

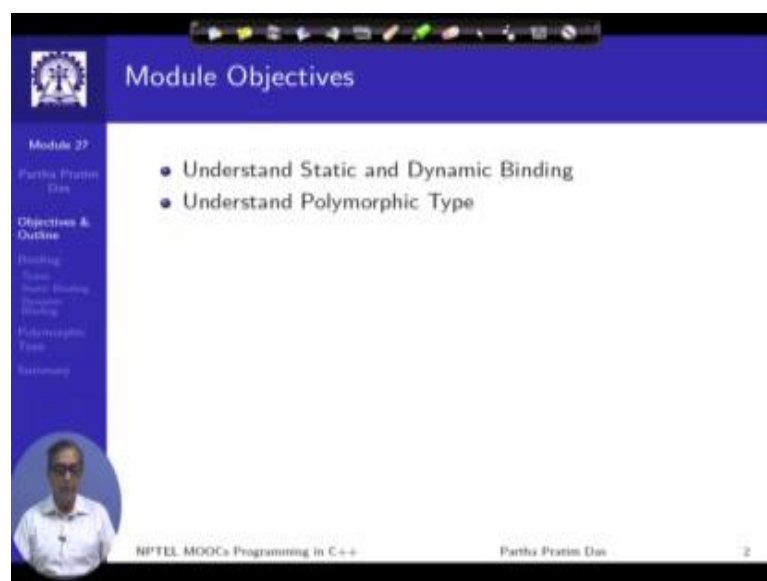
**Programming in C++**  
**Prof. Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**

**Lecture - 42**  
**Dynamic Binding (Polymorphism): Part 2**

Welcome to Module 27 of Programming in C++. We have been discussing about static and dynamic binding. In fact, we talked at length in the last module about various casting options, particularly when casting is done on a class hierarchy; and we observed that if we up cast from a specialized class to a generalized class then that are valid operation.

Because by interpreting specialized class object as a generalized class object we are only using part of the information that is available. But downcast when you try to cast a generalized class object as a specialized class object, we are trying to interpret information that does not exist for the specialized class, so that will be an error. And we had discussed about this casting issue, because particularly when we discuss dynamic binding over class hierarchies then regularly we need to perform a different kinds of up cast.

(Refer Slide Time: 01:37)



The image shows a presentation slide with a blue header and a white main content area. The header contains the IIT Kharagpur logo and the text 'Module Objectives'. The main content area lists two objectives: 'Understand Static and Dynamic Binding' and 'Understand Polymorphic Type'. A sidebar on the left lists 'Module 27', 'Partha Pratim Das', 'Objectives & Outline', 'Binding', 'Static Binding', 'Dynamic Binding', 'Polymorphic Type', and 'Summary'. A small circular portrait of the professor is in the bottom left corner. The footer contains 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the page number '2'.

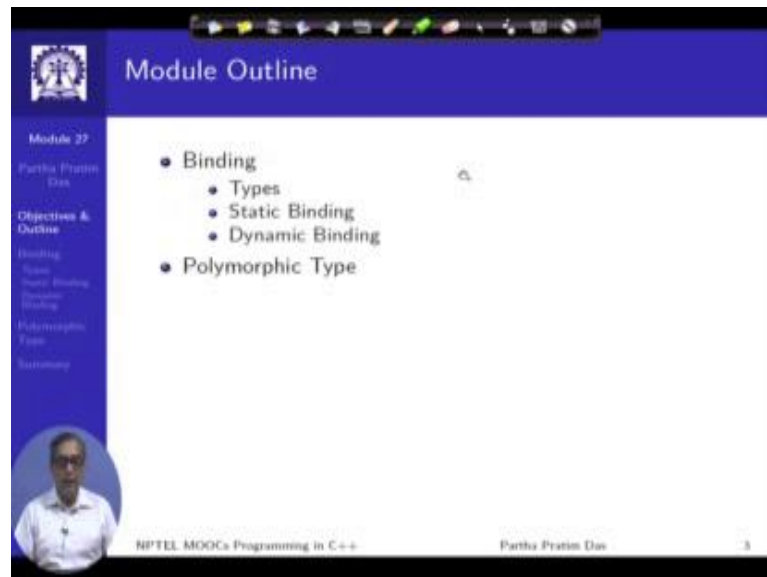
**Module Objectives**

- Understand Static and Dynamic Binding
- Understand Polymorphic Type

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

So, we have taken a look at an example. What we will do in this module we will continue to discuss about or detailed understanding of static and dynamic binding; and with that, we will introduce what is known as a polymorphic type in C++.

(Refer Slide Time: 01:52)



The image shows a presentation slide titled "Module Outline" for "Module 27". The slide is part of an NPTEL MOOCs course on "Programming in C++" by Partha Pratim Das. The main content of the slide is a bulleted list of topics:

- Binding
  - Types
  - Static Binding
  - Dynamic Binding
- Polymorphic Type

The slide also features a navigation bar at the top, a sidebar on the left with a table of contents, and a small circular video feed of the presenter in the bottom left corner. The footer contains the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the page number "3".

This will be the outline of the module, and will be visible on the left hand side of your screen always.

(Refer Slide Time: 02:02)

**Type of an Object**

- The static type of the object is the type declared for the object while writing the code
- Compiler sees static type
- The dynamic type of the object is determined by the type of the object to which it currently refers
- Compiler does not see dynamic type

```
class A {};  
class B : public A {};  
  
int main() {  
    A *p;  
    p = new B; // Static type of p = A  
              // Dynamic type of p = B  
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Let us understand what we mean by the static and dynamic type of objects. We have so far understood that in object if I have a class A and I define an object of instantiate an object of class a, then the type of this the type of a is the class A. We have understood that and that is what is uniform all through. Things start becoming different when we have a pointer variable or a reference variable, which is of a certain type, certain class which is a part of a hierarchy. So here we are showing such an instance, we have a class A, and we have another class B, the hierarchy diagram is like this, which is a specialization of A, B is A.

And in this context, when we have a pointer of type A, so it means that it is this pointer p can store an address where I expect to get an A type of object which is fine. So if I just do say if I say that I have A star p, and I have created a new object A, and kept the address in p so that will be in the memory somewhere this object A has been created dynamically, instance of A has been created dynamically instance of A has been created. B is the pointer, which is pointing to this. In this case, the type of p as we know as a compiler we will get to see is that of a pointer to A. And the object that it is actually pointing to is also of type A, which is the normal scenario.

(Refer Slide Time: 04:18)

**Type of an Object**

- The static type of the object is the type declared for the object while writing the code
- Compiler sees static type
- The dynamic type of the object is determined by the type of the object to which it currently refers
- Compiler does not see dynamic type

```
class A {};  
class B : public A {};  
  
int main() {  
    A *p;  
    p = new B; // Static type of p = A  
               // Dynamic type of p = B  
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

But, let us think that instead of creating A object, I have created a B object. So I have created here, dynamically I have created a B object which means that it has a base part which is of A type, and this is my whole dynamically created object. And I have a pointer p, which is of type A, so the pointer p points to this. This we have seen this is a scenario of up cast that we have seen is quite possible; and therefore, actually if we look at the object through p then we will be able to see only the base class part of it we have understood why this is so. Now, in this context, we can see that the actual object that is created has a type which is of B; this has a type which is of B. And the pointer has a type which is of A.

So we will distinguish these two now. We will say that the type of p statically the type of p is A, why are we say statically, because this is what the compiler has got to see. The compiler has got to see that p has been defined as of A type pointer. But what has happened in the run time, what has happened in the actual execution; in the actual execution act, really a B type object has been created, and it is pointer put to p. So, the dynamic type of what p points to is actually, it is pointing to a B type object though it is actually A type pointer. So this is the notion of the dynamic type.

As I mentioned that if I had just done A star p and we had done a instantiated A object there then the static and dynamic type both are A, and they will be same, but it is possible that the static and dynamic type are different. So, it is very important to keep track of what is the static type that is what the compiler c and what is a dynamic type which happens at the run time with the compiler cannot see. Now, certainly you would be curious to know as to why we are creating these two static and dynamic type concepts, and how they will be used, so that will unfold after maybe another module also when we actually show applications of how the dynamic type is really very critical to do object oriented programming.

(Refer Slide Time: 07:01)

**Static and Dynamic Binding**

- **Static binding (early binding):** When a function invocation binds to the function definition based on the static type of objects
- This is done at compile-time
- Normal function calls, overloaded function calls, and overloaded operators are examples of static binding
- **Dynamic binding (late binding):** When a function invocation binds to the function definition based on the dynamic type of objects
- This is done at run-time
- Function pointers, Virtual functions are examples of late binding

*Handwritten notes in red ink:*  
 - `void f(int)`  
`void f()`  
`void f(int)`  
`void f(int)`  
`void f(int)`  
 - `f(3)`  
`f()`  
 - `f(3)`  
`f()`

So, with this type of object, we can define static and dynamic binding. So, formally, I just shown an example, but now I am giving you the formal definition. We say that when a function invocation binds to the function definition that is done based on the static type we say that a static binding has happened. So, what we are saying we are saying I have a function say void f and I am calling f, so while I am calling f, I am binding to this f.

Let say different scenario, I have void f int, I have void f that is I have overloads of this and then I am calling f 3 or I am calling f, if I call f 3, I bind to this; if I call f, I bind to this. So, binding is the process by which from an invocation, I can say what is the actual

function that will be invoked from this invocation that is the process of binding that is what is known as binding.

And if I can resolve that at the compile time if I can decide on that at the compile time then I say that I have static binding, and alternately this is also called early binding, because early in the sense that the compilation certainly has to precede the execution of a program. So here at the compilation time at the program writing time itself I know what the binding will be, and therefore, this is also called early binding. So normal function calls as I was showing; overloaded function calls, overloaded operators and so on. The various different kinds of function invocations that we have seen are typically examples of static binding.

In contrast, the dynamic binding talks about when a function invocation binds to the function definition based on the dynamic type of the object that is if we have class B as a specialization of A, and let say both of them have a function f, and I am invoking a function based on a pointer. I need to decide whether it should bind here and whether it or it should bind here. If this decision depends on the dynamic type of p that is not the type of p itself, p could simply be a type of A - the base type, a pointer to the base type.

But if the binding depends not only on the type of p, but on the actual object that the p is pointing to. If it is pointing to A type object, then it should bind to the f member of class A, but if p is pointing to a B type object then it should bind to the f member of the b object, this is what is known as dynamic binding. Certainly whether p points to an A type object or it points to a B type object, p pointer f this expression does not change.

So, the compiler has got to see only p pointer f. So compiler, at the compilation time at the static time could not have decided as to whether a colon-colon f will be invoked or b colon-colon f will be invoked this will be decided later at the time of execution that is why this is called late binding. So this is dynamic binding, because it depends on the dynamics or the execution run time of the program. It is late binding because it is happening later than the compilation time. This is as it is the run time.

So, though it was not very formally discussed and organized, C also has this kind of a concept in terms of function pointers. So if I say that I have a typedef and I say void star p f, then p f becomes f type of a function pointer. So I can say p f then myf then myf is a function pointer which can point to any of the functions which takes a void and returns a void. So, in this context, if I write void g, if I write void h, then if I then invoke myf as a function then this expression may invoke g or may it may invoke h, depending on if I have assigned g to my f then this will invoke g. If I have assigned h to myf then this will invoke h. So this is the basic concept of the function pointer that you know.

And so this also is a situation of dynamic binding, because compiler from knowing this expression cannot know whether myf actually has been set to point to the g function or to point to the h function. So, function pointers also are the fundamental which offer the dynamic binding in c itself and certainly in C++. And then we will see that the virtual functions are the right example of late binding, dynamic binding that we have in C++.

(Refer Slide Time: 13:05)

**Static Binding**

Inherited Method	Overridden Method
<pre>#include &lt;iostream&gt; using namespace std; class B { public:     void f() { } }; class D : public B { public:     void g() { } // new function }; int main() {     B b;     D d;      b.f(); // B::f() ----- Inherited     d.f(); // D::g() ----- Added }</pre>	<pre>#include &lt;iostream&gt; using namespace std; class B { public:     void f() { } }; class D : public B { public:     void f() { } }; int main() {     B b;     D d;      b.f(); // B::f() ----- Overridden     d.f(); // D::f() ----- Overridden     // <del>Skips the base class function</del> }</pre>
<ul style="list-style-type: none"> <li>Object d of derived class inherits the base class function f() and has its own function g()</li> <li>Function calls are resolved at compile time based on static type</li> </ul>	<ul style="list-style-type: none"> <li>If a member function of a base class is redefined in a derived class with the same signature then it masks the base class method.</li> <li>The derived class method f() is linked to the object d. As f() is redefined in the derived class, the base class version cannot be called with the object of a derived class.</li> </ul>

NPTEL MOOCs Programming in C++ Partha Pratim Das

So let us go over into different cases. So, first little bit more on the static binding. I have a class B, as a base - class d is the derived specialized class. So, if I have a member f here and a member g here, and then I construct two objects. If I do B dot f certainly it will call this function because it knows statically. If I call d dot f, it will also call this

function, this will also call this function, why because d has inherited, and we know that being a specialization it will inherit, so d has inherited f so it will call B colon-colon f because it has been able to inherit that; though it is not explicitly written in the scope of d. And certainly if I call the new function g, d dot g then it will call the d dot g, which is so this is the basic notion of inherited functions that we had seen.

Now, if I overwrite, I am doing the same thing, but instead of introducing a new function I have introduced the signature of the same function in B. If I have done that then this will kind of, now if I do b dot f certainly, but now if I do d dot f, it will invoke the new d colon-colon f or the overwritten function. So, moment I overwrite I am actually masking the base class function that is for d, the base class function b colon-colon f is no more available so that is a basic structure of the overriding operations. So, we will have to be careful in dealing with that. We cannot use both the base class as well as the derived class functions.

(Refer Slide Time: 14:58)

**Member Functions – Overrides and Overloads: RECAP (Module 22)**

Inheritance	Override & Overload
<pre> class B { // Base Class public:     void f(int i);     void g(int i); }; class D: public B { // Derived Class public:     // Inherits B::f(int)     // Inherits B::g(int) }; B b; D d; b.f(1); // Calls B::f(int) b.g(2); // Calls B::g(int) d.f(3); // Calls B::f(int) d.g(4); // Calls B::g(int) </pre>	<pre> class B { // Base Class public:     void f(int);     void g(int i); }; class D: public B { // Derived Class public:     // Inherits B::f(int)     void f(int); // Overrides B::f(int)     void f(string); // Overloads B::f(int)     // Inherits B::g(int)     void h(int i); // Adds B::h(int) }; B b; D d; b.f(1); // Calls B::f(int) b.g(2); // Calls B::g(int) d.f(3); // Calls D::f(int) d.g(4); // Calls B::g(int) d.f("red"); // Calls D::f(string) d.h(5); // Calls D::h(int) </pre>
<ul style="list-style-type: none"> <li>• B::f(int) overloads B::f(int)</li> <li>• D::f(string) overloads B::f(int)</li> </ul>	

Now in this context, I would point you to an example that we had taken here earlier which had shown the same thing. And you have a base class is the function f, which is overwritten here, and is also overloaded here. So, the derived class has two functions f and f which takes an integer and f which takes a string, so when we make the derived



class object and we invoke f with 3, it invokes the f int function, if we invoke with red then it invokes the overloaded f function. So, you can override and overload at the same time, so this is what we have seen. So, I am just reminding of this because all this will now get mixed.

(Refer Slide Time: 15:53)

```
#include <iostream>
using namespace std;
class A { public:
void f() { }
};
class B : public A {
// To overload, rather than hide the base class function f()
// is introduced into the scope of B with a using declaration
using A::f;
void f(int) { }
};
int main() {
B b; // function calls resolved at compile time
✓ b.f(); // B::f(int)
✓ b.f(); // A::f()
}
```

• Object b of derived class linked to with inherited base class function f() and the overloaded version defined by the derived class f(int), based on the input parameters - function calls resolved at compile time

Now, suppose you have a situation, where you have a function in the base class, member function in the base class which you over one two overload in the derived class. You have a function in the base class which you want to overload in the derived class. So you have written the overload.

Now the question is as you overload, you will also hide the inherited function that you have inherited from the base class. So, if you just do this, and in then if you try to write b dot f, you will get an error, why will you get an error because now the compiler knows that you have just overloaded you have inherited, and overloaded that so the compiler knows that now you have f function in class b which is which will take an int and which will not work with no parameter. So, if you still want that you would like to inherit this, but unlike the earlier example would not like to overwrite this, then you can make use of what is known as a using definition or using construct, so what you say using a colon-colon f.

So what does it tell you, it tells you that you are inheriting this base class member function, and you have overloaded that, but as you inherit you do not want to override that; so with that, if you do `f b dot f 3` then it calls the B class function which is the overloaded 1, but if you just call `b dot f` without any parameter then it calls the inherited function. So we have seen that couple of things if we overwrite we hide the function; if we overload, we also hide the inherited function; if we overwrite the inherited function then we have a new function of the same signature in the derived class. But if we want that we would overload in the derived class, and also use the inherited function from the base class, then we can make use of this using construct.

So with that, all possible designed combinations can be done. And these are all decisions that are made in the static type so this is the different situations of static binding.

(Refer Slide Time: 18:29)

The slide is titled "Dynamic Binding" and is divided into two columns: "Non Virtual Method" and "Virtual Method".

**Non Virtual Method:**

```
#include <iostream>
using namespace std;
class B { public:
    void f() { }
};
class D : public B { public:
    void f() { }
};
int main() {
    B b;
    D d;
    B *p;
    p = &b; p->f(); // B::f()
    p = &d; p->f(); // B::f()
}
```

**Virtual Method:**

```
#include <iostream>
using namespace std;
class B { public:
    virtual void f() { }
};
class D : public B { public:
    virtual void f() { }
};
int main() {
    B b;
    D d;
    B *p;
    p = &b; p->f(); // B::f()
    p = &d; p->f(); // D::f()
}
```

**Key Points:**

- `p->f()` always leads to `B::f()`
- Binding is decided by the type of pointer
- **Static Binding**

**Key Points:**

- `p->f()` leads to `B::f()` for a B object, and to `D::f()` for a D object
- Binding is decided by the type of object
- **Dynamic Binding**

The slide also includes a navigation bar at the top, a sidebar on the left with "Module 27" and "Partha Pratim Das", and a footer with "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

Now, let us talk about dynamic binding. So, I again look into the static case; I have one member function here, which is specialized, which is overridden in the derived class function. So, if I have two objects, one of the base class and one of the derived classes. And if we try to put their addresses in a pointer of type base class, and invoke the function `f` as in here, then in both cases it will actually call the base class function. While it calls the base class functions because I am statically binding, because I know that `p` is a

pointer of type base class, which has this function. So if I do p pointer f, it will call this; if I do p pointer f, when p is actually pointing to a derived class object, it will still call the base class member, so that is the basic scenario.

We can change that by introducing the virtual keyword. If we say that this member function is virtual, and then if we overwrite it in the, again we have the base class thus everything else is in this example except the fact that the function is now said to be a virtual one. In the same scenario, I have the same base class type pointer; I have the same two objects; and the two scenarios, where this points to the base class object in this it points to the derived class object. I am again trying to do p pointer f look at statically these are the same expression, but if we do it with the base class object, it invokes the base class function; if I do it with a derived class object, it invokes a derived class function.

So, here this was both were going to the base class now the second one which actually has a dynamic type of being a d type of object, for that object the same pointer expression pointer invocation expression will take me to the derived class function. So, that is what dynamic binding is. So, you can see that the expression has not changed between these two, it is p pointer f, but depending on whether you are pointing to a derived class object or to a base class object, you are automatically being a bound to either the derived class function or the base class function. So, this is the basic concept of virtual function or dynamic binding virtual methods that we have in C++, which will have several use in.

(Refer Slide Time: 21:18)

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

class B {
public:
    void f() { cout << "B:f()" << endl; }
    virtual void g() { cout << "B:g()" << endl; }
};

class D: public B {
public:
    void f() { cout << "D:f()" << endl; }
    virtual void g() { cout << "D:g()" << endl; }
};

int main() {
    B b;
    D d;

    B *pb = &b;
    B *pd = &d; // UPCAST

    D *db = d; // UPCAST

    b.f(); // B:f()
    b.g(); // B:g()
    d.f(); // D:f()
    d.g(); // D:g()

    pb->f(); // B:f() -- Static Binding
    pb->g(); // B:g() -- Dynamic Binding
    pb->f(); // B:f() -- Static Binding
    pb->g(); // D:g() -- Dynamic Binding

    db.f(); // B:f() -- Static Binding
    db.g(); // B:g() -- Dynamic Binding
    db.f(); // B:f() -- Static Binding
    db.g(); // D:g() -- Dynamic Binding

    return 0;
}
```

So, this is just the example that we had seen in the last module. So, you could just go through this again for your understanding. So, we have one method, which is non-virtual; and we have other method which is virtual. And therefore, if we invoke all of these with the object then the respective member functions are invoked. So when we invoke from B the member functions of B are invoked; when I invoke from D member functions of D are invoked.

But when we use a base type of pointer to keep the address of either B or the address of d, of course, through an UPCAST, and then start doing that the same set of invocations through the pointer. Then for the base class object, I always invoke the base class member functions. But for the derived class object, I invoke the base class member function for a non-virtual method which is static binding which will take me here; but for a derived class object, I actually invoke the derived class member function because g is a virtual function g has a dynamic binding in contrast to f which has a non-dynamic or static binding.

And just to get clarify matters, the similar behavior would be shown if you use the instead of pointer, if you use reference as well. So these are to reference of the B class type which has alias to a B object and there is also a reference to a B class type, but it has

a alias to a D object through UPCAST. So, when I invoke g through this reference for the virtual function g, because actually I am maintaining the reference to a D object my invocation will go to the inherit will go the D class function the virtual function. So, this is the basic mechanism.

(Refer Slide Time: 23:32)

**Polymorphic Type: Virtual Functions**

- Dynamic binding is possible only for pointer and reference data types and for member functions that are declared as virtual in the base class.
- These are called **Virtual Functions**
- If a member function is declared as virtual, it can be overridden in the derived class
- If a member function is not virtual and it is re-defined in the derived class then the latter definition hides the former one
- Any class containing a virtual member function – by definition or by inheritance – is called a **Polymorphic Type**
- A hierarchy may be polymorphic or non-polymorphic
- A non-polymorphic hierarchy has little value

NPTEL MOOCs Programming in C++ Partha Pratim Das 11

So, from based on this, we define the polymorphic type with the virtual function. We can see that the dynamic binding is possible only for pointer and reference data type. So, we have shown that if we directly invoke a member functions from the object then there will be bound statically always. Such functions which are called written as virtual in the front are known as virtual function, and if a virtual member function is declared virtual it can be overridden in the derived class as we have seen. If a member function is not virtual, and it is redefined in the derived class as we have say then the latter definition will hide or suppress mask the former definition.

So any class that contains virtual member function either by definition or by inheritance, you may have defined a virtual member function yourself or you may have inherited it from your parent, but if you have at least one virtual member function, then that class is said to be a polymorphic type. Polymorphic, because it can take different forms at based on the run time object that the pointer of the reference is referring to. And certainly a

hierarchy as a whole could be polymorphic or non-polymorphic depending on; if a hierarchy is non polymorphic then certainly it does not have any polymorphic function or any virtual function in terms of the classes that it involves.

But, if the base class of a hierarchy has a virtual function or some class has a virtual function then the whole hierarchy that hangs from that class will become polymorphic. And as we will see that non-polymorphic hierarchies are really I mean they can be created, but they just have structural value, but they will have a little computational value, because you will not be using the major advantage of having the polymorphic type.

(Refer Slide Time: 25:27)

```
#include <iostream>
using namespace std;
class A { public:
    void f() { cout << "A::f()" << endl; } // Non-Virtual
    virtual void g() { cout << "A::g()" << endl; } // Virtual
    void h() { cout << "A::h()" << endl; } // Non-Virtual
};
class B : public A { public:
    void f() { cout << "B::f()" << endl; } // Non-Virtual
    void g() { cout << "B::g()" << endl; } // Virtual
    virtual void h() { cout << "B::h()" << endl; } // Virtual
};
class C : public B { public:
    void f() { cout << "C::f()" << endl; } // Non-Virtual
    void g() { cout << "C::g()" << endl; } // Virtual
    void h() { cout << "C::h()" << endl; } // Virtual
};

int main() { B *q = new C; A *p = q;

    p->f();
    p->g();
    p->h();

    q->f();
    q->g();
    q->h();

    return 0;
}
```

Now, I would just highlight a little bit of rule on polymorphism. I am taking another example, where A is the base class; B is a specialization of that C in turn is a specialization of B. So, this is a simple multilevel inheritance. I have three functions f, g and h. So, in class A, this is just simply defined, so this is non-virtual. This is defined as virtual and this is another h is also defined as non-virtual. So, what I can say that this now has A has at least one polymorphic function or at least one virtual function, so this whole hierarchy is a polymorphic hierarchy as a first thing we observe.

Then going to class B all these functions are overridden, so B has overridden this which continues to be non-virtual. g is again overridden in B which continues to be virtual this is what needs to be noted that once I make a function virtual in a class, then any class which derives it must get it as a virtual function. And for that it is not mandatory to write the virtual keyword here.

I may write that as I did in the last example, I may not write that. But even if I do not write the virtual keyword in front of this inherited function, which I am overriding, while that inherited function was virtual in the base class that it will continue to be virtual in the derived class as well. So, once g is polymorphic here, g is virtual here, irrespective of whether I simply inherit it or I inherit and override it. And I writing the virtual keyword in front of it may be a good practice, because anybody else can quickly understand that, but it is not mandatory.

Now, what I do I do something more interesting, I have also inherited h, which was non-virtual and coming to B, I make it virtual that is now I have written it virtual. So, what will happen in C, if as c overrides them. This continues to be non-virtual, this was non-virtual here and both of these which were virtual in B will now become also virtual in C. So in view of this, if I if I try to see if I have in key in q, I create a C type of object, and I am using two pointers, one is a q pointer two point, and one is a p pointer two point.

The difference being p is of type 'A' type pointer, and q is a 'B' type pointer. So, what happens if I do p pointer f then certainly it is pointing to a C type object, p has the static type of p is a, f is a non-virtual function in class A, so A colon-colon f gets called that is a simple static binding case.

But if I call p pointer g then g is a virtual function in A, so the dynamic type of p, which is type C will be used, and therefore, this function which is the virtual function in C the overridden virtual function is C which will get called. And if I invoke h, then naturally again it is like f, it is non-virtual in A, so if I since I am calling from p, it is decided by the type of p and a colon-colon h the function in a will be invoked.

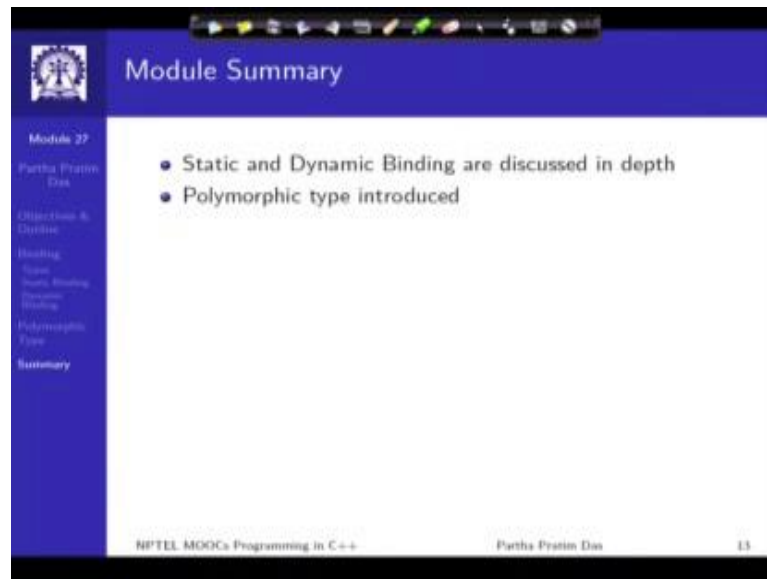
Now, let us consider q, let us consider the other pointer, I look q pointer f. What is f, f is non-virtual so this will be statically bound. So what it is non-virtual in B, because q is a B type pointer. So, now, I have to see what is the definition of the functions in B, because q is a B type of pointer. Now in q in B, f is a non-virtual, so if I do q pointer f, it will call the f functions in B, because it is statically resolved. g is a virtual function in B, so it will do a polymorphic dispatch, it will be decided based on the type of object that q is pointing to which is s c type object, therefore, this will invoke the g function in the C class because it is decided by the type of the object.

The interesting thing happens with when I do q pointer h. When I do q pointer h, q is of type B, so I will look up the h function in type B. In type B, h function is a virtual function; now since it is a virtual function, its invocation its binding will be decided by the dynamic type that is the type of C. So, this q pointer h now calls the h function in C. So, you can see that earlier, when I invoked for the same object, when I did the invocation from p, it invoked this function A colon-colon h. When I invoked it from using the pointer to B, it invokes the C colon-colon h. So, here this was static and here this has become dynamic. And the change has happened because in the derived class B, I have changed while overriding, I have changed the binding property of the function h.

So please study this example in further detail and try to understand the reason that this is a basic polymorphic rule that once a virtual it will continue to be virtual in all specializations. But any non-virtual function, at any stage could be made virtual, and from that point onwards, downwards in the hierarchy it will continue to be a virtual function. And the compiler will always take the static type of the pointer and go to that class see what is the function whether the function is virtual or non virtual. If it is non-virtual, it will use static binding; if it is virtual, it will create code for dynamic binding, so that the binding will be decided based on the actual type being used at the run time.



(Refer Slide Time: 32:30)



The image shows a presentation slide titled "Module Summary" with a blue header and a white main content area. The slide is part of an NPTEL MOOCs course on "Programming in C++". The slide content includes a list of two bullet points: "Static and Dynamic Binding are discussed in depth" and "Polymorphic type introduced". A vertical navigation menu on the left side of the slide lists the following items: "Module 27", "Partha Pratim Das", "Objectives & Outline", "Binding", "Static Binding", "Dynamic Binding", "Polymorphic Type", and "Summary". The footer of the slide contains the text "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "13".

Module Summary

- Static and Dynamic Binding are discussed in depth
- Polymorphic type introduced

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

To summarize, we have taken a deeper look into static and dynamic binding, and tried to understand the polymorphic type. And in the next module, we will continue our discussions on various specific issues that arise with the polymorphic types.