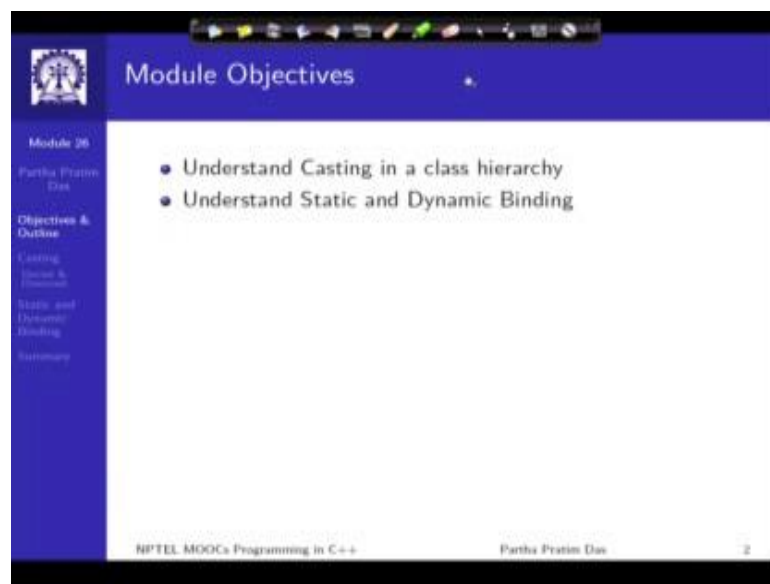


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 41
Dynamic Binding: Part I

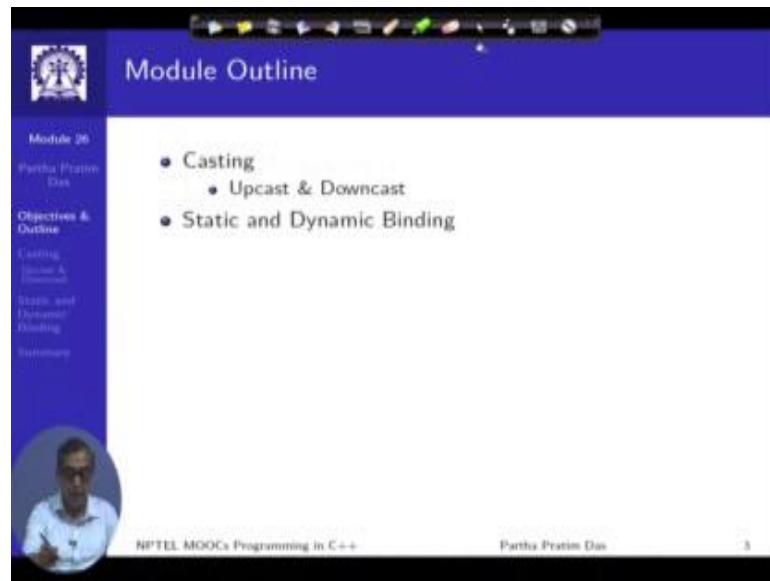
Welcome to module 26 of Programming in C++.

(Refer Slide Time: 00:21)



We have seen the inheritance structure in C++. We will next look into one of the or possibly the most powerful feature of C++ to support object-oriented implementation which is commonly known as the name of polymorphism. But before we can go in depth in terms of the discussion of polymorphism, we need to prepare ourselves with some more understanding in terms of the casting that can happen in a class hierarchy. And the core differences between the static and dynamic binding. So, this dynamic binding discussion will cover couple of modules and this is the first one and will lead us to the total understanding of the polymorphism in C++.

(Refer Slide Time: 01:32)



The slide shows a presentation interface with a blue header and a sidebar on the left. The main content area is white and contains a bulleted list of topics. The sidebar lists 'Module 26', 'Partha Pratim Das', 'Objectives & Outline', 'Casting', 'Static and Dynamic Binding', and 'Summary'. A small circular video inset of the presenter is in the bottom left corner. The footer contains 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '3'.

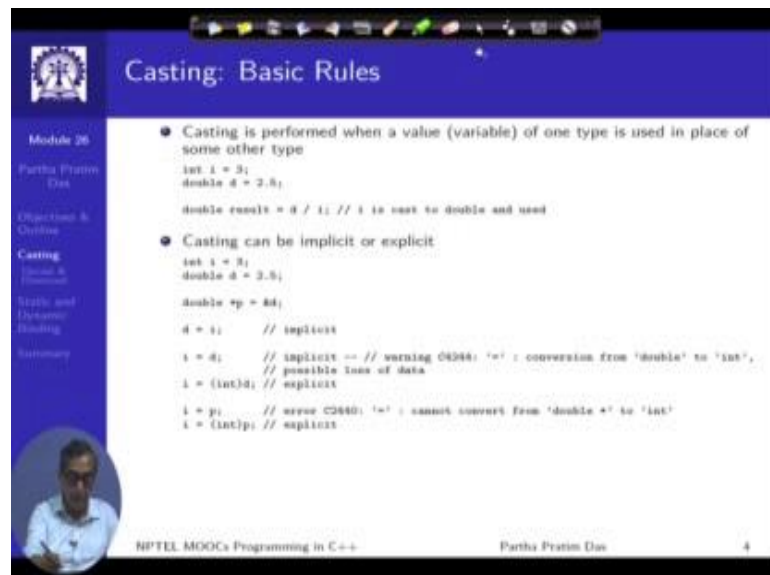
Module Outline

- Casting
 - Upcast & Downcast
- Static and Dynamic Binding

NPTEL MOOCs Programming in C++ Partha Pratim Das 3

So, the outline is casting and static and dynamic binding it will available on the left of your every screen.

(Refer Slide Time: 01:42)



The slide shows a presentation interface with a blue header and a sidebar on the left. The main content area is white and contains a bulleted list of rules for casting, followed by C++ code examples. The sidebar lists 'Module 26', 'Partha Pratim Das', 'Objectives & Outline', 'Casting', 'Static and Dynamic Binding', and 'Summary'. A small circular video inset of the presenter is in the bottom left corner. The footer contains 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '4'.

Casting: Basic Rules

- Casting is performed when a value (variable) of one type is used in place of some other type

```
int i = 3;  
double d = 2.5;  
  
double result = d / i; // i is cast to double and used
```
- Casting can be implicit or explicit

```
int i = 3;  
double d = 2.5;  
  
double ep = 84;  
d = i; // explicit  
  
i = d; // explicit -- // warning C4244: '=' : conversion from 'double' to 'int',  
// possible loss of data  
i = (int)d; // explicit  
  
i = pi; // error C2440: '=' : cannot convert from 'double *' to 'int'  
i = (int)pi; // explicit
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, let us talk about casting. We all know casting, you all know C. So, casting you know is basically performed when a value of one type, certain type is used in place some other

type is used in the context of some other type. So, for example, here *i* is an integer value variable, *d* is a double variable and I am writing *d* slash *i*. So, if I say *d* then if I say slash then the meaning of this slash is a division of floating point numbers. So, what is expected here is also a double value but what I have provided, I have provided an integer value. So, this still works in C how does it work? So there will have to be some process by which this integer value is kind of converted to a double value and then used. So, this mechanism is the mechanism of casting that *i* is cast to double and then it is used. So, this is a simplest notion that we have seen and in C this is particularly called particularly referred to as a mixed mode operation which we all know.

Now, let us go for the start developing little bit more on it we know that the casting can be implicit or it could be explicit. For example, look at first look at the types of the variables *i* is an int variable, *d* is a double variable and *p* is a pointer variable pointer to double. So, if I make an assignment of *i* to *d* then, this is double and this is int. So, certainly they are not of the same type assignment should not be possible, but implicitly it will cast the integer to the double and allow me to do this. I can have the reverse the double being assigned to *i* this will also allowed, but with a warning for the simple reason because double is a much bigger data type, whereas int is a smaller data type so, some data will be lost.

In contrast, I could use what is called the C style casting that is put the required type with in parenthesis before the available value and use that, this is known as explicit casting. And now the compiler will not shout. So, you can see that if I try do something like *p* is a pointer and *i* is an integer, if I want to assign the *p* to *i* set is an error your certainly you cannot take a double pointer and use it as an int. But if I explicitly cast then the C compiler will allow that. These are basic rule of casting.

(Refer Slide Time: 04:53)

Casting: Basic Rules

- (Implicit) Casting between unrelated classes is not permitted.

```
class A { int i; };
class B { double d; };

A a;
B b;

A *p = &a;
B *q = &b;

a = b; // error C2679: binary '=' : no operator found
// which takes a right-hand operand of type 'main::B'

a = (A)b; // error C2440: 'type cast' : cannot convert from 'main::B' to 'main::A'
b = a; // error C2679: binary '+' : no operator found
// which takes a right-hand operand of type 'main::A'

b = (B)a; // error C2440: 'type cast' : cannot convert from 'main::A' to 'main::B'

p = q; // error C2440: '=' : cannot convert from 'main::B *' to 'main::A *'
q = p; // error C2440: '=' : cannot convert from 'main::A *' to 'main::B *'

p = (A*)b; // Forced -- Okay
q = (B*)a; // Forced -- Okay
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 5

Let us move on let say, if we try to now extend this to C++. So, earlier we were talking about built in types only int, pointer, double and so on. Now, I have two classes A and B and I have two objects, I have two pointers one of A type another is of B type which will holds the address of A, and the address of B. So, if I try to make this assignment this assignment is a casting error because what is this assignment saying? This is saying that a is assigned, b means a dot; we all know it now operator assignment b. So, it means that it the class A to which this a belongs, this a belongs to class A must have an operator assignment which takes an operand of type b which does not exist.

If we try to explicitly cast, this will also fail because there is no explicit cast defined. Is not like int, double where you know how to convert an int to a double or double to a int. So, if you try to do them in the reverse way of course, this will fail. If you try to cast the two types of pointers pointer to a into pointer two b this will fail, the other direction this will also fail. So, all of these are casting failure because you are not allowed to do this. The only thing that you can force is you could take the address of b address of b is of type B star and force it to be A star. That is you can convert the pointer of unrelated classes one into to the other, but in a force manner. Naturally, you could not do this p assign q is not allowed. This of B star, this is of A star. You cannot make an assignment, you cannot use. But if you forcefully say that the B star is converted to A star a pointer,

then you can make this assignment. The symmetric one is will also work. So, this is basically that the story of casting between classes when we introduce C++.

(Refer Slide Time: 07:11)

```
● Forced Casting between unrelated classes is dangerous
class A { public: int i; };
class B { public: double d; };

A a;
B b;

a.i = 5;
b.d = 7.2;

A *p = &a;
B *q = &b;

cout << p->i << endl; // prints 5
cout << q->d << endl; // prints 7.2

p = (B*)b;
q = (A*)a;

cout << p->i << endl; // prints -950993400 ----- GARBAGE
cout << q->d << endl; // prints -0.2550647061 ----- GARBAGE
```

Now, if we do forced casting as we just saw between unrelated classes then, the results may be dangerous results may be actually very unpredictable. So, I have just the similar two classes. Now, I have just put in the data members in public. I want to just demonstrate you. So, these data members have been initialized and I have a pointer of type p which keep a type A which keeps the A address the q and I print the values certainly the values are correctly printed. a dot; p pointer i is basically a.i which is 5, q pointer d is 7.2.

Now, suppose I have forcefully cast it. So, if I have forcefully cast it as said that this is what I want. Is this is a b object and the pointer to that I have cast as a A pointer and then I am trying to do this. This is what it prints. If you try this it is not necessary that it will print this. In my system it printed this, but it will certainly print some garbage. Why is this happening? This is happening because if you look into the a object this as a int. let us say we are on a 32 bit system. So, int has 4 bytes in which this value 5 is written. In a b object I have a double which has possibly 8 bytes. Now, if I take the pointer p which is of type A; that means, the pointer if I write p pointer i then it the compiler always know that

it should point 2 and a type of object so that it can take 4 consecutive bytes and make it an integer.

Now, that is what I have violated. What I have done? I have actually taken the p and make it point to the b address. Now here what was written here in 8 bytes the 7.2 was written, but p knows that it is of type A. So, it knows that it has to read 4 bytes and think it is an integer. So, it takes that floating point representation of 7.2 arbitrary reads the possibly the first four bytes and start thinking that it is an integer and prints it as an integer.

When I do the reverse think of when I do the reverse then what I am doing? I have q pointer which is of pointing to B type. So, if I do q pointer d it expects 8 bytes to be representing a double. Now, I have made q point to this object a, which means that it is actually reading 8 bytes of which the first 4 bytes is a integer representation of 5. The next 4 byte, God knows what is there. It is invalid part of memory, it goes there takes that value interpret it is at a floating number and prints. So, that is the basic problem that casting can get you to. So, force casting is a dangerous thing.

(Refer Slide Time: 10:48)

The slide is titled "Casting on a Hierarchy" and contains the following C++ code:

```
class A { };
class B : public A { };

A *pa = 0;
B *pb = 0;
void *pv = 0;

pa = pb; // okay ----- // UPCAST
pb = pa; // error C2440: '=' : cannot convert from 'A *' to 'B *' // DOWNCAST

pv = pa; // okay ----- // Lose the type
pv = pb; // okay ----- // Lose the type
pv = pv; // okay ----- // Lose the type

pv = pa; // error C2440: '=' : cannot convert from 'void *' to 'A *'
pv = pb; // error C2440: '=' : cannot convert from 'void *' to 'B *'
pv = pv; // error C2440: '=' : cannot convert from 'void *' to 'C *'
```

The slide also includes a navigation bar at the top, a sidebar on the left with a table of contents, a small portrait of the speaker, and footer information: "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "7".

Now, let us see how casting will look like if you try to do it on a hierarchy. So, we have class A and class B is a specialization of A. So, I have two pointers of two class types. Now earlier when the types were unrelated it was not possible to assign the pointer of one type in to the point of other, but now I can assign p b to p a, and this will be permitted. But if I try to do the reverse assign p a to p b this will not be permitted. The reason is simple. If I have an A object and think about a B object what does B object have? It is a specialization of A. So, B object internally has an A object which is the base object.

So, if I have p a, which is a pointer to A type, it is a pointer to A type. Then, when I assign p b to it, what was p b? p b is a pointer to the B type it is pointer to this. So, when I take this value and put it to A then p a refers to the same area. But, what this p n know, p n o knows that it is a pointer of A type which means p a knows that it will get an A object. So, what will it do? It will simply not take will not be able take cognizance of the extended part of B, but it will get valid A object which is the base part of the B object and will just give you that object. So, that is quite to do that. You just get a limited view of that.

But if you try to do the other way, if we try to take p b and make it point to the A object then p b things knows that there is a A object as a base part and then there are more things. So, it will try to think that this whole thing is the B object which actually it is not exist only the A object exist. So, this would be something dangerous it will get into violations in terms of the memory. So, that is the reason this is allowed. Let me change the color whereas this is not allowed. And when we are doing this if I go back to the hierarchy diagram then this is the diagram B is a A. So, as I do this I am moving from the direction of B or specialized object to the generalized object so I am going up. So, we say this is up cast which is allowed. But if I try to move down that is if I take a generalized object and think that it is a specialized one then say down cast which will have to be forbidden.

Of course, in the same example I have also shown that what will happen if you have a void pointer. Certainly you can take objects of any type and yes keep that address in the void pointer all that happens is you lose the type. Similarly, you could do the reverse. Try

to do the reverse that is take a I think there is a typo here what these are typo I will just correct that later on in the presentation. So, this is what is meant is if p a is assigned p v or p b is assigned p v then these all will be error because p a is a void star p v is a void star pointer. So, it does not know, where does it point to? It does not know how many fields what size it should point to. So, if you take that and try to interpret as an A, then certainly you have all kinds of dangers looming. So, these will not be allowed, but it is quite ok if you actually take a pointer of any type and put it to a void pointer. Of course, in C++ we will see that there will be badly any use for that.

(Refer Slide Time: 14:58)

Casting on a Hierarchy

- Up-Casting is safe

```

class A { public: int dataA; };
class B : public A { public: int dataB; };

A a;
B b;

a.dataA = 2;
b.dataA = 3;
b.dataB = 5;

A *pa = &a;
B *pb = &b;

cout << pa->dataA, << endl; // prints 2
cout << pb->dataA, << " " << pb->dataB, << endl; // prints 3 5

pa = &b;

cout << pa->dataA, << endl; // prints 2
// cout << pa->dataB, << endl; // error C2039: 'dataB' : is not a member of 'A'

```

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Now, if we look into the up casting we can we are just enhancing the class with data members and if we look into that if we put a value 2 to the a object, that is the data part of A object value 3 and 5. So, this is an object which is 2. This is the b object which is 3 in the, a part and 5. And then if we take their addresses and try to print the fields. We get the 2 and 3, 5. This is printing the a object, this is printing the b object. Now, let say we have done an up cast. This is a situation of up cast I have taken the address of b and put it in a pointer of type A. So, what does p a get to see? p a is pointing here, but it has the knowledge of only a. So, it gets to see only this part. So, what happens is if I try to print p a pointer data it prints this and you get a 3 as expected, but if you try to do this p a pointer d b data B that is if you try to print this certainly the compiler gives you an error

because the compiler knows that p a is a pointer of A type which does not have any data B member and therefore, this will not be allowed.

So, if you up cast there is no situation where you can get into an error situation because either you will be accessing the base part of the class which is just making limited access which is fine or you and the compiler will restrict you from using or accessing any part of the specialized class which actually does not exist. So, up casting is safe. Down cast you can argue very easily that down casting will be risky.

(Refer Slide Time: 17:03)

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

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

    pb->f(); // B:f()
    pb->g(); // B:g()
    pd->f(); // D:f()
    pd->g(); // D:g()

    return 0;
}
```

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

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Given this let me just introduce the basic concept of static and dynamic binding and it will become clear slowly as to why did I discuss about the casting before introducing this? So, I have a very simple situation. Let me get back to red. So, a very simple situation there is a base class and there is a specialized derived class B, class D. We have seen this earlier. So, base class has a function f method f derived class has inherited and then overridden this method. There is nothing special about that. Base class has another function g and derived class as overridden that function as well. Only thing different in this is that in case of the function g we have written another additional word keyword virtual and we will see how that changes the behavior. So, this is the situation always keep that diagram in mind.

So, I have created two instances and we have two pointers p b and p d both of the base class type. So, I keep the address of b in the p b pointer which is normal and I keep the address of d in the pointer p d of the B class type which means I am doing an up cast. There is an object actually here pointer is of this type. So, I am making a representation assignment here. So, I have done up cast. Similarly, the same thing I have written in terms of the reference. This is the reference to the r b is the reference to the b object, r d is a reference to the d object, but the only difference is r d is of type B type reference. So, it will r d will up cast and think of d as if it were a B type object.

In this so this is the setting and in this we are trying to make function calls of all types. So, we are calling for both objects b and d. We will call both functions. So, four combinations and we will call them in three different ways. The first we call the functions using the object. So, b dot f, b dot g, d dot f, d dot g. We know what function will get called b dot f; this function will get called b dot g, this function will get called b dot d, certainly this function will get called. The corresponding function of that class will get called. So, there is no surprise in that.

Now, let us try to make that function calls through pointer. So, p b is a pointer to the b object, p d is a pointer to the d object, but both of them are of B type. So, since they are of B type so if I invoke p b pointer f certainly I expect the function of the B class to be invoked. So, p b pointer f should invoke b colon colon f this function. Similarly the next p b pointer g should invoke the g function; p d pointer f p d also is of type the base class. So, p d also knows out only about these two functions. So, if I do p d pointer f it again invokes the function f of the B class. But something totally surprising happen when you do p d pointer g. Mind you p d is a pointer of this type. It has it is actually currently pointing to an object of this type. So, the type of p d is here. That object actually is here, but when I invoke the method, it somehow is able to find out that it actually has a d object and instead of invoking this function it actually invokes this function. And that is what is known as dynamic binding.

So, what is binding? Binding is given an expression in terms of invocation you decide which function will get called. So, we have talked a binding in the context of overloading also that if there are multiple functions overloaded which particular function will be

bound. So, it is a similar concept. So, given this p pointer p b pointer f or p d pointer g the question that we want to ask as to which particular function will get bound by with this call.

Now, what we see is, in case of the function f the binding is static which means the binding is decided by let me again clear it. So, in case of function f here and here the binding is static which means which function it calls depends on the type of the pointer. Which means something that is statically known that is known at the compile time compiler always know what is the type of the pointer, but the fact is this pointer is having a B object pointing to a B object whereas this pointer is pointing to a D object.

In case of static binding those that is no consideration this is a B type pointer this also is a B type pointer. So, if I invoke f this invokes B colon colon f that is a f method of the base class. This also invokes f method of the base class. Scenario change in the other case where we say we are having dynamic bind if you look into these now again these two are both are pointers of the base type. And this is pointing to the b object, this is pointing to a d object.

Now, we find that when this pointer is pointing to a b object, B colon colon g that is here this function is getting invoked. Where as in a different identical expression where the pointer still is of B star type, but when it invokes g given that the object it actually is pointing to is a d type object now the D colon colon g that is this overridden function in the derived class gets invoked. So, it does not depend on the type of the pointer. So, the compiler has no way to decide as to p b pointer g or p d pointer g which function will it call because it is not getting decided by the type of the pointer, but it is getting decided by actually at the run time. What is the type of the object or what is the object that it is pointing to.

So, the binding between the expression like p d pointer g and the functions B colon colon g or d colon colon g, this binding whether it binds here or it binds here does not depend on the pointer. It depends on the pointed object. It does not depend on p d it depends on what is the type of the pointed object. If the type of pointed object is of b type as it was here b type then, the g method of the base class is invoked. If it is of d type

then the g method of the derived class gets invoked. And therefore, this is called the dynamic binding this is in contrast to the static binding. And what makes the difference? The difference is made by this key word before the function. So, of the two functions this is called a virtual function where I have written the virtual in the front and this is called a non-virtual function.

And if I have a non-virtual function which is what we had earlier. I will have a static binding and if I have a virtual function then i will have a dynamic binding where the if I call that function through a pointer then it will not depend on the type of the pointer, but it will depend on the actual object type of the actual object that the pointer points to at the run time. So, that is a basic difference between static and dynamic binding of course, I am just trying to introduce the semantics to you. We have we are still a little way to go to understand to realize why we are doing this. How will this really help in terms of modeling an implementation, but this is this basic notion of virtual function and dynamic binding is what we want to understand from here.

And in the last section where we use the reference we can see exactly the same behavior. These two are the here the reference are b refers to the b object; r d refers to the d object. For function f it again if I do it through reference I have a static binding, but certainly if I invoke the g function g method for r b and r d because r d is referring to a d object and because r b is referring to a b object. According to dynamic binding the in this case, I get a get the g function of the derived class invoked. In this case I get the g function of the base class invoked. So, if I access the functions as object they will always be static. They will always be based on what that object type is. But if I invoke the methods through functions or through reference then I may have static or dynamic binding depending on whether the member function I am invoking is a non-virtual one where the static binding will happen or when the member function is a virtual one where dynamic binding will happen. So, this was just to introduce the basic notion of binding will build up further on this.

(Refer Slide Time: 29:15)

Module Summary

- Introduced casting and discussed the notions of upcast and downcast
- Introduced Static and Dynamic Binding

NPTEL MOOCs Programming in C++ Partha Pratim Das 38

So, to sum up we have introduced the concept of casting and discussed the basic notion of cast up casting and down casting and seen that up cast is safe and down cast is risky. We have done this because in the next module, we will need to use this notion of casting in the context of binding. After that we have introduced the basic semantics of up static casting and dynamic casting or the basic definition of a virtual function which is a new kind of member function that we are introducing further classes. We will continue discussions on the dynamic binding in the next module.