridden Base1::f(int)
    // -- Inherited Base1::g()
    void h(string) { cout << "Derived::h(string) \n"; } // -- Overloaded Base2:: h(int)
    void e(char) { cout << "Derived::e(char) \n"; }    // -- Added Derived::e(char)
};

Derived c(1, 2, 3, 4, 5);

c.f(5);      // Derived::f(int)    -- Overridden Base1::f(int)
c.g();       // Base1::g()         -- Inherited Base1::g()
c.h("ppd");  // Derived::h(string) -- Overloaded Base2:: h(int)
c.e('a');    // Derived::e(char)   -- Added Derived::e(char)
```

Programming in Modern C++ Partha Pratim Das M18.11

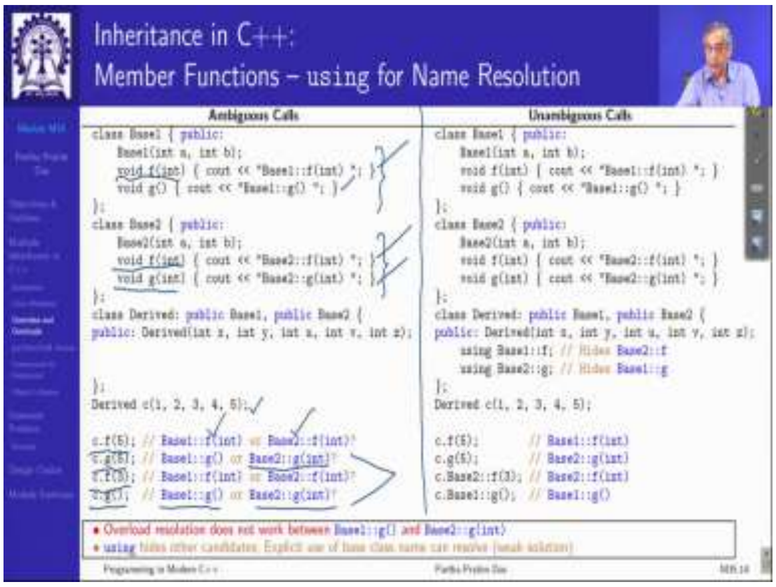
Let us look at an example. We have Base1 we have Base2 and we have Derived class which derives from these two base classes. Base 1 has two methods f and g and Base2 has one method h. Now, let us look at in view of this, let us look at what the derived class is doing. It has put an f which has the same signature as Base1::f, the end signature, so it is actually inheriting and then overriding the function. It has not put any definition for signature for g.

So, it simply inherits Base1::g, it inherits h from Base2, but changes the signature to string. So, it is overloading Base2::h and it has added a new member function. So, this is this example looks very similar to the earlier one except in that all these member functions were in the same base class now they are in two different base classes.

So, in view of this, if I look at a possible derived class object, c, I am not given the details of you know, population of data members and you know, execution those are not really important here. We are just looking at the function binding. So, c.f will be naturally Base1::f because you are doing a static binding, c.g will be Base::Base1::g because you have just inherited it. Here you have got it from derived. So, you have the overridden function which you will be able to call, here it is the same.

If you call c.h with a string, you will get the derived class h function you have overloaded. So, in that process you have also hidden the Base2::h int, we have seen and if we have to make it expose it, we can use the using function we will soon using feature we can immediately see, and actually the added member function can always be invoked. So, this is the basic extension of the simple rules that we had earlier. And we can make use of them to decide which function will be called on which different object.

(Refer Slide Time: 15:51)



Inheritance in C++: Member Functions – using for Name Resolution

Ambiguous Calls

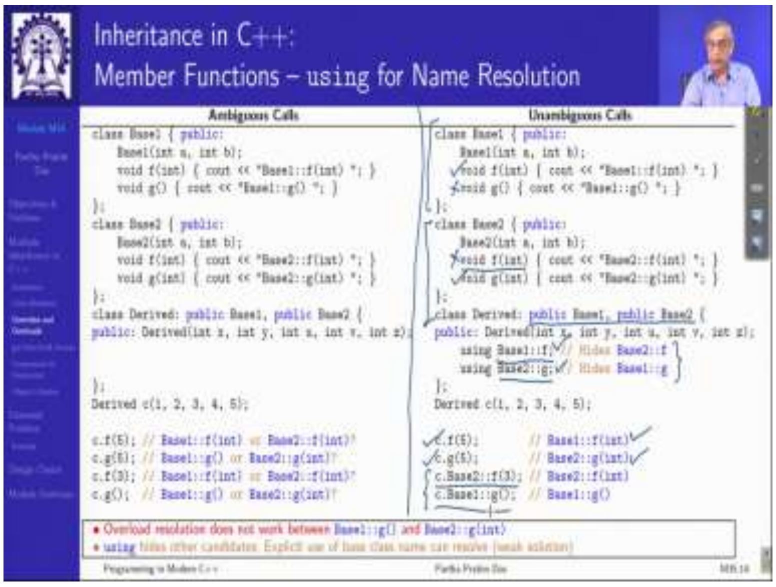
```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);


};
Derived c(1, 2, 3, 4, 5);

c.f(5); // Base1::f(int) or Base2::f(int)?
c.g(5); // Base1::g() or Base2::g(int)?
c.f(3); // Base1::f(int) or Base2::f(int)?
c.g(); // Base1::g() or Base2::g(int)?
```

Unambiguous Calls

```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);
    using Base1::f; // Hides Base2::f
    using Base2::g; // Hides Base1::g
};
Derived c(1, 2, 3, 4, 5);

c.f(5);          // Base1::f(int)
c.g(5);          // Base2::g(int)
c.Base2::f(3);   // Base2::f(int)
c.Base1::g();    // Base1::g()
```

- Overload resolution does not work between Base1::g() and Base2::g(int)
- using hides other candidates. Explicit use of base class name can resolve (weak solution)

Programming in Modern C++ Partha Pratim Das M18.16



Inheritance in C++: Member Functions – using for Name Resolution

Ambiguous Calls

```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);


};
Derived c(1, 2, 3, 4, 5);

c.f(5); // Base1::f(int) or Base2::f(int)?
c.g(5); // Base1::g() or Base2::g(int)?
c.f(3); // Base1::f(int) or Base2::f(int)?
c.g(); // Base1::g() or Base2::g(int)?
```

Unambiguous Calls

```
class Base1 { public:
    Base1(int a, int b);
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};
class Base2 { public:
    Base2(int a, int b);
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};
class Derived: public Base1, public Base2 {
public: Derived(int x, int y, int u, int v, int z);
    using Base1::f; // Hides Base2::f
    using Base2::g; // Hides Base1::g
};
Derived c(1, 2, 3, 4, 5);

c.f(5);          // Base1::f(int)
c.g(5);          // Base2::g(int)
c.Base2::f(3);   // Base2::f(int)
c.Base1::g();    // Base1::g()
```

- Overload resolution does not work between Base1::g() and Base2::g(int)
- using hides other candidates. Explicit use of base class name can resolve (weak solution)

Programming in Modern C++ Partha Pratim Das M18.16

Now, let me complicate this example. There is a mental line here, on the left, I show how the base classes may have different members so that your whole resolution process may get confused. What we did in the last example, though, we had two base classes, their member functions were having independent names. But now, we assume that they have common names. And if you look at f from Base1 and Base2, they have same signature. If you look at g from Base1 and Base2, they have different signature.

Under this scenario. If I look at a derived class object, which has inherited from here, I am not talking about any overloading overriding anything, just trying to look at the semantics of what do you inherit. Now, if I do c.f, then the question is which function Base1 f or Base2 f, which one am I talking of? We do not know, there are both functions have been inherited in the derived, so the compiler is confused.

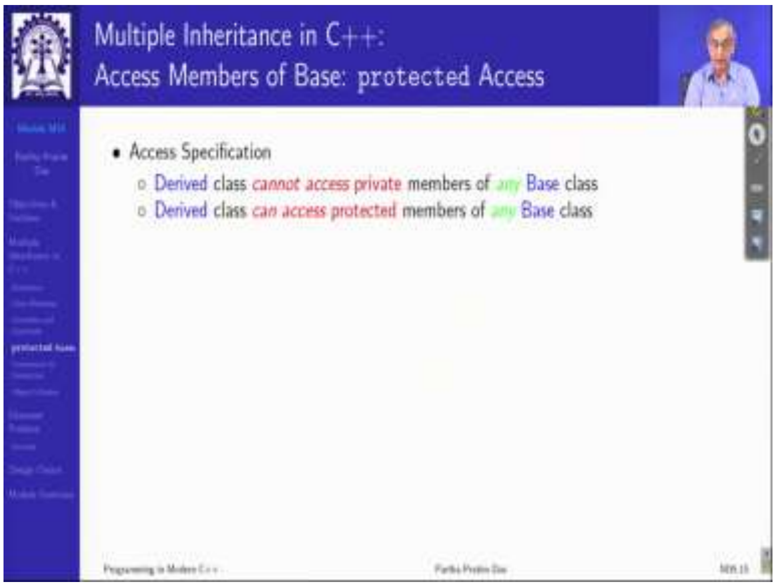
Similarly, if I do c.g as 5, then well, you would say that because it is 5, I am possibly using this, but there is some confusion with g as well, because g is also inherited inside the derived. So, in these two cases, will the compiler be able to resolve rightly just based on the fact that the signatures are different? So, you have two g functions in derived which are not overloads in derived, but they are kind of coming in the form of overload from the two bases.

Finally, if I have f c.f 3, we will again have that kind of confusion between Base1 and Base2. These are not I mean kind of exhaustive list, but these are indicative, which show you what are the possible confusions that can happen. So, on the right, I tried to disambiguate this, the I do not change the classes, neither the inheritance, now multiple inheritance, but I specify, since I have two f functions, I specify which f function I am going to use.

So, I say that I am going to use Base1::f, which will hide Base2::f, it will hide this one, it will allow this one. Similarly, I would say that I am going to use the g function of Base2. So, I am going to use this and therefore I will hide this one. So, now if I do c.f, then I will call Base1::f because I am using that, if I do c.g, I will get Base2::g, because I am using that, the other two functions are hidden, but mind you they can still be called by fully qualified with the name of the base class that if I want to call the function f of Base2 I can still do that by qualifying that function not as f but as Base2::f, same thing I can do for g.

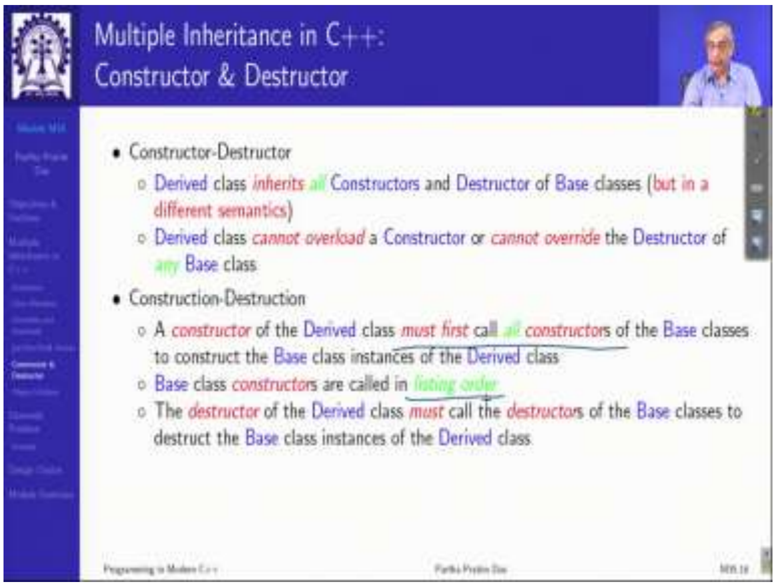
You have seen this in terms of the general using the use of using feature here that is just extended and it takes care of. So, in the design, you will have to take care of the fact that if you are having two functions coming from the two base classes, which have the same name, you have to make sure which is your default choice for the derived class as an inherited function, and then you can override overload that as you want.

(Refer Slide Time: 20:25)



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

- Access Specification
  - Derived class cannot access private members of any Base class
  - Derived class can access protected members of any Base class



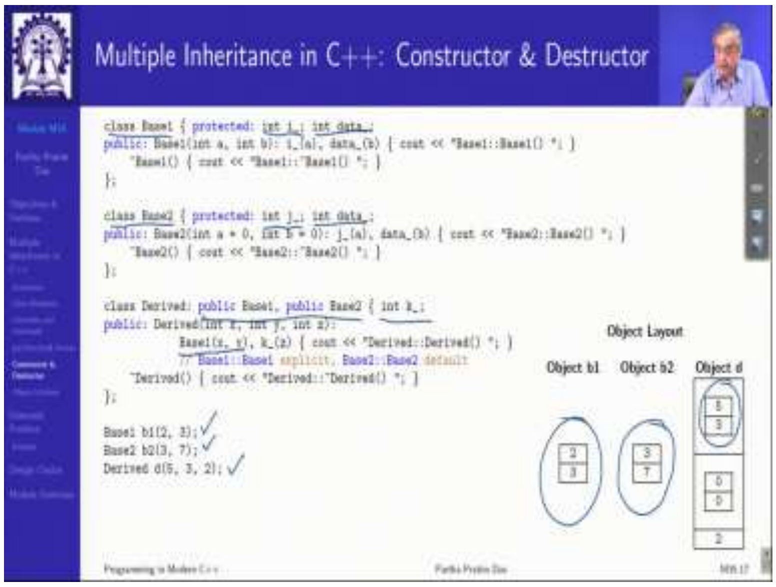
Multiple Inheritance in C++: Constructor & Destructor

- Constructor-Destructor
  - Derived class inherits all Constructors and Destructor of Base classes (but in a different semantics)
  - Derived class cannot overload a Constructor or cannot override the Destructor of any Base class
- Construction-Destruction
  - A constructor of the Derived class must first call all constructors of the Base classes to construct the Base class instances of the Derived class
  - Base class constructors are called in listing order
  - The destructor of the Derived class must call the destructors of the Base classes to destruct the Base class instances of the Derived class

Now, in terms of access specification, it is simple that private members of base classes cannot be accessed, protected members can be accessed by the derived class, there is nothing new. In terms of construction, destruction, derived class will inherit all constructors, it should be able to call on constructors, but it may be in a different semantics because it has to build that object inside its own space. It cannot overload a constructor or override a destructor of any base class, you already understand that.

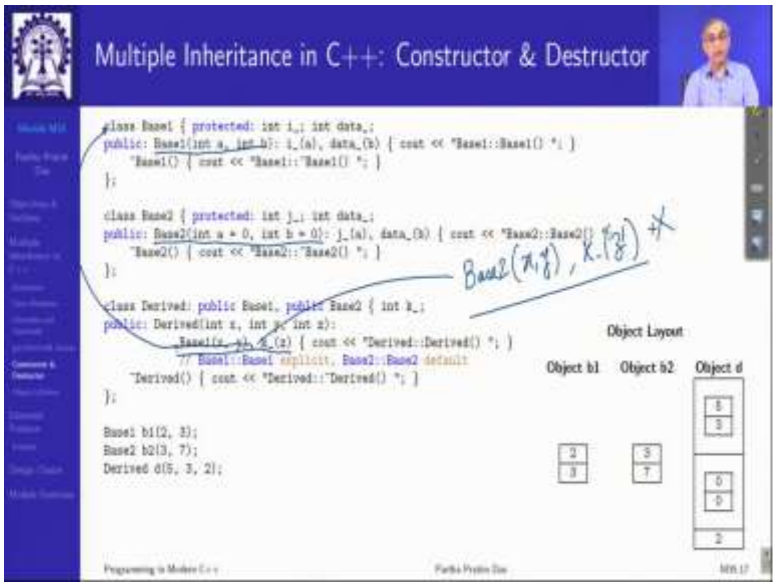
In the process, the constructor of the derived class must first call all constructors of all base classes to create the base class instances, and then the derived class instance will be created they will be called in the listing order I mentioned. And the destructors will be called in the reverse order of construction.

(Refer Slide Time: 21:26)



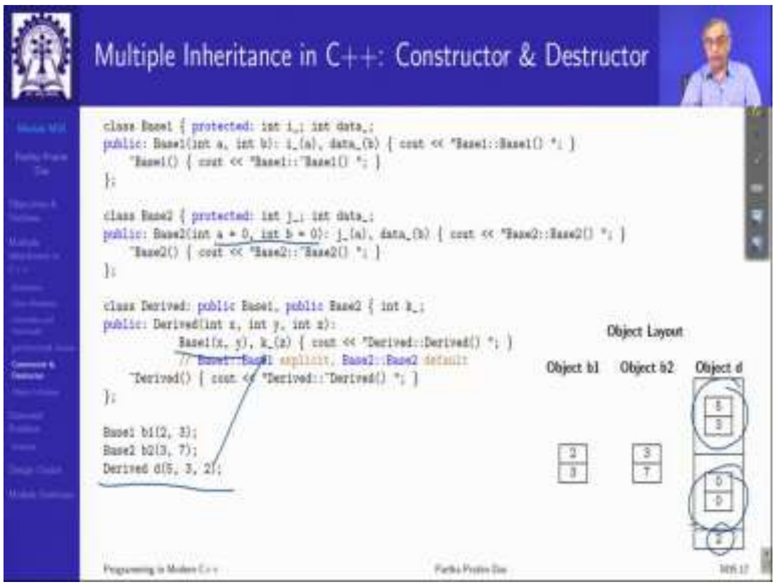
Multiple Inheritance in C++: Constructor & Destructor

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b) { cout << "Base1::Base1() "; }
    ~Base1() { cout << "Base1::~Base1() "; }
};

class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b) { cout << "Base2::Base2() "; }
    ~Base2() { cout << "Base2::~Base2() "; }
};

class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int z):
            Base1(x, y), k_(z) { cout << "Derived::Derived() "; }
            // Base1::Base1 explicit, Base2::Base2 default
    ~Derived() { cout << "Derived::~Derived() "; }
};

Base1 b1(2, 3);
Base2 b2(3, 7);
Derived d(5, 3, 2);
```



Multiple Inheritance in C++: Constructor & Destructor

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b) { cout << "Base1::Base1() "; }
    ~Base1() { cout << "Base1::~Base1() "; }
};

class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b) { cout << "Base2::Base2() "; }
    ~Base2() { cout << "Base2::~Base2() "; }
};

class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int z):
            Base1(x, y), k_(z) { cout << "Derived::Derived() "; }
            // Base1::Base1 explicit, Base2::Base2 default
    ~Derived() { cout << "Derived::~Derived() "; }
};

Base1 b1(2, 3);
Base2 b2(3, 7);
Derived d(5, 3, 2);
```



Multiple Inheritance in C++: Constructor & Destructor

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b) { cout << "Base1::Base1() "; }
    ~Base1() { cout << "Base1::~Base1() "; }
};

class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b) { cout << "Base2::Base2() "; }
    ~Base2() { cout << "Base2::~Base2() "; }
};

class Derived: public Base1, public Base2 { int k_;
public: Derived(int x, int y, int z):
            Base1(x, y), k_(z) { cout << "Derived::Derived() "; }
            // Base1::Base1 explicit, Base2::Base2 default
    ~Derived() { cout << "Derived::~Derived() "; }
};

Base1 b1(2, 3);
Base2 b2(3, 7);
Derived d(5, 3, 2);
```

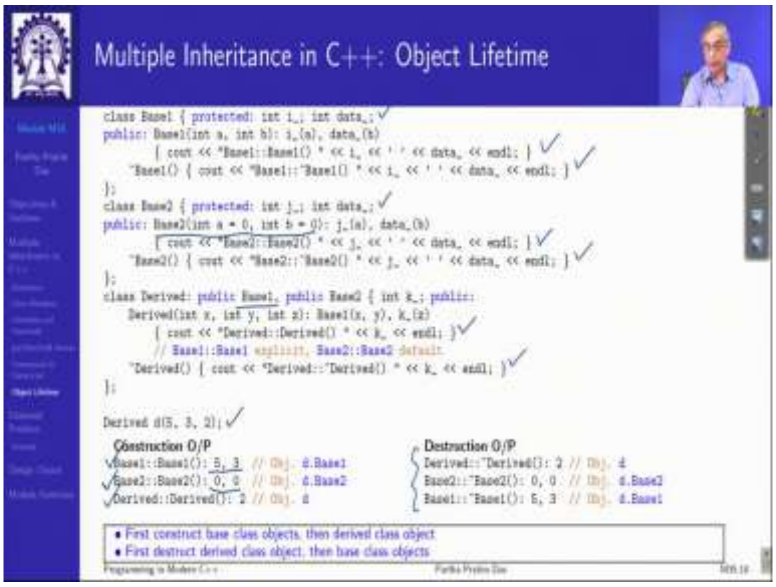
Programming in Modern C++ Partha Pratim Das M35.17

So, if we do that, then we have a Base1, the earlier example int i and data Base2 int j and data here, and I have a multiple inheritance here, which adds another member. So, if I construct a Base1 object, I will get something like this, I do a Base2 object, I will get something like this. And if I do a Derived object, then I will get this Base1. Why I will get the Base1? Because the x y have passed to Base1 that is explicitly constructed Base1.

Whereas if you look into this list, this is an important point needs to be looked at. If you look into this point, the initializer list for the derived class, I have explicitly constructed Base1 have used the non-default constructor of Base1, whereas I have not mentioned anything for Base2, it will also have to get constructed. So, if I do this, then it is mandatory that the base class that I am not mentioning the constructor of the base class that I am not mentioning must have a default constructor. Otherwise, naturally the compiler will give error.

For example, in this situation, in this case, you could not have written this, say you wanted to write base, you wrote Base2 x y and then k underscore z, if you write this, the compiler will give you an error because Base1 if you are skipping Base1, then Base1 needs a default constructor which it does not. So, given that naturally, when I constructor for this, x y will be used to construct Base1, the Base1 object, the Base2 will be constructed by default 0 0, so I get the 0 0 and 2 will go to the added data member. So, it will be here. Again, remember that this order an organization is not something that is standard specified. I have just taken one scheme to do it in the order of the base classes, but some compiler might do it in a different manner. You can check out for your compiler what it does.

(Refer Slide Time: 23:59)



Multiple Inheritance in C++: Object Lifetime

```
class Base1 { protected: int i_; int data_;
public: Base1(int a, int b): i_(a), data_(b)
        { cout << "Base1::Base1() " << i_ << ' ' << data_ << endl; }
    ~Base1() { cout << "Base1::~Base1() " << i_ << ' ' << data_ << endl; }
};
class Base2 { protected: int j_; int data_;
public: Base2(int a = 0, int b = 0): j_(a), data_(b)
        { cout << "Base2::Base2() " << j_ << ' ' << data_ << endl; }
    ~Base2() { cout << "Base2::~Base2() " << j_ << ' ' << data_ << endl; }
};
class Derived: public Base1, public Base2 { int k_; public:
    Derived(int x, int y, int z): Base1(x, y), k_(z)
        { cout << "Derived::Derived() " << k_ << endl; }
        // Base1::Base1 explicit, Base2::Base2 default
    ~Derived() { cout << "Derived::~Derived() " << k_ << endl; }
};

Derived d(5, 3, 2);
```

Construction O/P

```
Base1::Base1(): 5, 3 // Obj. d.Base1
Base2::Base2(): 0, 0 // Obj. d.Base2
Derived::Derived(): 2 // Obj. d
```

Destruction O/P

```
Derived::~Derived(): 2 // Obj. d
Base2::~Base2(): 0, 0 // Obj. d.Base2
Base1::~Base1(): 5, 3 // Obj. d.Base1
```

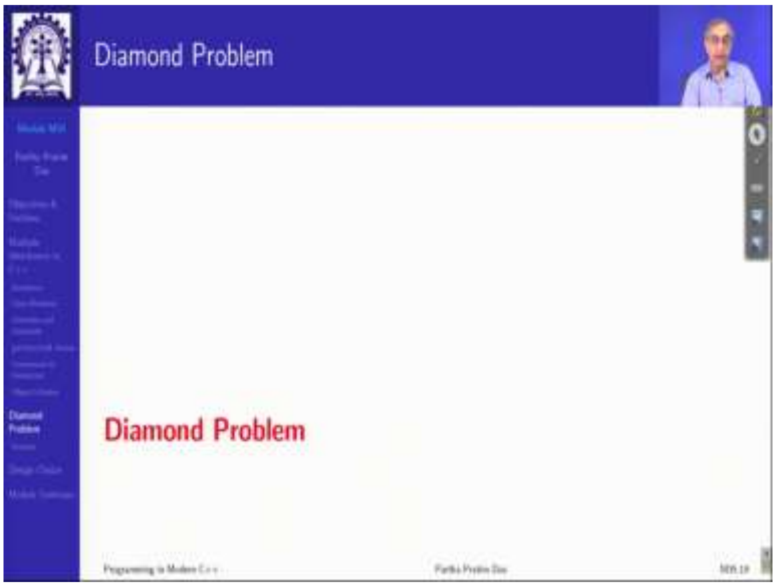
- First construct base class objects, then derived class object
- First destruct derived class object, then base class objects

Programming in Modern C++ Partha Pratim Das M16.18

Now, for the same example, let us look at what will be the object lifetime construction, destruction. So, now along with the data members, we have also provided the constructor, destructor has been provided with cout to actually tell us what is going on. So, when I construct this derived class object naturally first, the Base1 object will have to be constructed. So, you first that constructor gets called and you will get 5, 3 as set to the base 1 part of the instance.

Then naturally based 2 gets called you get 0 0 which is a default constructor that gets set. And finally, the derived class constructor is called which sets 2 into the whole into its own data member, destruction happens in the reverse order as is expected. So, this you can see that except for the order of the base classes the objector construction destruction the lifetime happens exactly following the same rules as we had for the single inheritance.

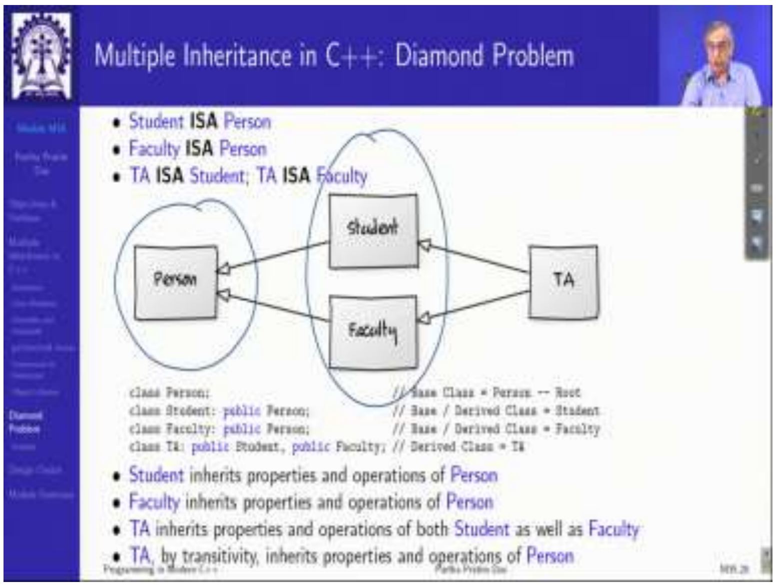
(Refer Slide Time: 25:28)



Diamond Problem

Diamond Problem

Programming in Modern C++ | Partha Pratim Das | M18.19



Multiple Inheritance in C++: Diamond Problem

- Student **ISA** Person
- Faculty **ISA** Person
- TA **ISA** Student; TA **ISA** Faculty

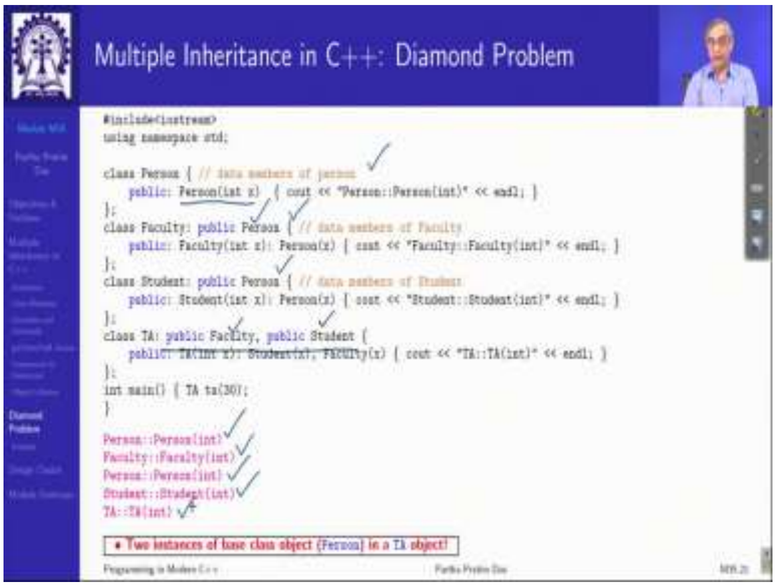
```
class Person;                                  // Base Class = Person -- Root
class Student: public Person;                  // Base / Derived Class = Student
class Faculty: public Person;                  // Base / Derived Class = Faculty
class TA: public Student, public Faculty;      // Derived Class = TA
```

- Student inherits properties and operations of Person
- Faculty inherits properties and operations of Person
- TA inherits properties and operations of both Student as well as Faculty
- TA, by transitivity, inherits properties and operations of Person

Programming in Modern C++ | Partha Pratim Das | M18.20

So, that lays out lays down the basic properties of multiple inheritance. Now, we will go back and look into the diamond problem. What is a diamond problem? Diamond problem is you have a common base class of your base classes that you are deriving from. So, the question is in the object of the class TA, which person will be taken? So, let us see what happens if you just code this.

(Refer Slide Time: 26:00)



Multiple Inheritance in C++: Diamond Problem

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

class Person { // data members of person
    public: Person(int x)  { cout << "Person::Person(int)" << endl; }
};
class Faculty: public Person { // data members of Faculty
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: public Person { // data members of Student
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};
int main() { TA ta(30);
}
```

```
Person::Person(int)
Faculty::Faculty(int)
Person::Person(int)
Student::Student(int)
TA::TA(int)
```

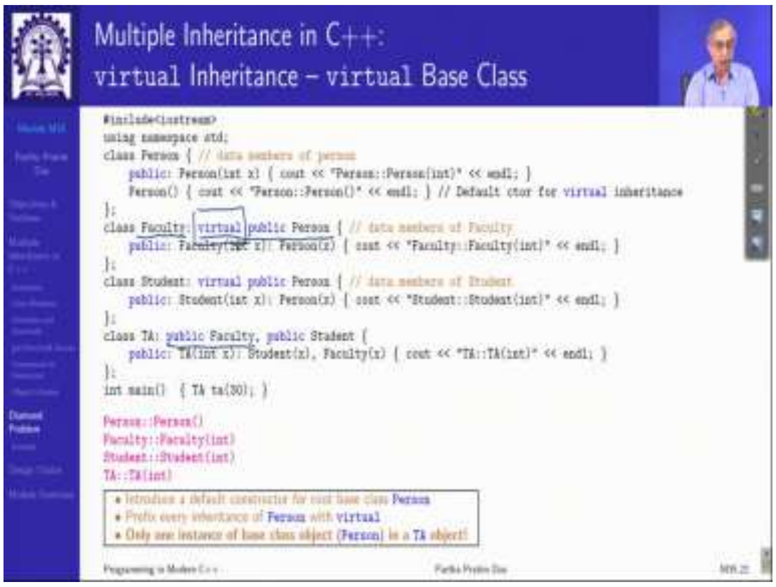
- Two instances of base class object (Person) in a TA object!

Programming in Modern C++ Partha Pratim Das M09.21

If we just do not think about anything just code this part. So, you have a Person which has a parameterized constructor, you have derived Faculty and Student from that and you derive TA from Faculty and Student and just put cout in the constructors, so that you know what is getting constructed. Obviously, Faculty will have to get constructed first for the Faculty, Person will have to get constructed.

So, Person gets constructed, Faculty gets constructed. Then the Student has to get constructed, but the Student also has a base class Person. So, Person is again constructed, Student is constructed, TA is constructed. So, you get into a difficult situation that the same key object has two instances of the Person object, which certainly is not going to be manageable. This is a semantic contradiction that the diamond problem throws up.

(Refer Slide Time: 26:57)



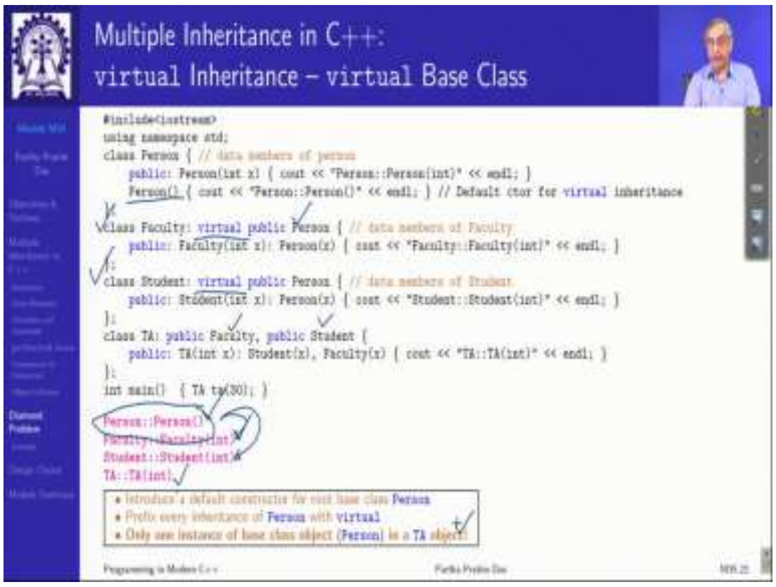
Multiple Inheritance in C++: virtual Inheritance – virtual Base Class

```
#include<iostream>
using namespace std;
class Person { // data members of person
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; } // Default ctor for virtual inheritance
};
class Faculty: virtual public Person { // data members of Faculty
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: virtual public Person { // data members of Student
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};
int main()  { TA ta(30); }
```

```
Person::Person()
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)
```

- Introduce a default constructor for root base class Person
- Prefix every inheritance of Person with virtual
- Only one instance of base class object (Person) in a TA object!

Programming in Modern C++ Partha Pratim Das M09.22



Multiple Inheritance in C++: virtual Inheritance – virtual Base Class

```
#include<iostream>
using namespace std;
class Person { // data members of person
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; } // Default ctor for virtual inheritance
};
class Faculty: virtual public Person { // data members of Faculty
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: virtual public Person { // data members of Student
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x): Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
};
int main()  { TA ta(30); }
```

```
Person::Person()
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int)
```

- Introduce a default constructor for root base class Person
- Prefix every inheritance of Person with virtual
- Only one instance of base class object (Person) in a TA object!

Programming in Modern C++ Partha Pratim Das M09.22

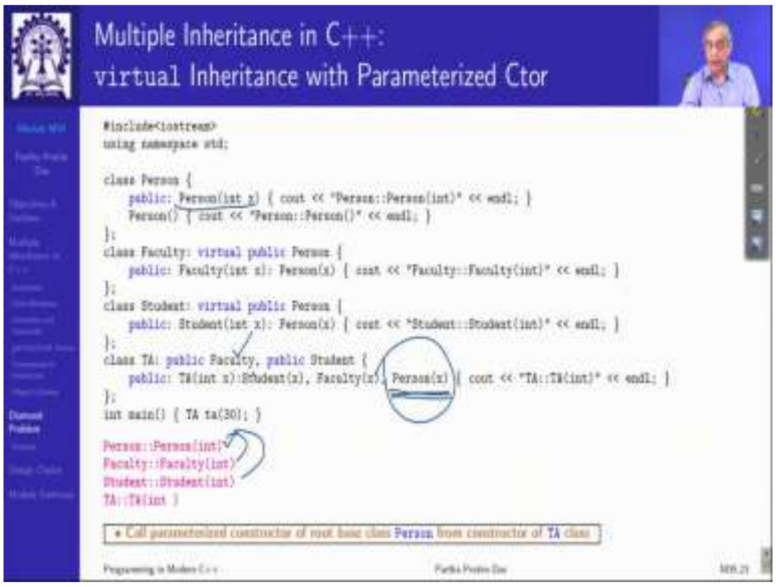
So, to avoid that, what we do is we introduce virtual base classes. What is a virtual base class, a virtual base class is one which you want to use for multiple inheritance from, right, Faculty is a virtual, I want Faculty to be a virtual base class because it needs to be used for derivation multiplying. Now, when Faculty derives from its parent, when Faculty derives from its parent, it uses this key word virtual. The sense of the virtual is absolutely different from the sense of virtual functions, or runtime polymorphism.

What this means is when I and you do the same thing for Student as well, so these are the virtual base classes, and you do the same thing. What I have done is I have added a default constructor in person, which I will explain why I did. Now, if I do the construction Faculty we need to construct Person. So, once Person is constructed, Faculty is constructed, then the Student has to be constructed.

Now, what the Student sees is it is inheriting from Person, but it is virtually inheriting. So, in the context of TA, the compiler will check if an instance of Person has already been created. If it has already been created, it will not be created again, the same instance will be used, unlike the previous case, where two separate instances were being used.

Here, the same instance will be used by both Faculty and Student which solves the basic problem of multiple base classes, base class instances in the derived class object for a diamond configuration. And rest of it is simple. Now, you will have only one instance of TA in the …, only one instance of person in your TA object.

(Refer Slide Time: 29:04)



Multiple Inheritance in C++: virtual Inheritance with Parameterized Ctor

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

class Person {
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; }
};
class Faculty: virtual public Person {
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
};
class Student: virtual public Person {
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
};
class TA: public Faculty, public Student {
    public: TA(int x):Student(x), Faculty(x), Person(x) { cout << "TA::TA(int)" << endl; }
};
int main() { TA ta(30); }
```

```
Person::Person(int)
Faculty::Faculty(int)
Student::Student(int)
TA::TA(int )
```

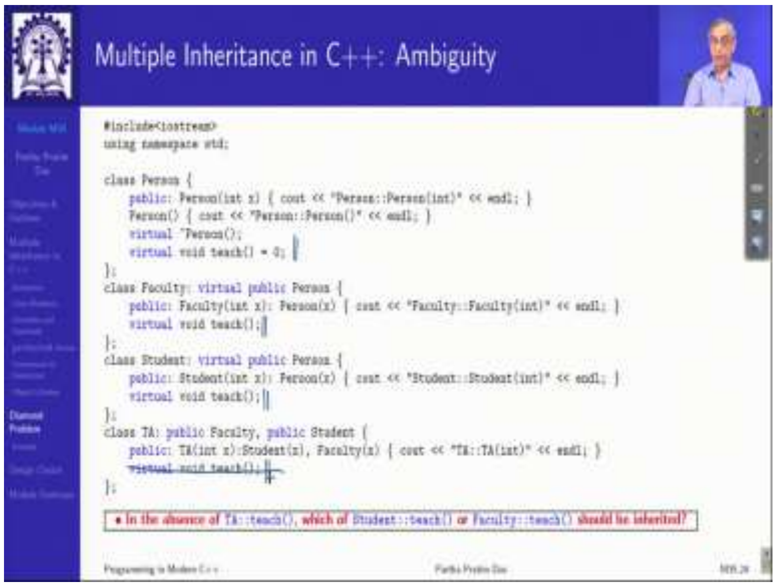
- Call parameterized constructor of root base class Person from constructor of TA class

Programming in Modern C++ Partha Pratim Das M09.21

So, virtual base classes solve that. Naturally I did that through a parameterized, constructor, a default constructor, you can use parameterized constructor also. But if you do that, then your responsibility would be to call that constructor of the route, the Person, so what will that mean that we will construct this by the parameterized constructor and then the Students then the Faculty will be constructed and the Faculty will already find that a Person object exists. It is virtually, it is a virtual base class specializing from Person.

So, once it finds a Person it will not create another Person so the same Person will be used here. But this is the additional that you need to do if you use parameters in your base class root class, then you'll have to explicitly call that and first create that object. So, that, that instance so that that can be subsequently used by the base classes.

(Refer Slide Time: 30:11)



Multiple Inheritance in C++: Ambiguity

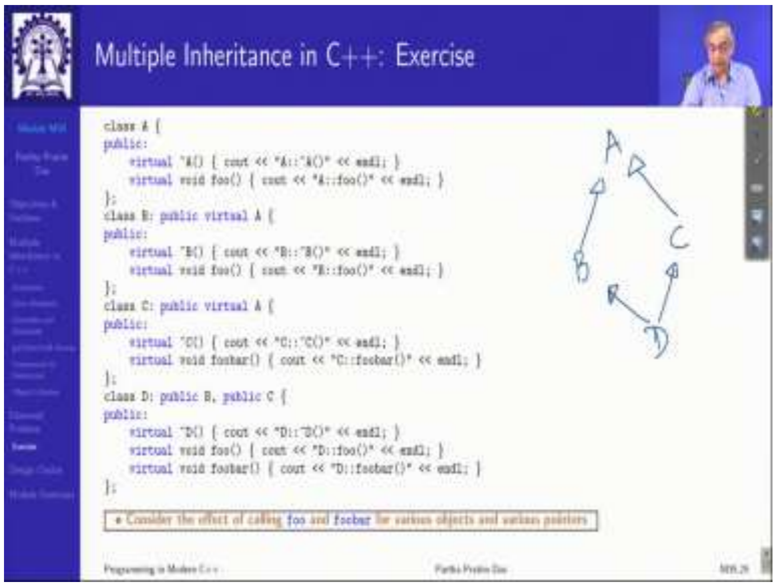
```
#include<iostream>
using namespace std;

class Person {
    public: Person(int x) { cout << "Person::Person(int)" << endl; }
    Person() { cout << "Person::Person()" << endl; }
    virtual ~Person();
    virtual void teach() = 0;
};
class Faculty: virtual public Person {
    public: Faculty(int x): Person(x) { cout << "Faculty::Faculty(int)" << endl; }
    virtual void teach();
};
class Student: virtual public Person {
    public: Student(int x): Person(x) { cout << "Student::Student(int)" << endl; }
    virtual void teach();
};
class TA: public Faculty, public Student {
    public: TA(int x):Student(x), Faculty(x) { cout << "TA::TA(int)" << endl; }
    ~~virtual void teach();~~
};
```

- In the absence of TA::teach(), which of Student::teach() or Faculty::teach() should be inherited?

Programming in Modern C++ Partha Pratim Das M09.28

This is in terms of diamond, there could be other issues to deal with, for example, you have a teach, which you override in Student and Faculty and in TA, fine, good enough it will work. What if I do not provide this? So, the question that I leave with you is, which of the teach will be used? Or how would you resolve that problem. So, it is not easy if you get into diamond, it is not easy to always handle this.

(Refer Slide Time: 30:47)



Multiple Inheritance in C++: Exercise

```
class A {
public:
    virtual ~A() { cout << "A::~A()" << endl; }
    virtual void foo() { cout << "A::foo()" << endl; }
};
class B: public virtual A {
public:
    virtual ~B() { cout << "B::~B()" << endl; }
    virtual void foo() { cout << "B::foo()" << endl; }
};
class C: public virtual A {
public:
    virtual ~C() { cout << "C::~C()" << endl; }
    virtual void foobar() { cout << "C::foobar()" << endl; }
};
class D: public B, public C {
public:
    virtual ~D() { cout << "D::~D()" << endl; }
    virtual void foo() { cout << "D::foo()" << endl; }
    virtual void foobar() { cout << "D::foobar()" << endl; }
};
```

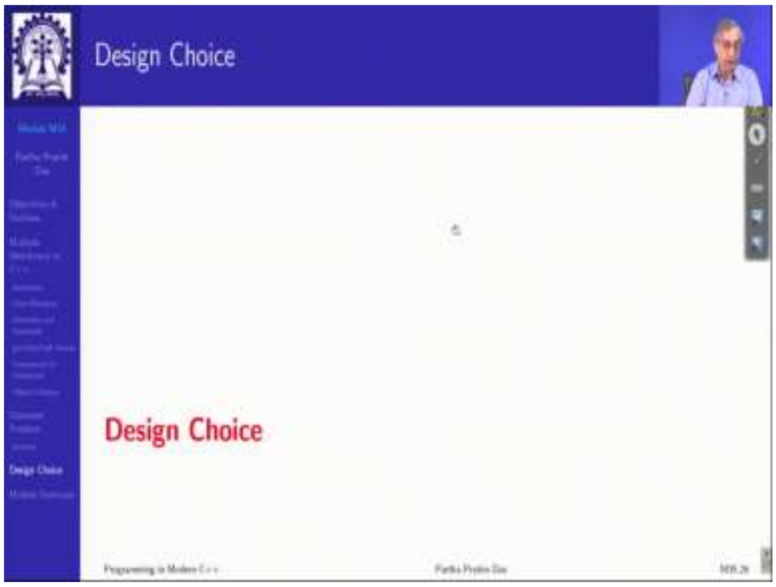
$$\begin{tikzcd} & A & \\ B \arrow[ur] & & C \arrow[ul] \\ & D \arrow[ul] \arrow[ur] & \end{tikzcd}$$

- Consider the effect of calling foo and foobar for various objects and various pointers

Programming in Modern C++ Partha Pratim Das M09.29

Here, I leave you with an exercise. So, where a I have a multiple inheritance scenario, I am sorry. I have a multiple inheritance scenario where you have a diamond and we have different functions foo and foobar in the context of this classes. And you have to create objects A, B, C, D and so on and create different types of pointers and try to access them to see what kind of function binding you actually can get, I have not worked this out for you. We have done it extensively for single inheritance, you should be able to extend that for multiple inheritance.

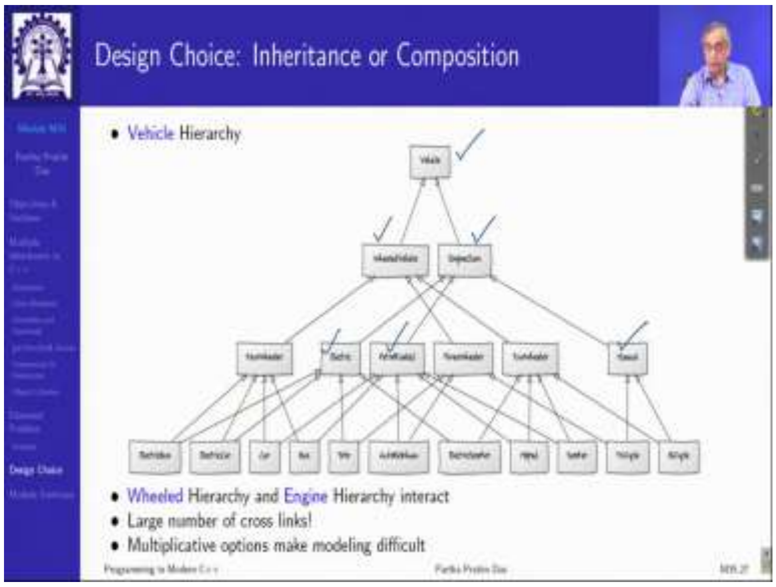
(Refer Slide Time: 31:46)



Design Choice

Design Choice

Programming in Modern C++ Partha Pratim Das M19.26

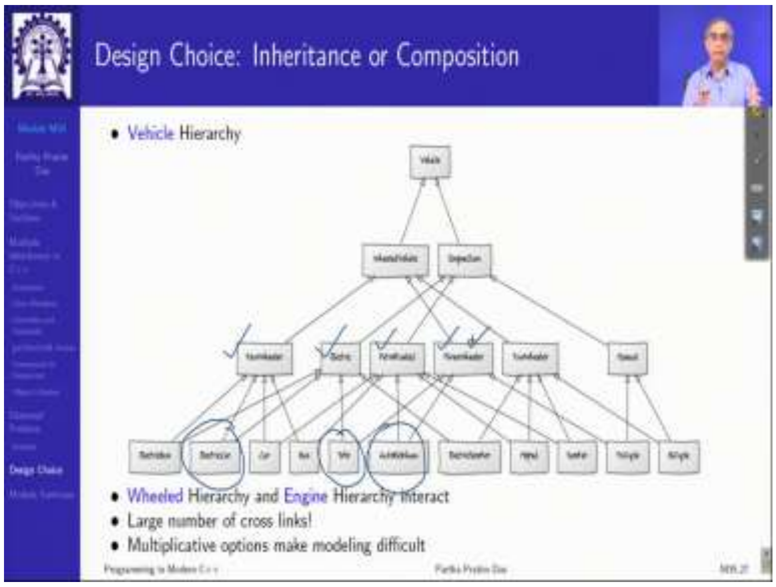


Design Choice: Inheritance or Composition

- Vehicle Hierarchy

- Wheeled Hierarchy and Engine Hierarchy interact
- Large number of cross links!
- Multiplicative options make modeling difficult

Programming in Modern C++ Partha Pratim Das M19.27



Design Choice: Inheritance or Composition

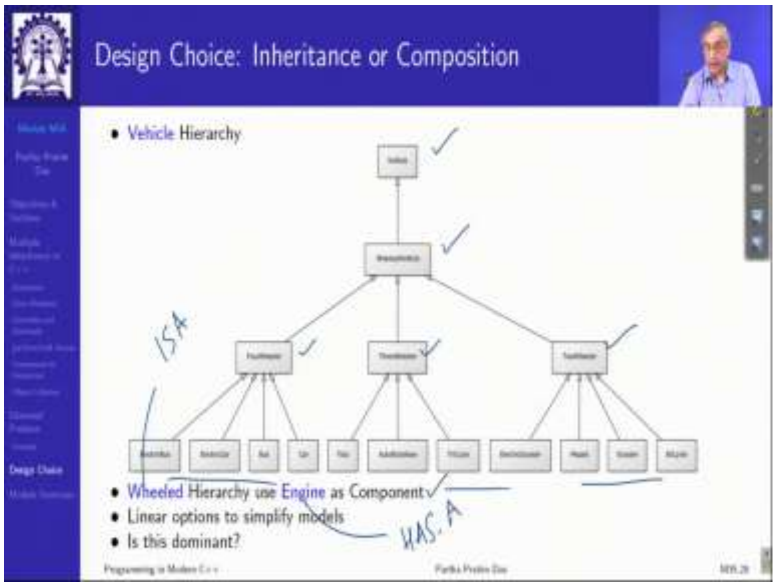
- Vehicle Hierarchy
- Wheeled Hierarchy and Engine Hierarchy Interact
- Large number of cross links!
- Multiplicative options make modeling difficult

Now, having done this, the question is, does multiple inheritance really solve a huge lot of problems? Here is a simple instance of …, well, with this diagram, we will possibly not consider it simple but it is a simple real world. I am talking about vehicles and I am talking about two dominant properties of vehicles. One certainly is of wheels, we have always talked about 4-wheeler 3-wheeler 2-wheeler and based on that a lot of things differ, price, comfort, licensing and so on so forth.

But what has become very dominant for last couple of decades is how it is fueled? What is the engine type? Is it a petrol vehicle, fossil fuel vehicle? Is it manually driven vehicle like a bicycle or is it an electric vehicle? So, if you try to given these two, now, naturally there are several combinations of these properties that can happen.

For example, an electric car is a four wheeler which is electric, if Toto is a three wheeler which is electric a and common auto rickshaw is petrol fueled and his three wheeler. So, you have a mesh of multiple inheritance happening, which naturally make it kind of a mess of understanding.

(Refer Slide Time: 33:19)



Design Choice: Inheritance or Composition

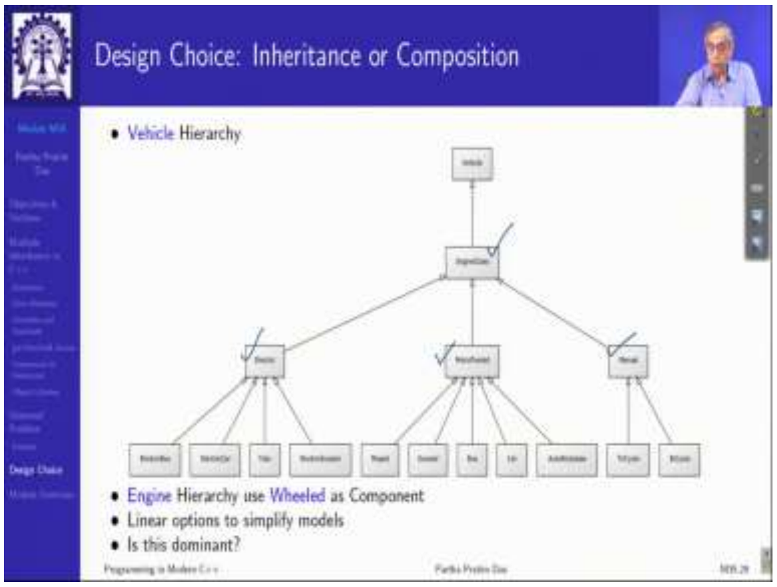
- Vehicle Hierarchy
- Wheeled Hierarchy use Engine as Component
- Linear options to simplify models
- Is this dominant?

Programming in Modern C++ Partha Pratim Das

So, one way we can look at is this modeling is not very suitable to handle because for all of these you have to do virtual base class this that. So, what we suggest is when you have this scenario, choose one property which you consider the most dominant property and have a single inheritance structure based on that, use the other property as a data member, that is as a component. So, engine becomes a component if you create the hierarchy based on the wheels.

So, you have WheeledVehicles, FourWheeler, ThreeWheeler, TwoWheeler, and different instances. And then each one of them has an engine property or composition. So, what I am doing is instead of doing ISA for both, I am doing ISA for wheeled, and I am doing HAS A for the engine component. So, this is a design style, which often is used instead of using the complications of multiple inheritance.

(Refer Slide Time: 34:21)



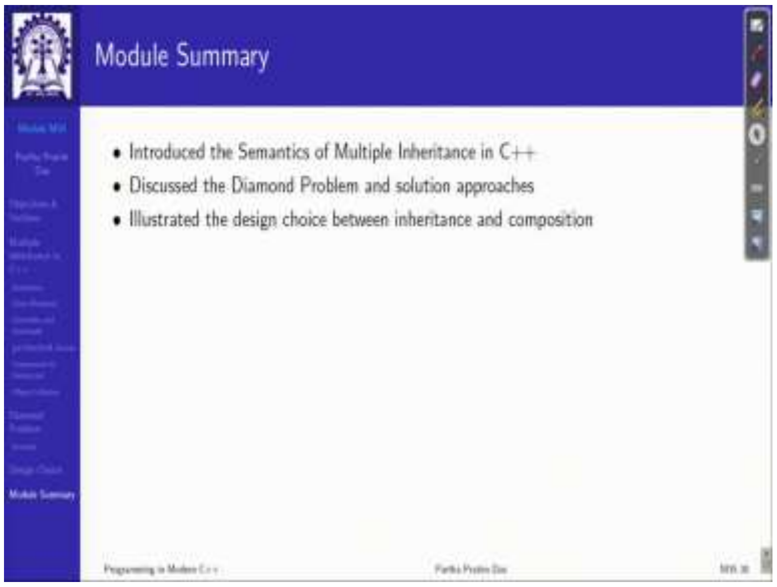
Design Choice: Inheritance or Composition

- Vehicle Hierarchy
- Engine Hierarchy use Wheeled as Component
- Linear options to simplify models
- Is this dominant?

You can look at this in from the other properties perspective also you can take engine to be your dominant property, because that is deciding laws and you know, what all you can do electric fossil fuel manual, and then have all different vehicles and in each you can put the wheels, number of wheels as a composition component and whatever its consequences are wheels, number of wheels it has a lot of other consequences. So, you can do the inheritance on the engine and use the winds as compensation.

So, these are the two ways you can look at decompose the multiple inheritance in terms of composition, it is normally advised that do not try to use multiple inheritance that really complicates matter in even in a simple situation like this, use only one dominant property for simple inheritance and use others for your composition purposes.

(Refer Slide Time: 35:24)



Module Summary

- Introduced the Semantics of Multiple Inheritance in C++
- Discussed the Diamond Problem and solution approaches
- Illustrated the design choice between inheritance and composition

Programming in Modern C++ Partha Pratim Das

So, we have talked about multiple inheritance and the diamond problem and the solution approaches that are all different semantics, and particularly the design choice between inheritance and composition. Thank you very much for your attention, and we will meet in the next week.