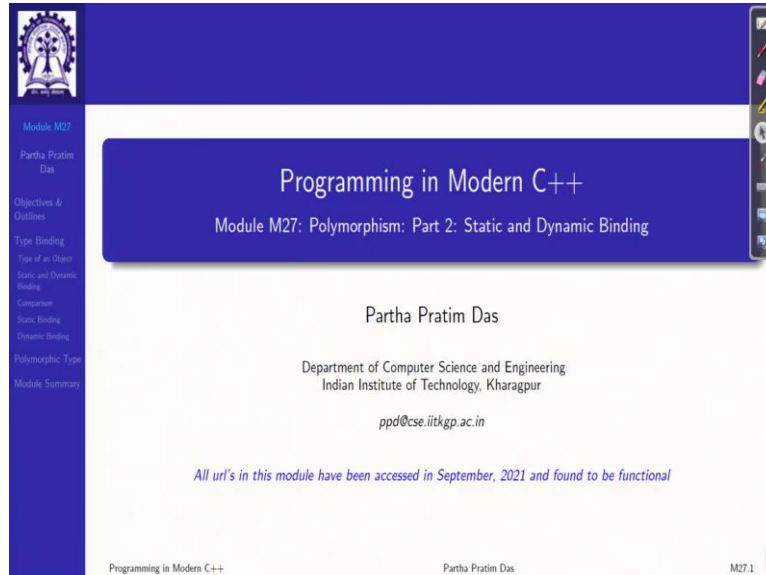
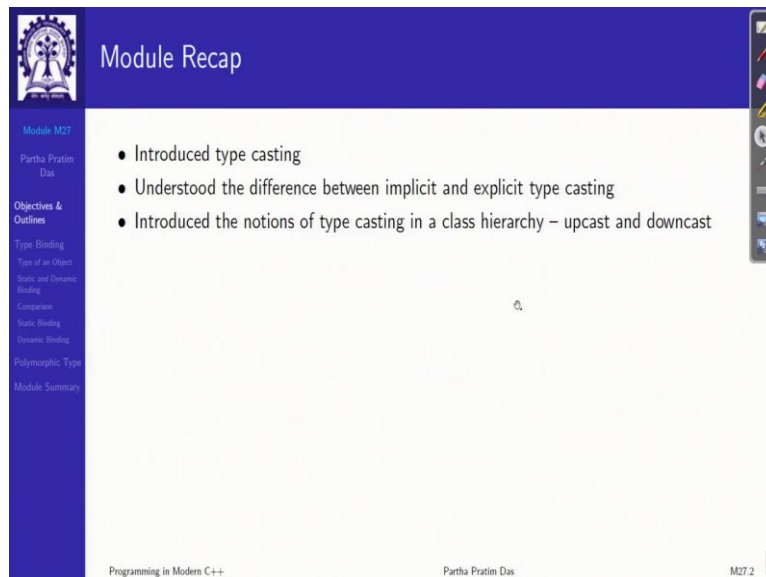


Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture - 27
Polymorphism: Part 2: Static and Dynamic Binding

(Refer Slide Time: 00:35)



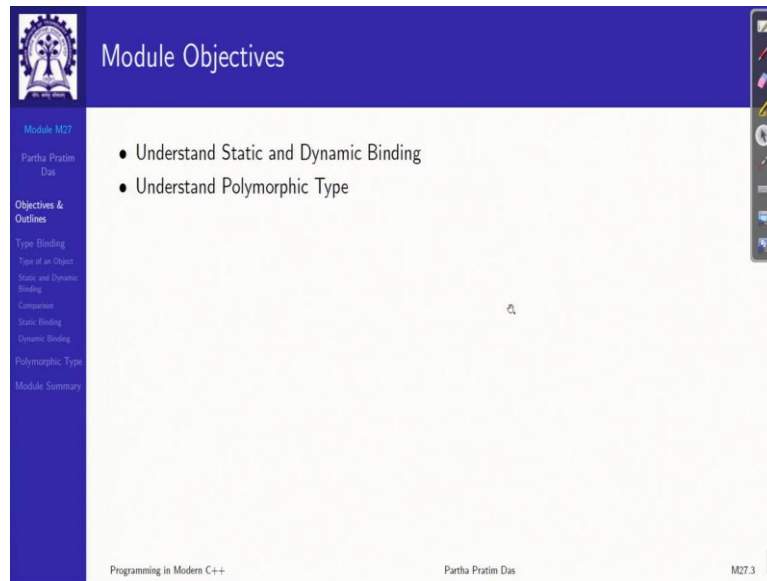
The slide features a blue header with the IIT KGP logo on the left and a navigation toolbar on the right. The main content area is white with a blue title bar. The title bar contains the text "Programming in Modern C++" and "Module M27: Polymorphism: Part 2: Static and Dynamic Binding". Below the title bar, the presenter's name "Partha Pratim Das" is centered, followed by his affiliation: "Department of Computer Science and Engineering, Indian Institute of Technology, Kharagpur" and his email "ppd@cse.iitkgp.ac.in". A note at the bottom states: "All url's in this module have been accessed in September, 2021 and found to be functional". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M27.1".



The slide features a blue header with the IIT KGP logo on the left and a navigation toolbar on the right. The main content area is white with a blue title bar. The title bar contains the text "Module Recap". Below the title bar, a bulleted list is centered: "• Introduced type casting", "• Understood the difference between implicit and explicit type casting", and "• Introduced the notions of type casting in a class hierarchy – upcast and downcast". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M27.2".

Welcome to Programming in Modern C++. We are in week six and we are going to discuss module 27. In the last module, we have introduced the notion of type casting, implicit and explicit typecasting, basic rules for casting for the built in types, numerical as well as pointer and the rules for casting between non-related types and classes on a hierarchy particularly up cast and downcast.

(Refer Slide Time: 1:03)



Module Objectives


- Understand Static and Dynamic Binding
- Understand Polymorphic Type

Module M27
Partha Pratim Das

Objectives & Outlines

Type Binding
Type of an Object
Static and Dynamic Binding
Comparison
Static Binding
Dynamic Binding
Polymorphic Type
Module Summary

Programming in Modern C++ Partha Pratim Das M27.3



Module Outline

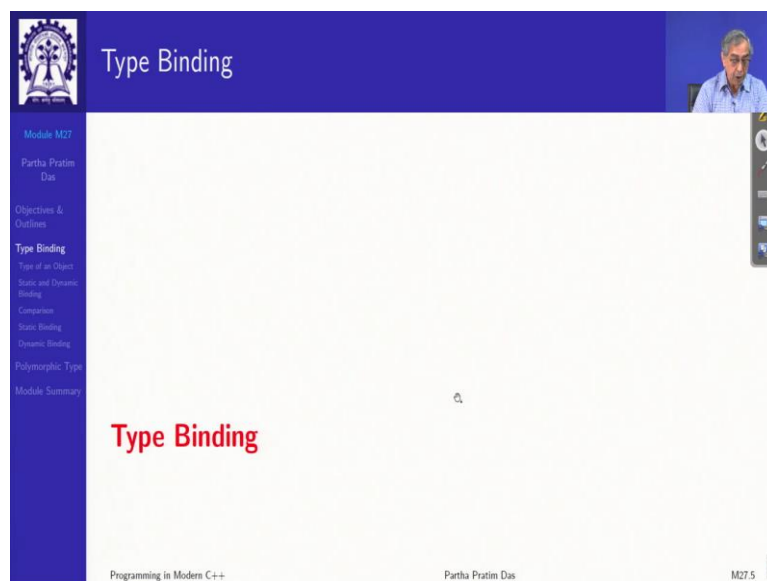
- 1 Objectives & Outlines
- 2 Type Binding
 - Type of an Object
 - Static and Dynamic Binding
 - Comparison of Static and Dynamic Binding
 - Static Binding
 - Dynamic Binding
- 3 Polymorphic Type
- 4 Module Summary

Module M27
Partha Pratim Das

Objectives & Outlines

Type Binding
Type of an Object
Static and Dynamic Binding
Comparison
Static Binding
Dynamic Binding
Polymorphic Type
Module Summary

Programming in Modern C++ Partha Pratim Das M27.4



Type Binding

Type Binding

Module M27
Partha Pratim Das

Objectives & Outlines

Type Binding
Type of an Object
Static and Dynamic Binding
Comparison
Static Binding
Dynamic Binding
Polymorphic Type
Module Summary

Programming in Modern C++ Partha Pratim Das M27.5

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 **refers at run-time**
- 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*
}

```

Handwritten diagrams include: a circle around 'int x;' with an arrow pointing to the code; a class hierarchy diagram showing 'A' at the top and 'B' below it with an arrow pointing from 'B' to 'A'; and a circle around 'p' in the code with an arrow pointing to the 'new B' expression.

In the module today, we will discuss about static and dynamic binding and what is polymorphic type, it follows from the casting rules but we will have to learn about few basic notions first. This is the outline as will be available on the left. So, first we talk about type binding and this will be a little bit of new stuff for you all because in C, when we talk of we talk of primarily of static type that is, we say `int x` and you will know that what is the type of `x`, that is integer.

So, the compiler knows that accordingly compiler allocates memory, accordingly compiler allows operations, the casting and so on so forth. So, everything is primarily around this. This type that you have is known as the static type. The type that you have declared an object with and compiler sees this type.

In contrast, we also have a dynamic type which is not, may not be what you have declared it, declare that object for but it is the type of the object that actually exist or refers to at the run time. So, since it is at the run time the compiler does not see the dynamic type. So, here I have a very small example to illustrate this. There are two classes A, and B is derived from A, a specialization of that. I have A type pointer and I am doing `new B`, that is constructing an object of the B class type.

So, if I think of the type of `p`, it is declared as `A*`. So, the static type of `p` is `A*`, that is it is compiler knows that `p` will point to A type object but on the hierarchy, I have allowed an up-cast and actually what it points to is a B type object. So, the dynamic type of `p` is `B*`, run time type is `B*`, so this is the basic difference.

That if the compiler knew that the static type which is pointing to an A type object whereas in the allocation at the runtime or through assignments and so on, the type of the object that it is actually pointing to is a B type object. So, the runtime type and the compile time type differ. The dynamic type and the static type differ and that is where a big story of polymorphism in C++ starts.

(Refer Slide Time: 4:27)

The slide is titled "Static and Dynamic Binding" and features a blue header with a logo on the left and a small video inset of the presenter on the right. The main content area is white with blue and green text. It contains two bullet points: one for static binding (early binding) and one for dynamic binding (late binding). A diagram at the bottom shows "Binding" branching into "Early Binding" and "Late 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*

Binding

- Early Binding (Function Overloading, Operator Overloading, ...)
- Late Binding (Virtual Functions)

Programming in Modern C++ Partha Pratim Das M27.7

So, this type assignment to a variable is known as binding. So, static I mean, what I meant is deciding that what is the type of an object that process is called binding, your binding as if you have an object you have multiple types you bind it the object to one specific type. So, a static binding is when it binds based on the static type of the objects. So, this is all compile time.

So, your normal function calls overloaded functions, overloaded operations are different examples of static binding because the whole thing is based on the static type of the parameters and different objects involved and the compiler knows all of them and accordingly the compiler binds. In contrast we have a dynamic binding in C++, when a function invocation binds to the function definition based on the dynamic type of the object. You have seen that the dynamic type of the object could be different and so this is done at run time.

Function pointers, we have seen is an example of dynamic type which can be kind of simulated in C but more explicit in C++. Virtual functions are really the dynamic binding mechanism in C++. Since static binding is done in the compiler it is called early binding,

since dynamic binding happens at the runtime it is called late binding. So, this is the basic binding notions.

(Refer Slide Time: 6:19)

Basis	Static Binding	Dynamic Binding
<ul style="list-style-type: none"> • Event Occurrence • Information 	<ul style="list-style-type: none"> • Events occur at <i>compile time</i> – <i>Static Binding</i> • All information needed to call a function is known at <i>compile time</i> 	<ul style="list-style-type: none"> • Events occur at <i>run time</i> – <i>Dynamic Binding</i> • All information needed to call a function is known only at <i>run time</i>
<ul style="list-style-type: none"> • Advantage • Time • Actual Object • Alternate name • Example 	<ul style="list-style-type: none"> • <i>Efficiency</i> • <i>Fast execution</i> • Actual object is <i>not used for binding</i> • <i>Early Binding</i> • <i>Method Overloading</i> Normal function call, Overloaded function call, Overloaded operators 	<ul style="list-style-type: none"> • <i>Flexibility</i> • <i>Slow execution</i> • Actual object is <i>used for binding</i> • <i>Late Binding</i> • <i>Method Overriding</i> Virtual functions

So, if I quickly compare then between static and dynamic binding then naturally the events for static binding happen at compile time and those for dynamic binding happens at runtime. All information for a call is known at compile time for static binding but all information is not known for dynamic binding, they are known only at the runtime as we will see the static binding is efficient for any function call dynamic binding is flexible but not as efficient.

Naturally leading from that is static binding has faster execution compared to the dynamic binding. In the static binding the actual object is not used for binding, only its type is used whereas in dynamic binding the actual object is used because the type of that object at the runtime is what is deciding the binding. The examples include method overloading, normal function call, overloaded functions, overloaded operators these are all statically bound whereas method overriding and virtual functions may be that will be dynamically bound. So, these are the basic differences.

(Refer Slide Time: 7:41)

The slide is titled "Static Binding" and is divided into two columns: "Inherited Method" and "Overridden Method".

Inherited Method:

```
#include<iostream>
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() ✓
    ✓d.f(); // B::f() ----- Inherited
    ✓d.g(); // D::g() ----- Added
}
```

Overridden 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.f(); // B::f() ✓
    ✓d.f(); // D::f() ----- Overridden
    // masks the base class function
}
```

Notes:

- Object d of derived class inherits the base class function f() and has its own function g()
- Function calls are resolved at compile time based on static type
- 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
- 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

Navigation sidebar on the left includes: Module M27, Partha Pratim Das, Objectives & Outlines, Type Binding, Type of an Object, Static and Dynamic Binding, Comparison, Static Binding, Dynamic Binding, Polymorphic Types, Module Summary.

Footer: Programming in Modern C++ | Partha Pratim Das | M27.9

So, let us quickly look at we have seen this kind of similar slides before. So let us quickly take a look of a hierarchy, B is a base class, D is the derived class, B defines a function f(), D defines another function g(). So, if I now have two objects b and d then naturally b.f() will call the function in the class B, d.f() will call the function in the class B again because it has inherited it and naturally d.g() will call the function that you have added.

If I use overriding, I have a function f() in B and in the derived class I have overridden the function f(). So, if I do, b.f(), it will be this will call the B class function, if I do d.f(), then this will call the D class function because I have overridden it.

So, when I override, then I have actually lost the original function that I had in the base class, that has got must. There is no way to call that function anymore, unless I explicitly put the scope and write that. So, if I just, if I want to inherit a function from my base class and just overload it, I will not be able to do that because the moment I overload that will hide the previous function.

(Refer Slide Time: 9:27)

The slide is titled "Member Functions: Overrides and Overloads: RECAP (M". It is divided into two columns: "Inheritance" and "Override & Overload".

Inheritance:

```
class B { public: // Base Class
    void f(int i);
    void g(int i);
};
class D: public B { public: // Derived Class
    // 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)
```

Override & Overload:

```
class B { public: // Base Class
    void f(int);
    void g(int i);
};
class D: public B { public: // Derived Class
    // 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 D::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)
```

Legend:

- D::f(int) overrides B::f(int)
- D::f(string&) overloads B::f(int)

Programming in Modern C++ Partha Pratim Das M27.10

Let us see an example, we will come to that example. Now this is from module 22. Just I am asking you to go and revisit that, f() and g() are in base class. They are inherited in the derived class and naturally everything that you call from b or d is in the base class function. Here you have b and f() and g() in the base class, you have overridden f() here, same signature you have overloaded f() here, you have inherited g() just like that, did not do anything with it and introduced a new function.

So, when you call from b it is all B class objects, when you call from f() from d you get the overloaded f(), overridden f(3), I am sorry overridden f(3) which is the D class function and g() of course is just inherited, so you will get to see the g() of the B class. And if you call the overloaded f(), then naturally you will get the f() in the D class and so on. So, you have already seen that, this is just a you know recapitulation reminder.

(Refer Slide Time: 10:45)

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

class A { public:
    void f() { }
};

class B : public A { public:
    // To overload, rather than hide the base class function f(),
    // it is introduced into the scope of B with a using declaration
    using A::f;
    void f(int) { } // Overloads f()
};

int main() {
    B b; // function calls resolved at compile time

    b.f(3); // 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 we were talking about what happens if I inherit a function from the base class and then override it or want to overload it. Say for example, I have a class A with a function f() and B is inherited from that. Now if I just do f(int), it will overload the function f() in B. But you cannot see, is what you cannot see here is there is a function f() available here which comes from the scope of B. So, when you put a different signature for f(), you are actually overloading that function.

As you do that you will not be able to call the original function you will with b and object b you will not be able to call b.f() because it does not exist, you have already hidden it, masked it. So, to avoid this problem, what is introduced is a new use of the using keyword. You already have seen that in terms of name space is a different context where we are doing using.

So, it says using a::f(). What does it mean? It means that do preserve the function f() from the base class and treat this as a overload. So, when you want to do that, then you do this using keyword and give the fully specialized, fully qualified name of the function that you want to use from your parent context.

So, with that you will now be able to call your overloaded function and as you do b.f() you actually call the function from the base class A. So, this is how you can make the overridden functions or overloaded functions from the base class, also available to your class if you need. So, this was about static binding.

(Refer Slide Time: 13:08)

Non-Virtual Method	Virtual Method
<pre>#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() }</pre>	<pre>#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() }</pre>
<ul style="list-style-type: none">• <code>p->f()</code> always binds to <code>B::f()</code>• Binding is decided by the <i>type of pointer</i>• Static Binding	<ul style="list-style-type: none">• <code>p->f()</code> binds to <code>B::f()</code> for a B object, and to <code>D::f()</code> for a D object• Binding is decided by the <i>type of object</i>• Dynamic Binding

Now let us talk about dynamic binding which is, so this is B is a D which overrides the function, I am sorry D is a B that is D is a derived class, B is the base class and I am overriding the function f(). I have overwritten it here and when I make the call, naturally just look at what I am doing. There are two b, two objects b and d and a B type pointer which is a base type pointer. Now, if I assign the address of the base class object and invoke, it will invoke b::f().

No issues, if I assign the derived class address of the derived class object to the base class pointer and then try to call f(), it will still call the base class function because this is done by the compiler at the static time, at the compile time. So, the compiler knows in p here, in resolving p here compiler knows that it is of B type. So, it binds with this function, it does not bind with this function and but probably d being an object of the class D, you probably would have wanted to use the other function in the class D.

It is of course your choice, when this happens this is called static binding. Now, how can we change this scenario. There is a very simple way to change this, all that you need to do is to put a keyword virtual before the function signature, everything else is same. Instances, this up to this point there is no difference but if you do this, p pointer, p you have set to the address of a D object and you do p-> f(), it does not call B::f(). Instead it calls D::f(). That is the compiler does not resolve that since p is a class B, base class type pointer.

So, if that is trying to do will have to come from the base class, compiler does not resolve that. Compiler does some mechanism which we will subsequently talk of, which allows that at the runtime when the pointer actually is pointing to an object which is a derived class

object. Since, it is pointing to a derived class object, it will call the function, overridden function in the derived class not of the base class and this is what is dynamic binding. So, such functions that you write is known as virtual function and that is the backbone of dynamic binding polymorphism in C++.

(Refer Slide Time: 16:30)

The slide displays the following C++ code:

```
#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

    B &rb = b;
    B &rd = 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
    pd->f(); // B::f() -- Static Binding
    pd->g(); // D::g() -- Dynamic Binding

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

    return 0;
}
```

The slide also features a sidebar with navigation options and a small video inset of a speaker in the top right corner.

So, just to compare how do they look like in terms of static and dynamic binding, I have a function f() here, which is overridden in the derived class, which is non virtual and I have a g() which is virtual. So, you can do this, you can have a pointer pb which holds the base class object pointer pd, which holds the derived class object but pd is of type base class. So, there is an up cast involved here.

Similarly, just to illustrate the same mechanism can be done for reference. I have also created two references, one is a reference to the object b and the other is a reference to object d but the reference is of type B, base class type, again an up cast. Now if I call the functions with the object directly, then they get called according to that type. So, calls from b goes to B class function calls from d go to D class function there is no surprise of that.

Now suppose, I call these functions using pointer, pb->f(). Now what is f()? f() is a non-virtual function. So, it is statically bound. So, which function it will call is decided by the type of pb which is the base class. So, it calls B::f, we have a static binding. But what happens if I call pb->g().

Now, g() is a virtual function. So, it will be dynamically bound, so it depends on what pb is pointing to. What pb is pointing to? pb is pointing to a B class object. So, since pb is pointing

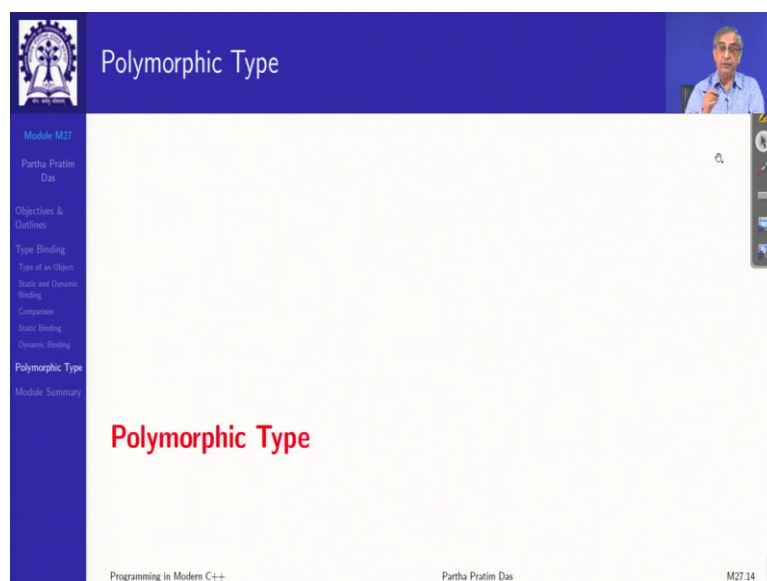
to a B class object at the runtime. It will invoke the B class function, B::f but this is happening through dynamic binding.

Now take a look as to when we have pd, what is pd? pd is of type base class but it is actually pointing to a derived class object. So, when it does that for a non-virtual function, you have static binding so it is decided by the type of pd and B::f() is called. But when you call the virtual function through pd though pd is of type base class but it is actually pointing to a derived class object. So, it resolves at the run time that it will call the g() function in the derived class of which the object is currently there. So, this is the illustration of the dynamic binding.


Similar behaviour will be seen also for the references. So, the references and these calls if you tally, you will see exactly the same mechanism. So, I mean, we get to see that dynamic binding is possible with only with virtual functions that are overridden and if you are making the access to the function through a pointer or through a reference.

Otherwise it is static binding, if you are dealing with non-virtual functions then it is always statically bound. If you are invoking through the object directly, it is always statically bound but only with pointers and with references and virtual functions, you will have this dynamic binding behaviour. This is a very, again, a very profound one slide summary of the difference so please mark this slide and keep on referring to it whenever you have some confusion.


(Refer Slide Time: 20:33)



The image shows a presentation slide with a dark blue header and footer. The header contains a logo on the left, the title "Polymorphic Type" in white, and a small video inset of a speaker on the right. The main content area is white with the text "Polymorphic Type" in red. A left sidebar lists navigation items: "Module M27", "Partha Pratim Das", "Objectives & Outlines", "Type Binding", "Type of an Object", "Static and Dynamic Binding", "Comparison", "Static Binding", "Dynamic Binding", "Polymorphic Type", and "Module Summary". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M27.14".



Polymorphic Type: Virtual Functions



Module M27

Partha Pratim Das

Objectives & Outlines

Type Binding

Type of an Object

Static and Dynamic Binding

Comparison

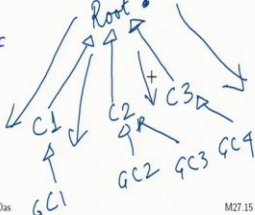
Static Binding

Dynamic Binding

Polymorphic Type

Module Summary

- *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



Programming in Modern C++
Partha Pratim Das
M27.15

Now this dynamic binding leads to polymorphic and non-polymorphic type notions of polymorphic and non-polymorphic type. Polymorphic again, I would reiterate means that something that can multiply morph or multiply interpreted. Poly is many, morph is change. So, same thing but can be seen in multiple ways.

Dynamic binding is basically giving as the polymorphism. So, as we have noted dynamic binding is possible only for pointer and reference data types and for member functions that are declared as virtual in the base class. Such member functions are called virtual functions. So, we say access by pointer or reference and available for virtual functions only.

If a member function is declared as virtual, it can be overridden in the derived class. As you override non-virtual functions as well but you can do the same thing for virtual functions. If a member function is not virtual and it is redefined in the derived class, then the later definition hides the former one. We have already seen that, how by overriding you actually lose the earlier definition.

But unless you explicitly do using and want to use it but there are complications if you try to do that often. But in dynamic binding that is the mechanism, that you are overriding but all of the functions from the base to the current class, all of these overridden functions that are virtual will be available based on the actual type of the object that is being used in the invocation of the function through the pointer or a reference.

So, given all this any class which contains a virtual member function at least one, it may be many but if a class contains at least one virtual member function which could be either by definition that it is declared as virtual or it has got by inheritance because its parent had a

virtual function and being a specialization, it has inherited from the parent, virtual functions are always inherited as virtual functions. Then such a class or such a type is known as polymorphic.

So, it will lead to dynamic binding for that class. So, a hierarchy may be polymorphic or it may be non-polymorphic. I may not have any virtual function from the at the base, so that the whole hierarchy is non-polymorphic. The reality is well, so far whatever we have been discussing with about the hierarchy were naturally non-polymorphing but in the actual practice a non-polymorphic inheritance hierarchy has little value.

It is used only for certain specific cases but most often you will see that the real power of object oriented programming in C++ will come when you have polymorphic types, that is you have a hierarchy where you have say root and you have inheritance. There could be inheritance, so this is child one, this is child two, this is child three classes, this is grandchild one, this is grandchild two, this is grandchild three, this is grandchild four and so on.

Now if I have a virtual function here, one or more virtual functions those will all be available all through this hierarchy. So, that is what is the meaning of a polymorphic type or a polymorphic hierarchy which will be the main stay of object oriented design for C++.

(Refer Slide Time: 24:59)

Polymorphism Rule

```

#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();
}

```

Handwritten Hierarchy Diagram: A → B → C

Function Call Results:

- A::f() ✓
- C::g() ✓
- A::h() ✓
- B::f() ✓
- C::g() ✓
- C::h() ✓

So, that is just a quick set of illustrations. Here is a class a, so this is the hierarchy diagram, class a which is a base class, which has one virtual function g(), f() and h() are non-virtual. Then it specializes into b where f() continues to be non-virtual, g() is virtual. You would asked, why is g() virtually? You have not written virtual here?

The rule is once a function is virtual in a class then in all its derived classes, it is virtual. You cannot change that. But you can do the reverse, for example in here h() is non virtual but what I have done while overriding I have made it virtual. So, in class A, h() is non-virtual but in class B, h() is virtual.

So, B becomes a new route for the subsequent classes where h() will be treated as a virtual function. Now I have C, which is overriding all three of them. Now f is non-virtual because it was non-virtual, g() and h() both have become virtual in B, so it will be virtual in C as well. So, with that, if I create a C object and put the address to a B type pointer and have another pointer p where I have up cast this pointer, so p is a A-type pointer, q is a B-type pointer and the actual object both of them are pointing to is a C-type object.

So, what will happen? If I do p->f(), what is f()? F() is a non-virtual function. So, what it will do? It will decide by the type of the pointer, p is A-type. So, this is what you get, A::f(). What if you do p pointed g()? P->()g is a virtual function and p is currently pointing to a C-type object, g() is a virtual function in class A, p is of class A. So, in class A, it checks whether this function is virtual or non-virtual, decides whether it will be statically bound or dynamically bounded. It decided for f() earlier, now it is deciding for g() it finds that it is virtual. So, it is dynamically bound.

So, it does not decide based on the type of p, it decides based on the type of the object it is actually pointing to which is C. So, what it calls? It calls the g() function of class C. What happens when I do p->h()? H() is non-virtual. So, it has to be statically bound. So, it will call A::h().

Now, let us think about the pointer q, if I do q->f(), q is of type B. So, it will look for this B class for resolving whether it should statically bind or it should dynamically bind. So, it finds that f() is non-virtual here, it has not inherited virtualness from the parent either. So, it will bind by the type of q, the type of the pointer. So, it type of the pointer is B, B* so it calls b::f().

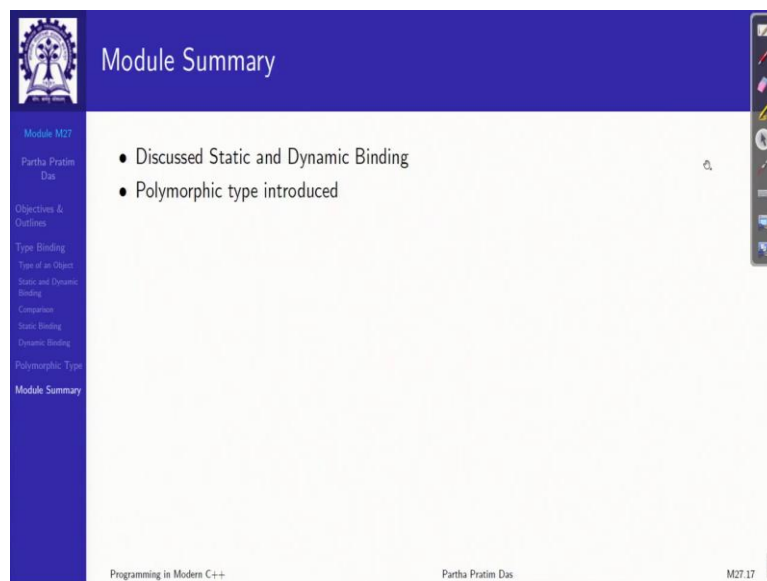
Now if I look at g(), what will happen? G() is not explicitly written here but is inherited the virtual function property from its parent. So, it is a virtual function so it will not be statically bound. It will be bound to the current type of the object q is pointing to which is a C-type object. Therefore it calls the g() function of the class C. Finally a it is q->h(), which was not inherited as virtual but has been defined as virtual in the class B.

So, q will treat this as a virtual function now, so q will not this binding the compiler will not generate based on the type of q, if we would have done that you would have called the function B::h(). But it will not do that because the actual object is of type C and this is a case of dynamic binding. So, it will call the function h() of class C, C::h().

So, this is, again one slide summary rules I would say, that you can use to quickly refer to any time you have any issue in understanding this. The example is worked out with pointers, the same example can be done with the reference also. You should run this, check this out, make changes. The basic idea is once a virtual downwards its always virtual. But I can take a non-virtual function and make it virtual in a class but I cannot do the reverse. The virtualness cannot be erased but it can be added and when I resolve for the binding, the pointer of the reference looks at the class of the type it is defined by.

So, a pointer to a will always look for the status of the functions defined in a. Whether they are virtual, whether they are non-virtual and based on that for virtual functions they will do, it will do a dynamic binding for non-virtual function it will be static binding.

(Refer Slide Time: 32:03)



If you understand this rule then the whole of polymorphism or so to say handling of polymorphic hierarchy in C++ which is the core (core, core) of object oriented design in C++ will be like an easy cake for you.

So, here we have discussed about static and dynamic binding and we have just introduced the polymorphic type. We will talk more about that in the coming modules. Thank you very much for your attention and we will meet in the next module.