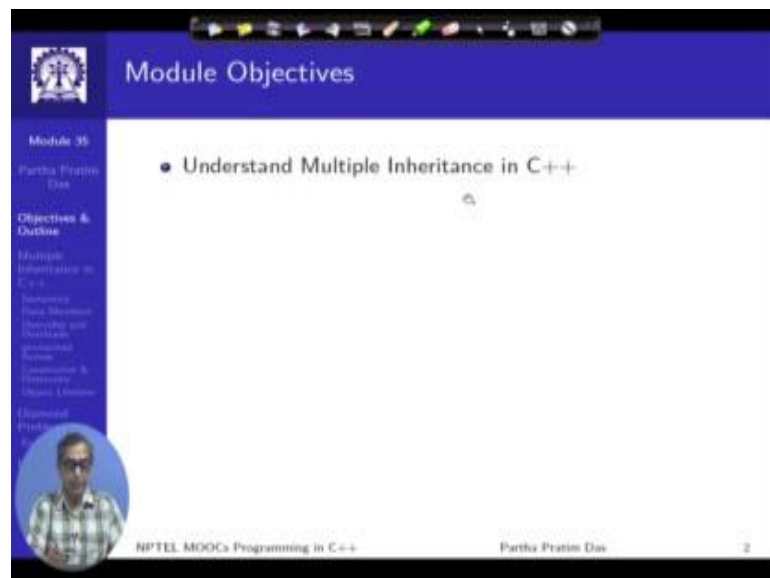


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 50
Multiple Inheritance

Welcome to module 35 of Programming in C++. In this module, we will talk about Multiple Inheritance in C++. We have already discussed the inheritance mechanism in C++ at length; we have also discussed dynamic binding or polymorphism in the context of polymorphic hierarchies.

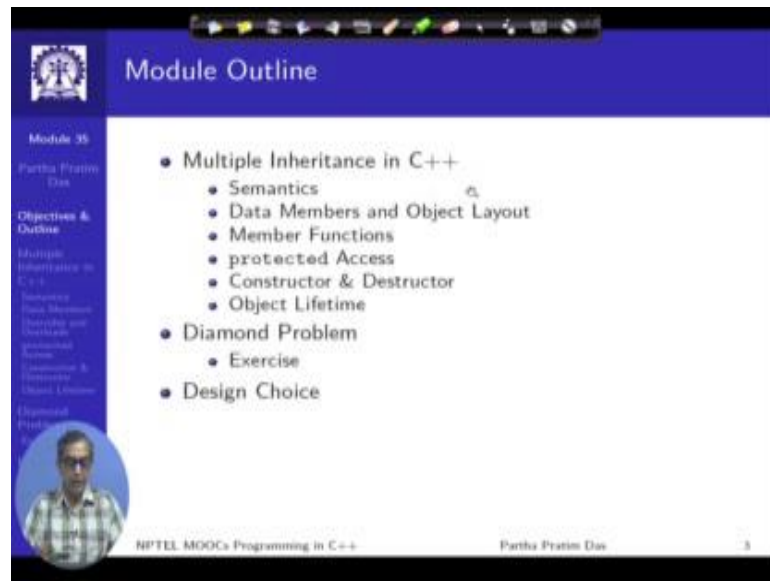
(Refer Slide Time: 00:53)



The image shows a presentation slide with a blue header and a white main content area. The header contains the text 'Module Objectives' and a small logo on the left. The main content area has a single bullet point: '• Understand Multiple Inheritance in C++'. On the left side of the slide, there is a vertical sidebar with a blue background containing the text 'Module 35', 'Partha Pratim Das', and 'Objective & Outline'. Below this sidebar is a circular portrait of a man. At the bottom of the slide, there is a footer with the text 'NPTEL MOOCs Programming in C++' and 'Partha Pratim Das'.

So, we will here try to specifically take a look into the multiple inheritance aspect of C++.

(Refer Slide Time: 01:01)



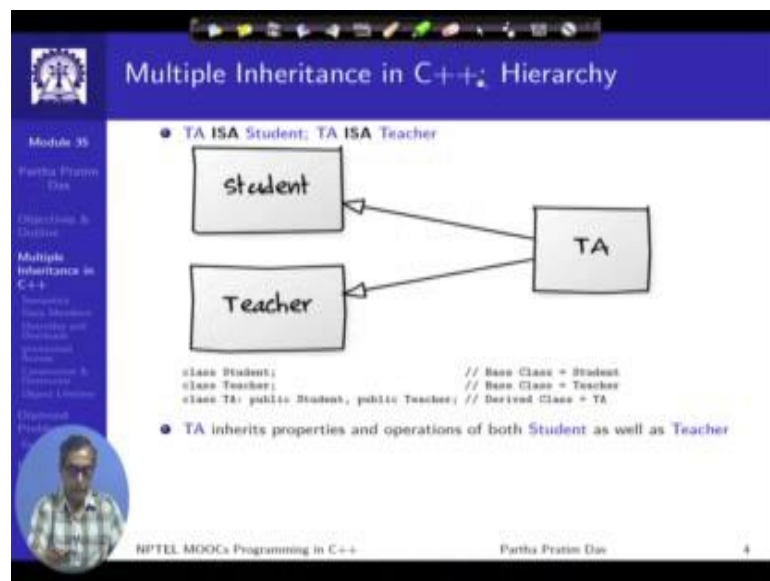
The slide is titled "Module Outline" and is part of an NPTEL MOOC on "Programming in C++". It features a navigation bar at the top with various icons. On the left side, there is a vertical menu with the following items: "Module 35", "Partha Pratim Das", "Objectives & Outline", "Multiple Inheritance in C++", "Semantics", "Data Members and Object Layout", "Member Functions", "protected Access", "Constructor & Destructor", "Object Lifetime", "Diamond Problem", and "Design Choice". The main content area contains a bulleted list of topics to be covered in this module:

- Multiple Inheritance in C++
 - Semantics
 - Data Members and Object Layout
 - Member Functions
 - protected Access
 - Constructor & Destructor
 - Object Lifetime
- Diamond Problem
 - Exercise
- Design Choice

At the bottom of the slide, there is a small circular portrait of the instructor, Partha Pratim Das, and the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". The slide number "3" is visible in the bottom right corner.

This is somewhat advanced topic. So we would cover the multiple inheritance at a top level, and we will live some of the final issues as exercised to you all. This is the Module Outline and it will be available on the left of every slide that you see.

(Refer Slide Time: 01:29)



The slide is titled "Multiple Inheritance in C++: Hierarchy" and is part of the same NPTEL MOOC. It features the same navigation bar and left-side menu as the previous slide. The main content area contains a diagram illustrating multiple inheritance:

```
graph RL; TA[TA] --> Student[Student]; TA --> Teacher[Teacher];
```

The diagram shows a box labeled "TA" on the right, with two arrows pointing to boxes labeled "Student" and "Teacher" on the left. Below the diagram, there is a code snippet:

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

Below the code, there is a bullet point:

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

At the bottom of the slide, there is a small circular portrait of the instructor, Partha Pratim Das, and the text "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". The slide number "4" is visible in the bottom right corner.

Multiple inheritance is a specific form of inheritance where a particular class has two or more base classes. We have seen the representation of inheritance, so TA is a student we know how to represent that, if we see this I am sorry, if we look at this part then we know it represents TA is a student. Similarly TA is a teacher, so that represents it here. This is a scenario where a TA or Teaching Assistant is a student, she attends courses and tries to complete some degree, at the same time she helps some course in terms of tutorials assistance and assignment evaluation and so on. So she also performs a number of operations that are usual for the teachers. So given this we will say that the student and teacher both are base classes of the TA. And when that happens we will say that we have a situation of multiple inheritance.

(Refer Slide Time: 02:47)

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**

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

So, here we show another example of multiple inheritance. There is an employee class which is kind of the root class where all employees of an organization belong, so we can say that manager is a employee. This is represents manager is a employee. We also can say that director is a employee. So, manager manages, director directs the policies of the company. Then we have a managing director, who is a manager as well as a director. So, there is a situation of multiple inheritance that will happen here. When multiple inheritance happen then two things would happen one is every individual inheritance involved in that multiple inheritance will inherit everything in using the rules that we

have learnt for inheritance, but in addition there could be some complications which we will have to discuss.

Then usually if we have a multiple inheritance then it is common that we have some base class which is common between the different classes which worked as a base for the multiple inheritance. So, here we see some kind of a diamond structure being created where we have the final leaf level class of the derived class here, we have intermediate base classes here, which the derived class of the leaf class actually inherits. In turn, these intermediate classes specialize from some combine concept. Because certainly if we are inheriting in a multiple way then this managing director certainly is multiply inheriting from managing and director because it has some commonality with both. So it is expected that manager and director themselves will have certain common properties and certain common operations that are here represented in terms of employee.

In terms of definition of multiple inheritance or multiple inheritance in C++ it is not mandatory to have a common base class, but it is typical that we will often have a common base class representing the whole situation.

(Refer Slide Time: 05:17)

The slide is titled "Multiple Inheritance in C++: Semantics". It features a navigation sidebar on the left with a logo at the top and a list of topics including "Module 35", "Partha Pratim Das", "Objectives & Goals", "Multiple Inheritance in C++", "Semantics", "Accessed", "Encapsulation & Access", "Default Private", and a small circular portrait of the presenter. The main content area has a blue header with the title and a bullet point: "Derived ISA Base1, Derived ISA Base2". Below this is a class diagram showing a box labeled "derived" with two arrows pointing to boxes labeled "Base1" and "Base2". To the right of the diagram is C++ code:

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

 Below the code is a list of four bullet points: "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", and "public and private inheritance may be mixed". At the bottom, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das" with a small number "5" in the bottom right corner.

So, we will go into the actual syntactic details. We say generically derived is a base 1, derived is a base 2, this is a generic scenario. And when we have that this is a class base 1, this is a class base 2, the two base classes. And we write the derived class or the multiply inherited specialized classes; public base 1, public base 2. Earlier, when we had single inheritance if there is a single inheritance you just stop here, here we continue we use a comma and continue to write the next base class which is also inherited by derived. It is not that though I am showing examples where there are 2 base classes, but it is not limited to 2 base classes I can have two or more any number of base classes in a multiple inheritance scenario.

And also as we know that the basic inheritance mechanism in C++ that is the inheritance mechanism what I mean is, is a relationship is represented in terms of public inheritance and we have heard long discussions regarding what does that mean. But at the same time we know that there are other forms of inheritance in C++ particularly something which is known as a private inheritance which changes the visibility of the base class members in the derived class by restricting them to the private access alone. We saw that this is gives us something like a semantics of is implemented as kind of a semantics where we try to represent that if we are inheriting in private from a base class then all that we are saying that this base class actually implements the derived class.

So while we do multiple inheritance it is not necessary that these will have to be public, these could be mixed if you in fact all of them could be private in which case it will mean that both base classes are implementing certain parts of the derived class it could also be that this is public and this is private so if that be that the case then it will mean that a derived is basically specializing from base one in the sense of ISA relationship. Whereas, the class base two implements or helps in implementing the class derived, this is a basic mechanism that we have.

(Refer Slide Time: 07:55)

The slide is titled "Multiple Inheritance in C++: Semantics" and is part of an NPTEL MOOC on Programming in C++. It lists the following semantics:

- Derived **ISA** Base1, Base2
- 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 Base class by **redefining** it with the **same signature**
 - Derived class may **overload** a member function of Base class by **redefining** it with the **same name**, but **different signature**
- Access Specification
 - Derived class **cannot access private** members of Base class
 - Derived class can **access protected** members of Base class
- Construction-Destruction
 - A **constructor** of the Derived class **must first** call all **constructors** of the Base classes to construct the Base class instances of the Derived class – Base class **constructors** are called in **ascending 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

NPTEL MOOCs Programming in C++ Partha Pratim Das

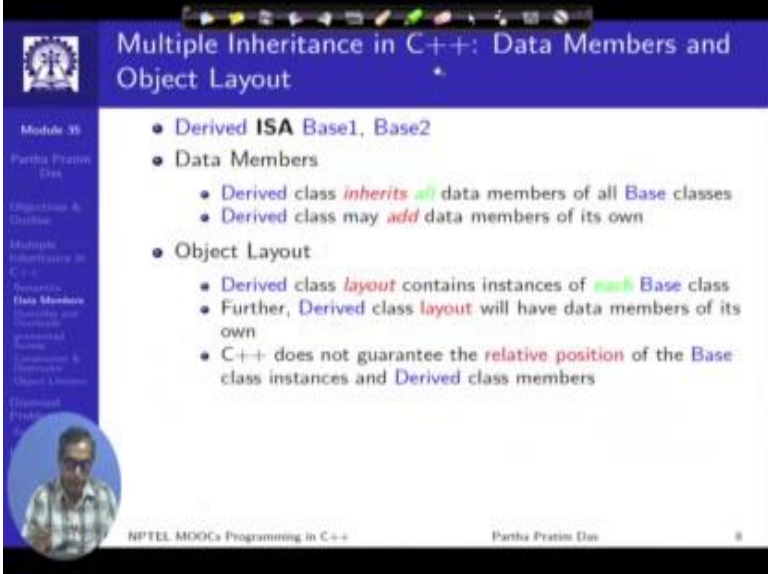
So, we will try look at the semantics. I would suggest that you compare this slide with the semantics of inheritance that we had done earlier. All those semantics are maintained so that as you move from single inheritance to multiple inheritance the basic properties remain same. So, the derived class now inherits all data members of all base classes. It is not just one base class, it inherits all data members of all base classes and it may add new data members. It inherits all member functions of all base classes again it can override or any member function of any base class it can overload any member function of any base class and so on. All these were earlier, this context of multiple base classes were not there so the semantics did not have that, but now since there are multiple base classes all of these will be possible. So, inheritance will mean that all properties and operations of each one of the base class will be inherited and they can be suitably overloaded or overridden.

Access specification will similarly have same semantics of being private, private being completely restricted to the base class. So, the derived class will not be able to access the private data members of any of the base classes. If I use protected for some data members of a base class then those data members would be available in the derived class. In terms of construction and destruction, we will see in terms of constructor. All base class objects will have to be constructed, because all base class objects will become

part of the derived class object. Now, we have two or more base classes that the derived classes is deriving from so we also need to understand the order in which the constructors of these base classes will get executed.

So that will be in terms of listing order as we will see and when it will come to the order of destruction then the same principle that we had seen earlier that the first that is the derived class is destructed, then the base class is destructed, since there are multiple base class objects. So they will be destructed in the reversed order in which they were originally constructed. This is a summary of the semantics for multiple inheritance there is some more details into that which will come through as we go through the examples.

(Refer Slide Time: 10:30)



The slide is titled "Multiple Inheritance in C++: Data Members and Object Layout". It contains the following bullet points:

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

The slide also features a navigation bar at the top, a sidebar on the left with a table of contents, and a small circular portrait of the speaker at the bottom left. The footer includes "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

We start with the specific semantics of data members. So it inherits all data members of all base classes may add new data members. Now in terms of the layout therefore, we have discussed about the layout that, in the layout if a derive class inherits from a base class then it contains an instance of the base class objects. Now, since there are multiple base classes so it will have one instance each for each of the base classes. Again, like in the case of some single inheritance the C++ does not guarantee the relative position of the base class instances. How they will be organized whether first the base class objects

will be there then the derived class object members will be there and so on that specific order is not decided not given by the standard.

(Refer Slide Time: 11:28)

The slide displays the following C++ code:

```
class Base1 { protected: int i, j; public: // ... }; class Base2 { protected: int j, k; public: // ... }; class Derived : public Base1, public Base2 { int k; public: // ... };
```

The 'Object Layout' diagram shows three objects: 'Object Base1' with members 'i' and 'data'; 'Object Base2' with members 'j' and 'data'; and 'Object Derived' with members 'i', 'data', 'j', 'data', and 'k'. A red note states: 'Object Derived has two data members! Ambiguity to be resolved with base class name: Base1: data, & Base2: data.'

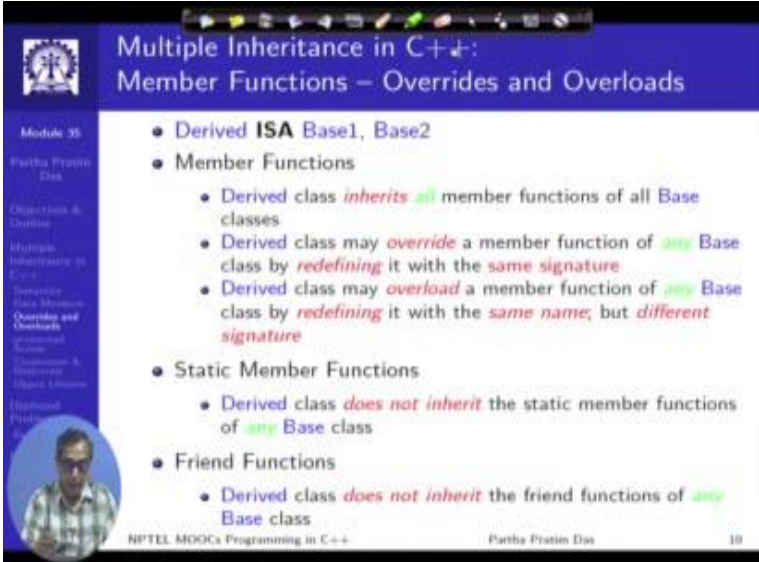
If we look into the instance you have a base class b 1 with two members i and data. I have another base class b 2 with two members; j and data. And I have a derived class which derives from base 1 as well as base 2 and it adds a member k. If I look at one object of base 1 type it will have a member i and a member data, if I look at the object of derived base class b base 2 then it have an instance like this it will have a member j and data. So when we construct a derived class object will have an instance of the base 1 object which one class will have instance of the base 2 class and will have whatever data members we have added in the derived. So, you can clearly see that it is just a direct extension of the semantics of data members that we had or the semantics of layout that we had in case of the single inheritance.

Of course, as I said there could be some pitfalls. For example, as you can see here that base 1 has declared a member data and base 2 also has a member by the exactly the same name. Since base classes are independent of the derived class you cannot control that they would not have members with the same name. When they have members with same name so the object derived this object has two data members by the same name. If I say

that my object is derived d then I want to write derived data which I am authorized to write because data is protected here as well as it is protected here so derived class has access to them.

But if I try to write this certainly the compiler will say that I am confused, because there are two members by the same name. There is a case of ambiguity so if two base classes two more base classes have data members by the same name then the responsibility would lie with the programmer or the designer of the derived class to resolve that ambiguity. You will have to refer to the members explicitly with the class name qualifier. This will not be acceptable by the compiler, but I can write d dot base 1 colon colon data, if I write this it will mean this data member or it will mean this data member in the object if this is d. But if I write base 2 colon colon data, d dot base 2 colon colon data then it will mean that this data member so that resolution additionally would required to be done. This is one added complexity that will have in terms of the multiple inheritance.

(Refer Slide Time: 14:39)



The slide is titled "Multiple Inheritance in C++: Member Functions – Overrides and Overloads". It contains a list of bullet points:

- 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

The slide also features a navigation bar at the top, a sidebar on the left with a table of contents, and a small circular portrait of the presenter at the bottom left. The footer includes "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the slide number "10".

Now, let us move on to the Member Functions - Overrides and Overloads. As I already said the all member functions are inherited from all base classes and you can override any member function, overload a member function from a base class. And like in single

inheritance the static member functions and the friend functions are not inherited by the base class here as well.

(Refer Slide Time: 15:09)

The slide displays C++ code for multiple inheritance. It defines three classes: Base1, Base2, and Derived. Base1 has member functions f(int) and g(). Base2 has member functions h(int) and g(). Derived inherits from both Base1 and Base2. In Derived, f(int) is overridden, h(int) is overloaded, and a new member function e(char) is added. A test program demonstrates the behavior of these functions.

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

class Base2 { protected:
int j_;
int data_;
public: Base2(int a, int b) : j_(a), data_(b);
void h(int) { cout << "Base2::h(int) "; }
};

class Derived : public Base1, public Base2 {
int k_;
public: Derived(int a, int y, int u, int v, int n);
void f(int) { cout << "Derived::f(int) "; } // -- Overridden Base1::f(int)
// -- Inherited Base1::g()
void h(string) { cout << "Derived::h(string) "; } // -- Overloaded Base2::h(int)
void e(char) { cout << "Derived::e(char) "; } // -- Added Derived::e(char)
};

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

d.f(0); // Derived::f(int) -- Overridden Base1::f(int)
d.g(); // Base1::g() -- Inherited Base1::g()
d.h("ppd"); // Derived::h(string) -- Overloaded Base2::h(int)
d.e('a'); // Derived::e(char) -- Added Derived::e(char)
```

If we look at an example, just look at this carefully the same set of base classes; base class base 1, class base 2, and class derived which specializes from base 1 as well as base 2. Here, I have two member functions f and g, and here in base 2 I have a member function h. And what I have done is this is f int in the derived class I have included a member function with the same signature which means this is a case of overriding. So, the derived class is overriding the f member function from base 1. Base 1 also has a member function g and derived class has no mention of any member function by the name g, therefore it inherits g simply and would be able to use that. And when it uses g it will mean the g member function of the base 1 base class. Base class 2 has a member function h and the derived class introduces a member function h with the different signature. We know what will be the effect, this h new h derived colon colon h will hide base 2 colon colon h which was taking integer and now you will be able to call h for a derived class object with a string parameter. So this is a case of overloading. This is simple that derived class can have a new member function e added to the kt and that can be used.

In this context, if we have a derived class object c if I do c dot f 1 then f function in the derived class will be called because base 1 colon colon f have been overridden. If we call c dot g it is a member function of base 1 will be called because it has been inherited, if we called c dot h with, this is where I have used a constant char star kind of parameter which will get cast automatically implicitly to string. So overloaded h function in derived will be called not the base class function because that has got hidden, and if I call c dot e with character a then it will call the e member function that has been introduced in the derived class.

This is the basic story of overriding and overloading that will happen, and certainly like the data member it must be occurring in your mind that what happens if 2 base classes have one function which has the same name.

(Refer Slide Time: 18:06)

The slide is titled "Inheritance in C++: Member Functions – using for Name Resolution". It is divided into two columns: "Ambiguous Calls" and "Unambiguous Calls".

Ambiguous Calls:

```

class Base1 { public:
    Base1(int a, int b) : i_(a), data_(b) {}
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};

class Base2 { public:
    Base2(int a, int b) : j_(a), data_(b) {}
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};

class Derived : public Base1, public Base2 {
public:
    Derived(int a, int y, int s,
           int x, int d);
    using Base1::f;
    using Base2::g;
};

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

d.f(6); // Base1::f(int) or Base2::f(int)?
d.g(6); // Base1::g(int) or Base2::g(int)?
d.f(3); // Base1::f(int) or Base2::f(int)?
d.g(); // Base1::g(int) or Base2::g(int)?
    
```

Unambiguous Calls:

```

class Base1 { public:
    Base1(int a, int b) : i_(a), data_(b) {}
    void f(int) { cout << "Base1::f(int) "; }
    void g() { cout << "Base1::g() "; }
};

class Base2 { public:
    Base2(int a, int b) : j_(a), data_(b) {}
    void f(int) { cout << "Base2::f(int) "; }
    void g(int) { cout << "Base2::g(int) "; }
};

class Derived : public Base1, public Base2 {
public:
    Derived(int a, int y, int s,
           int v, int d);
    using Base1::f; // Hide Base2::f
    using Base2::g; // Hide Base1::g
};

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

d.f(6); // Base1::f(int)
d.g(6); // Base2::g(int)
d.Base2::f(3); // Base2::f(int)
d.Base1::g(); // Base1::g()
    
```

At the bottom, there are two bullet points:

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

The slide footer includes "NPTEL MOOCs Programming in C++", "Partha Pratin Das", and the number "12".

What happens if 2 base classes have a common name for a member function. So I illustrate that in this slide, again I have base 1, base 2, the derived class derives from the same base classes. The difference that is being made is I have f here and I have f here both in base 1 and base 2 and their signatures are same. I have g here in base 1, I have g here in base 2 their signatures are different. Then in further moment ignore this part. Then in the derived class let us say you do not have member function at all just ignore

this part, this you should not consider right now. Now, I try to write `c dot f` the question is what is `c dot f`? Is it `base 1 dot f`, `base 1 colon colon f` or it is `base 2 colon colon f` you have no way of knowing because it has got two versions of `f`.

What is `c dot g`? I have passed a parameter `5` expecting that the compiler would be able to resolve that it is `base 1 colon colon g` because, I am sorry there is a small type of here this should not be `int`, this should be `int`. If I call `c dot g 5` then it I would expect that `base 2 colon colon g` will be called, but unfortunately the compiler would not be able to do that. The reason it will be not able to do that it is a fact that overload is resolved only between same name spaces. If you have two different name spaces then the compiler has no track of what the names are going to be. So, `g` function in `base 1` and `g` function in `base 2` are in into two different name spaces of two different classes so the overload resolution does not kick in here.

So consequently, `c dot g 5` or `c dot g` without any parameter both of them will actually turn out to be ambiguous also. In gist all of these four calls will turn out to be ambiguous and the compiler would not be able to resolve between them. So, if you want to make them resolve that ambiguity you have a simple way of saying that the basic issue that is happening is here you have as you inherit from `base 1` and `base 2` you get two copies of function `f` who wants to come into the derived class. Now, we have learnt about the using declaration for the parent class function, so you can use that. Suppose you say using `base 1 colon colon f`, if you say I am using `base 1 colon colon f` then what it will happen is, this function will be included in derived and this function will not be included, this function `base two's` function will be hidden.

Similar I can do for the `g` function as well, say I am using `base 2 colon colon g` which means this `g` function will be included, but `base ones` `g` function will be hidden. In this context now if I would like to call `c dot f` then it will call `base 1 colon colon f` because I have been using this. If I do `c dot g 5` it will call `base 2 colon colon g` because I am using the `g` function of `base 2`. Now in this if I also want, there is a situation that in some case I want to actually also access the `f` function of `base 2` then like I did in case of data members I can explicitly write `c dot base 2 colon colon f 3` in which case it will actually call `f` member function of the `base two` class.

Even though in the derived class I have said I am using base 1 colon colon f. What actually using does, using basically allows me to do a short cut that I do not need to qualify the name of the member function with the base class name and I can use it as a default as I am doing here, but I still always have the option of actually providing the qualified name for the member function, like I did for the data member and use the other members in that form.

So, this is what is additionally required over single inheritance which will be very common because it is quite likely that the base classes that you are inheriting from may have one or more member functions which are have the same name between themselves. As we have seen there it does not matter in terms of what actually are their signature what matters is they have the same name, and if they have the same name then the derived class cannot use them without any using qualification.

(Refer Slide Time: 23:22)



The slide is titled "Multiple Inheritance in C++: Access Members of Base: protected Access". It contains a section titled "Access Specification" with two bullet points:

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

The slide also features a navigation bar on the left, a small circular portrait of the speaker, and footer text: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and "13".

Coming to the access members of a base, the protected access will allow any derived class object to access the protected members of the base class of any of the base class there is nothing to add in terms of multiple inheritance here. So, all that we have learnt in case of single inheritance will simply apply so we will skip further discussion on this aspect.

(Refer Slide Time: 23:46)

The slide is titled "Multiple Inheritance in C++: Constructor & Destructor". It contains the following content:

- Constructor-Destructor
 - Derived class *inherits all* Constructors and Destructor of Base classes (*but in a different semantics*)
 - Derived class *cannot override or overload* a Constructor or 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

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Prabin Das". There is also a small circular portrait of a man in the bottom left corner.

Let me move onto the Constructor, Destructor. The constructor, destructors of the derived class will inherit all the constructor destructors of the base classes, but in a different semantics as we saw in case of the single inheritance because it cannot directly inherit that because it has a different name, it adds name of the base class and certainly you cannot override or overload the constructor, destructor in any way. So, if we see with that then we can see that there are base class has a constructive here.

(Refer Slide Time: 24:22)

```
class Base1 { protected: int k_; int data_;
public: Base1(int a, int b) : k_(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 a, int y, int z) :
Base1(a, y), k_(z) { cout << "Derived:Derived() "; }
// Base1:Base1 explicit, Base2:Base2 default
Derived() { cout << "Derived:Derived() "; }
};
int main() {
Base1 b1(5, 3);
Base2 b2(0, 7);
Derived d(5, 3, 2);
}
```

Object Layout

Object b1	Object b2	Object d
5 3	0 7	5 3 0 0 z

NPTEL MOOCs Programming in C++
Partha Pratim Das 15

The derived class has a constructor and, sorry the second base class has another constructor and in the derived class has to; here is invoking the constructor of the base one. So what will happen? Now, it has to construct both the base class objects; the fact that is invoking base 1 means that the base 1 constructor will be invoked through this and since it is skipped base 2, the base 2 must have a default constructor which will be invoked after that. So, if I have just invoked the base 1 constructor and base 2 does not have a default constructor then I will have a compilation error, because to be able to construct a derived class object I need construct both base 1 and base 2 kind of objects.

So, if we see the instance this is an object of base one type; this is an object of base two type this is what we can constructed. Here you can see the object of a derived type being constructed where the base class 1 has the instance 5 3 which got created through this. The instance of base class 2 is by default so it has 0 0 as members and this is a data member of the derived class. This is the basic dynamics of the construction process.

(Refer Slide Time: 25:48)

Multiple Inheritance in C++ Object Lifetime

```
class Base1 { protected: int i, j; int data;
public: Base1(int a, int b) : i(a), data, 0 { cout << "Base1::Base1() "; }
      ~Base1() { cout << "Base1::~Base1() "; }
};
class Base2 { protected: int j, i; 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 a) : Base1(x, y), k,(a) { cout << "Derived::Derived() "; }
      // Base1::Base1 explicit, Base2::Base2 default
      ~Derived() { cout << "Derived::~Derived() "; }
};
Derived d(0, 3, 2);
```

Construction O/P	Destruction O/P
Base1::Base1(): 0, 3 // Obj. d.Base1	Derived::~Derived(): 2 // Obj. d
Base2::Base2(): 0, 0 // Obj. d.Base2	Base2::~Base2(): 0, 0 // Obj. d.Base2
Derived::Derived(): 2 // Obj. d	Base1::~Base1(): 3, 0 // Obj. d.Base1

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 16

If you put messages into the base class constructor and destructors and so on and the derived class constructor destructor, then you will be able to see that the first the base class 1 is constructed because it is first in the list, then the base class 2 because it is second in the list, and then the derived class constructed and the destruction happens exactly in the reverse order.

(Refer Slide Time: 26:20)

Multiple Inheritance in C++: Diamond Problem

- Student ISA Person
- Teacher ISA Person
- TA ISA Student; TA ISA Teacher

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

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 17

This is the basic mechanism of inheritance that goes on in terms of the multiple cases of base class being there for any particular derived class problem.