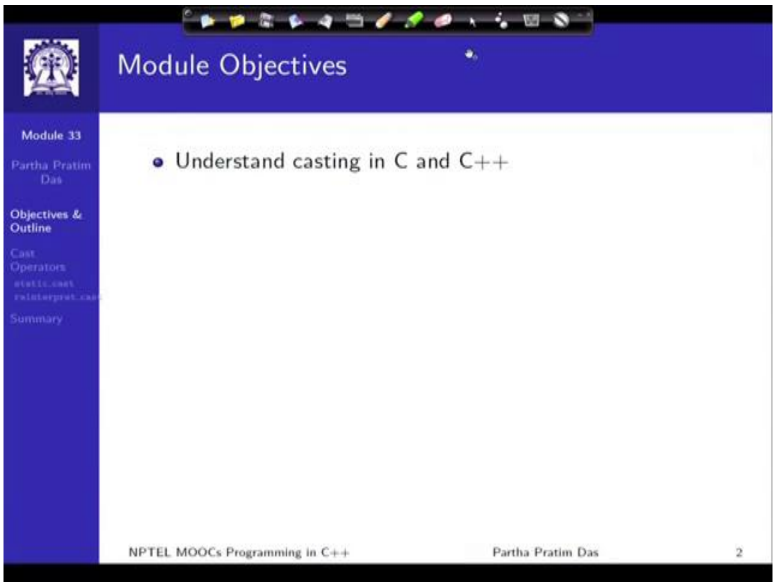
Programming in C++

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

Lecture – 48
Type Casting and Cast Operators: Part – II

Welcome to Module 33 of Programming in C++. We have been discussing about type casting in C and V plus plus, and particularly the cast operators in C++.

(Refer Slide Time: 00:31)



Module Objectives

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

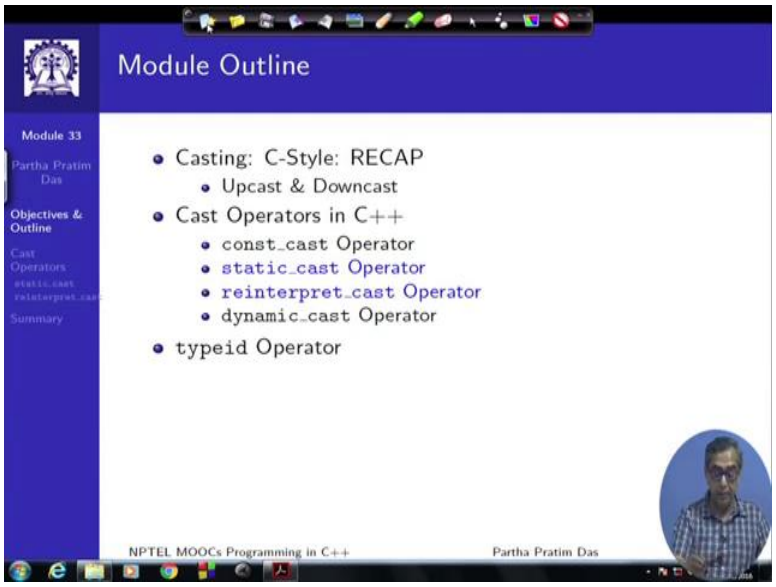
Summary

- Understand casting in C and C++

NPTEL MOOCs Programming in C++ Partha Pratim Das 2

So, we continue with that same objective of understanding casting in C and C++.

(Refer Slide Time: 00:37)



Module Outline

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

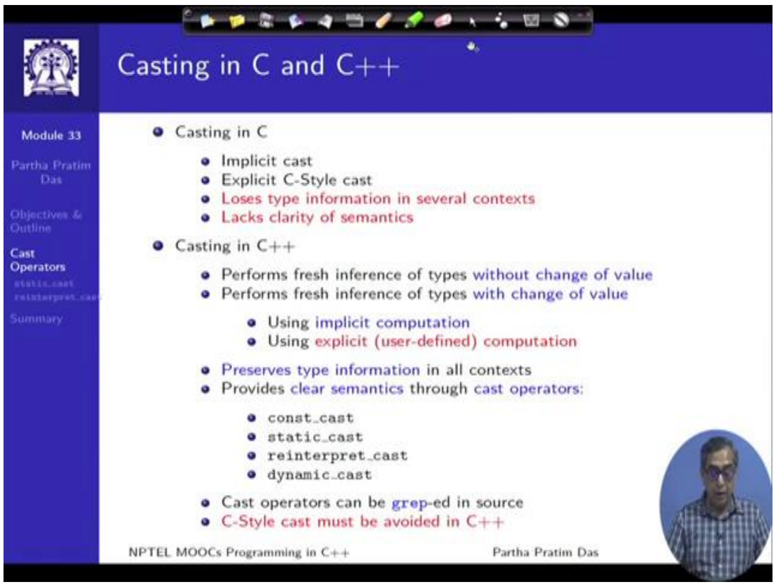
Summary

- Casting: C-Style: RECAP
  - Upcast & Downcast
- Cast Operators in C++
  - `const_cast` Operator
  - `static_cast` Operator
  - `reinterpret_cast` Operator
  - `dynamic_cast` Operator
- `typeid` Operator

NPTEL MOOCs Programming in C++ Partha Pratim Das

In terms of the module outline, I had explained in the last module itself that will cover spread over multiple modules. So, here the blue ones that is these two are enough focus to discuss in the current module - the static cast operator and the reinterpret cast operator.

(Refer Slide Time: 00:57)



Casting in C and C++

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

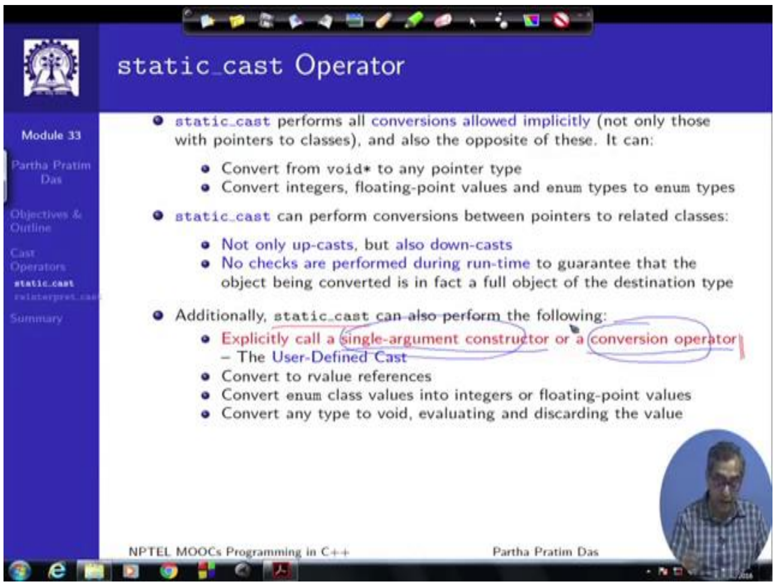
Summary

- Casting in C
  - Implicit cast
  - Explicit C-Style cast
  - Loses type information in several contexts
  - Lacks clarity of semantics
- Casting in C++
  - Performs fresh inference of types without change of value
  - Performs fresh inference of types with change of value
    - Using implicit computation
    - Using explicit (user-defined) computation
  - Preserves type information in all contexts
  - Provides clear semantics through cast operators:
    - `const_cast`
    - `static_cast`
    - `reinterpret_cast`
    - `dynamic_cast`
  - Cast operators can be grep-ed in source
  - C-Style cast must be avoided in C++

NPTEL MOOCs Programming in C++ Partha Pratim Das

We have looked at the basic issues of a casting in C and C++. We have noted that C has the two primary style of implicit and explicit C style casting, and these are the two major issues, it often losses the type information and lacks the clarity of semantics. In contrast, C++ preserves all that; and we will see that how it can introduce user defined computation through casting, and these are all done through cast operators. And we have taken a look into the one type of const cast operator which significantly falls in this category that in most cases when you do const cast. You are just making a fresh inference about the type it was some const used to exist in an object you are trying to create a reference which is a non-constant one. Or some constant was not there, which we are trying to add through a reference or through a pointer, but usually you do not add a new computation of values, but we will see in the next type of casting, static casting, that actual explicit user defined computation will come into play.

(Refer Slide Time: 02:08)



static_cast Operator

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

Summary

- static_cast performs all conversions allowed implicitly (not only those with pointers to classes), and also the opposite of these. It can:
  - Convert from void* to any pointer type
  - Convert integers, floating-point values and enum types to enum types
- static_cast can perform conversions between pointers to related classes:
  - Not only up-casts, but also down-casts
  - No checks are performed during run-time to guarantee that the object being converted is in fact a full object of the destination type
- Additionally, static_cast can also perform the following:
  - Explicitly call a single-argument constructor or a conversion operator – The User-Defined Cast
  - Convert to rvalue references
  - Convert enum class values into integers or floating-point values
  - Convert any type to void, evaluating and discarding the value

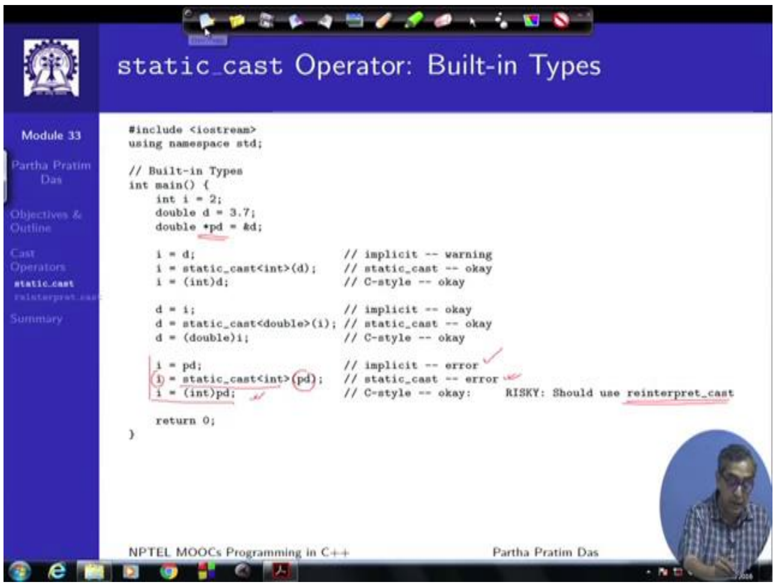
NPTEL MOOCs Programming in C++ Partha Pratim Das

So, with that we get start started with the static cast operator. The first thing about static cast operator is kind of this static cast operator is first of all it is static cast, that names tells us that this deals with casting which can be decided a compile time, anything that is static means it is static time. So, it is something that you can decide at the compile time. And the static cast performs or covers all conversions that are allowed implicitly, not only to those relating to pointer their reverse as well as conversion of void start to any pointer type or all of these different kinds of you know built in conversion, convert between integers, integer floating point, enum types to enum types, integers enum types and so on all these can be done through the static cast operator.

Second, it can further perform conversion between pointers to related types that is pointers which point to two different classes on a hierarchy. And it can not only do up-cast, but it can also do down-cast, of course, at a risk. So, the risk is what is stated next is, when you up-cast you know that you are always safe, because you have more information and you are looking at only the base part of it restricted part of this. When you down cast, the risk is that your actual pointer, your actual object may not be a specialized object, it may not have the additional information that you expect in terms of the specialized class, but you were still down casting. So, static cast run through that risk and should not normally be used for down casting, because it does not check at the run time that whether you appropriately have an object which is of the specialized type, but it still allows you to do that cast.

Further and that this is the most interesting thing is the static cast actually can explicitly call a single argument constructor or a conversion operator. So, you can call a single argument constructor or a conversion operator; and certainly since it can call these, these are codes that the user can write. So, static cast can often be used to actually cast through user-defined conversion routines; besides that it can convert the r value references, enum to integer and so on so which we have.

(Refer Slide Time: 04:59)



static_cast Operator: Built-in Types

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

// Built-in Types
int main() {
    int i = 2;
    double d = 3.7;
    double *pd = &d;

    i = d;                          // implicit -- warning
    i = static_cast<int>(d);        // static_cast -- okay
    i = (int)d;                     // C-style -- okay

    d = i;                          // implicit -- okay
    d = static_cast<double>(i);     // static_cast -- okay
    d = (double)i;                  // C-style -- okay

    i = pd;                         // implicit -- error
    i = static_cast<int>(pd);       // static_cast -- error
    i = (int)pd;                    // C-style -- okay:     RISKY: Should use reinterpret_cast

    return 0;
}
```

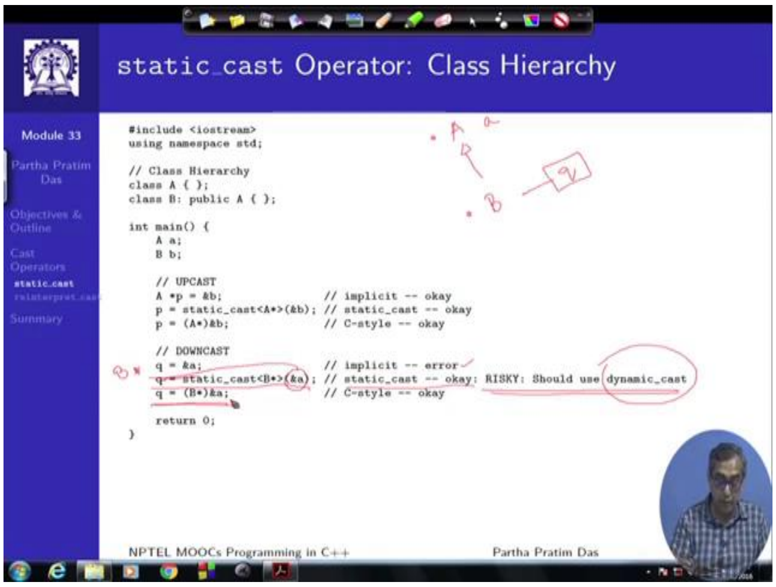
NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let us get started, let us look into some of the simple examples first. So, these are the three variables that we have defined an integer, a double and a pointer to double. And here we are doing a double to integer conversion. Double to integer conversion, this is an implicit one which is which goes through, but it gives you a warning. Why does it give you a warning, because an integer has a smaller representation size than double, so if you convert from double to integer, there is quite possible that you are losing some information, it still allows that, but you give you a warning that is the specification of the language?

But if you do it through a static cast it is ok, it does not give you any warning. Because if you are using static cast then the compiler knows that you are a where you are matured enough and you are aware that if you are taking a double to an integer, you are losing information and you that is exactly what you want to do because you have specified that. Interestingly C style cast also does the same thing, except that you just do not know what was intended here it is clear as to what is intended. The cast others direction integer to double, all things are same except for the implicit case, you do not have a warning because double as a bigger size. So, it any integer can always be represented as a double. So, you do not get the warning.

Look at another case of a trying to convert this pointer to double into integer. So, if you try to do that the implicit cast gives you an error which is what it should, because you have a pointer, what do you mean by putting it to an integer. So, there should not be any conversion, the implicit. Static cast also gives you an error, it says that you cannot do this right, so that is pretty clear because there is no sense of taking a pointer and thinking of it as an as an integer. But if you do C style, it allows that. So, you can see C style cast can allows you almost anything and everything, and therefore, you run a major risk if you use the C style casting. We will see in the next later on that actually in this context, if you really need to do this cast then you should use something like a reinterpret cast and not the C style cast and the static cast of course, will not work in this case. So, this is the simple cases of casting between the built in types.

(Refer Slide Time: 07:25)



static_cast Operator: Class Hierarchy

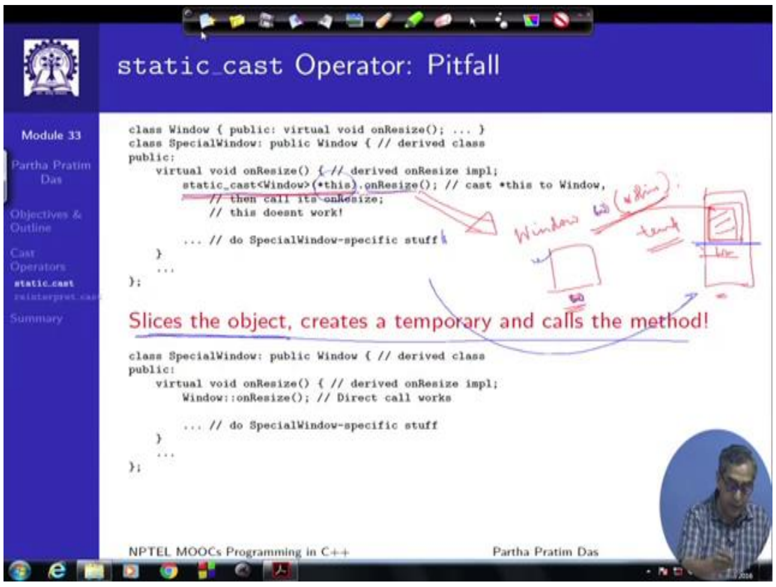
```
#include <iostream>
using namespace std;

// Class Hierarchy
class A { };
class B: public A { };

int main() {
    A a;
    B b;

    // UPCAST
    A *p = &b;                    // implicit -- okay
    p = static_cast<A*>(&b);      // static_cast -- okay
    p = (A*)&b;                   // C-style -- okay

    // DOWNCAST
    q = &a;                       // implicit -- error
    q = static_cast<B*>(&a);      // static_cast -- okay: RISKY: Should use dynamic_cast
    q = (B*)&a;                   // C-style -- okay

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let us move on and try to look at the casting in terms of a class hierarchy. So, I have a put a simple class hierarchy B is A, this is a hierarchy. So, I have two objects a and b. And I can take the address of b and put it to pointer p which is of type A. So, since B is A, so this is where the object b exist and this is where the pointer p exist. So, if I am doing it like this, then I am a doing up-cast. I am going from the specialized here to the generalized here so that is as we saw earlier, this is allowed, and so implicit is ok. I can do the same thing by a static cast, here I take the address of b as an expression and target is s star, I can do that same thing using C style, there is no a no addition, but certainly we will not encourage doing that because again we do not understand what is going on.

Let us try to look at the down cast that is B is A, and I have A, I have an A object. And q I think the declaration for q is missed out, q should be B star. So, I have a q which is of B star, so this is a pointer to B star. So, we are trying to do this. Now naturally as an implicit, this is an error, because this means that I am going from a specialized a generalize to a specialized object. So, there would be certain things missing. So, this is an error. In static cast, interestingly this is ok, why is it, because the compiler reasons that since you have taken a address of A type object and you are trying to put it to a B type pointer you know that you are going down in the hierarchy. So, you are taking responsibility of doing this by saying that you are doing a static cast. So, the compiler will allow that, but again I would strongly, strongly, strongly advice that do not do this, and use what we will discuss as dynamic cast. Of course, C style casting will work in almost every case. So, this is what will happen if you are on a class hierarchy.

(Refer Slide Time: 09:51)



static_cast Operator: Pitfall

```
class Window { public: virtual void onResize(); ... }
class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        static_cast<Window>(*this).onResize(); // cast *this to Window,
            // then call its onResize;
            // this doesnt work!

        ... // do SpecialWindow-specific stuff
    }
    ...
};
```

Slices the object, creates a temporary and calls the method!

```
class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        Window::onResize(); // Direct call works

        ... // do SpecialWindow-specific stuff
    }
    ...
};
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now I will just to show you something simple, if you use this kind of things and kind of pitfalls that you might get into. Think about that you have a class window, and you have a class special window which specializes from window. And let say window has a method on resize, so that basically is the functions if you resize the window this function should be called, and this is a virtual function, so it can be called. Now the special window decides to override the on resize function and implement it again. And in the override, what does it want to do, it first wants to call the function of the base class, so that whatever the resizing operation a general window needs to do that gets done and then it does the special window specific stuff. This is the basic intended design.

Now, in terms of doing that whatever did a programmer do, the programmer wrote the code like this that you are in the special window. So, this pointer points to a special window. So, it took that object cast it to window. So, it is simply cast that object to window. So, which you will say is absolutely fine because this has more information, I am just casting it to this and then you call resize on that. Now the point is this will not work, this will not work. This will lead to what is known as slicing the object, because what happens is when you try to, here what you are allow to do very freely as an up-cast is cast the pointer or cast the reference, but what this person has done is cast the whole object has a whole. Now naturally, you cannot take a special window object, and make it a window object. So, what you will do when you have this kind of a cast call this leads to having a new window object. So, you have some new window object from the current special window object. This is a temporary, so this temporary object gets created through this process. So, what happens here this was your total special window that had a window base and that window base has been copied to w.

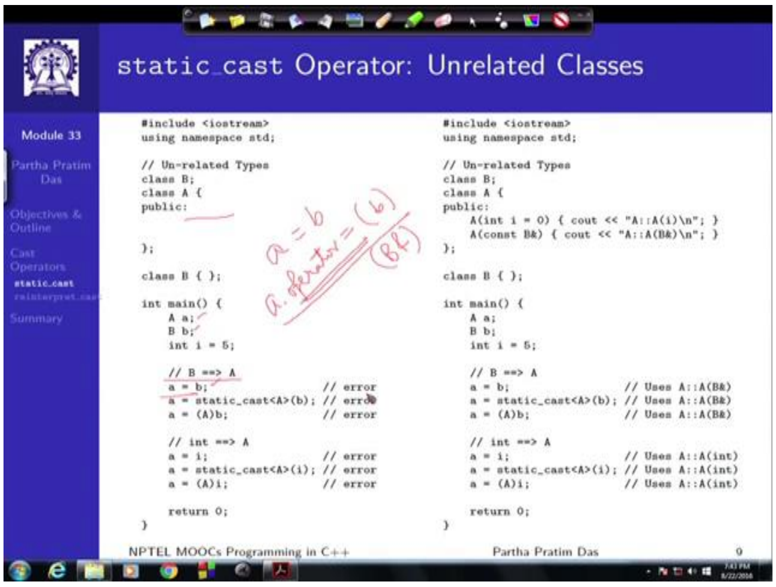
And then what happens, so this is now it has become another w, where it this has got copied because you have tried to do this casting. And then you have call on resize, so that resize gets called on this w, not on the original special window object, because it is temporary has got created. Remember when I discussed about casting in the earlier module, I highlighted that the casting will create temporary objects if the compiler needs to do that, here the compiler needs to do that because it is not a pointer which it can just think is of a different type. But here you are actually asking for a different object, so that object needs to resize. So, it is created an object is created from the base class part of your object. So, all those values are all correct, but what is different is in the process of doing that this has w has become a different object now. And so on the size happens this and then you come back and do the special window specific stuff on the original object.

So, this part is known as slicing. You have sliced the base part out.

We talked about slicing problem in a different context, when we talked about a virtual destructor. The need for virtual destructor, but you can get into slice in through improper casting as well. So, you need to be careful about that, and just you know high highlight the way you could do this is you do not need to do anything sophisticated; all that you need to do is to actually explicitly call the on resize function of the window class. So, within this, you just call this it already has this pointer of the special window. So, if you call this, this function that is this function will get called, and what is this pointer it is using it is using the this pointer of the special window object, because that is the this pointer it has. So, you are now you have.

So, in this process, what will happen, you will not actually cast the object and create a temporary, but you are actually casting that this pointer through an implicit cast which is an implicit up-cast. So, which as we know is valid and you are still referring to the same object. So, this is just to highlight that I mean casting on the face of it looks you know something very straight forward something which is pretty safe and so on, but it could really get you into nasty problems. So, you should be careful about that.

(Refer Slide Time: 14:56)



static_cast Operator: Unrelated Classes

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

Summary

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

// Un-related Types
class B;
class A {
public:


};

class B { };

int main() {
    A a;
    B b;
    int i = 5;

    // B ==> A
    a = b;                      // error
    a = static_cast<A>(b);      // error
    a = (A)b;                   // error

    // int ==> A
    a = i;                      // error
    a = static_cast<A>(i);      // error
    a = (A)i;                   // error

    return 0;
}
```

$a = b$

$a.\text{operator}=(b)$

$(B\&)$

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

// Un-related Types
class B;
class A {
public:
    A(int i = 0) { cout << "A::A(i)\n"; }
    A(const B&) { cout << "A::A(B&)\n"; }
};

class B { };

int main() {
    A a;
    B b;
    int i = 5;

    // B ==> A
    a = b;                      // Uses A::A(B&)
    a = static_cast<A>(b);      // Uses A::A(B&)
    a = (A)b;                   // Uses A::A(B&)

    // int ==> A
    a = i;                      // Uses A::A(int)
    a = static_cast<A>(i);      // Uses A::A(int)
    a = (A)i;                   // Uses A::A(int)

    return 0;
}
```

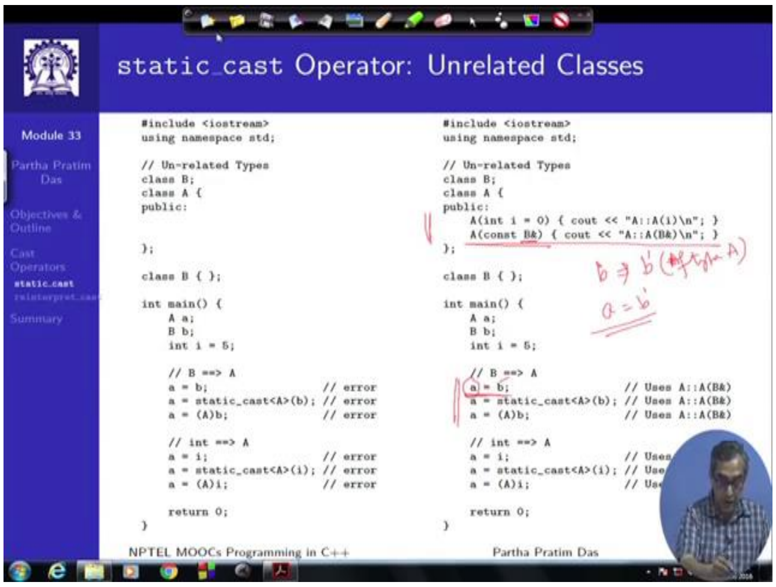
NPTEL MOOCs Programming in C++ Partha Pratim Das 9

So, this is another example of using static cast and the difference is I am now talking about unrelated classes. So, again I have class A and B, but the class A and B are not related, they are not on a hierarchy. And I am trying to convert an object to a b object. So, I this is a object, this is b object, so I am trying to assign b to a. I try that implicit conversion is an error, because if I do a assign b, then as we have understood through operator overloading the compiler looks for a dot operator assignment b, so it expects that this class will have an operator assignment, which takes a B type of object which does not exist. Therefore, the implicit does not work; this is gone. You try static cast you get an error; even you try you get desperate, and even try C style casting, you get an error.

Similarly, say you want to convert from integer to A type, integer is a built in type, you want to convert that to A. So, you try this, try this, try this, all of these are errors. So, what you need to do is here, what you are saying is basically, when you have two unrelated types, and you want to convert one to the other, certainly what you are saying is I have an object or an expression, and from that I am trying I need to create the an object of the target type. So, I have an object of type B, I need to use that to create an object of type A. I have an object of type int, I need to use that to create an object of type A.

If you are on a hierarchy, then you already have the object. So, it is more like you whether you are looking at it as a specialized one or a generalized one is a question, but you have some object to deal with, but here you just do not have the A object, where do you get the i object, so you need to construct that. So, it simply turns out that if you have to do allow this then all that you need to provide is provide a constructor for A, which takes a B object is as simple as that. So, if you provide that then all this three become valid. What it does is you do a assign b, it checks that do I have a way to create an A object using the B object, it finds that there is a constructor, so it does that.

(Refer Slide Time: 17:35)



static_cast Operator: Unrelated Classes

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

Summary

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

// Un-related Types
class B;
class A {
public:


};

class B { };

int main() {
    A a;
    B b;
    int i = 5;

    // B ==> A
    a = b;                  // error
    a = static_cast<A>(b); // error
    a = (A)b;               // error

    // int ==> A
    a = i;                  // error
    a = static_cast<A>(i); // error
    a = (A)i;               // error

    return 0;
}
```

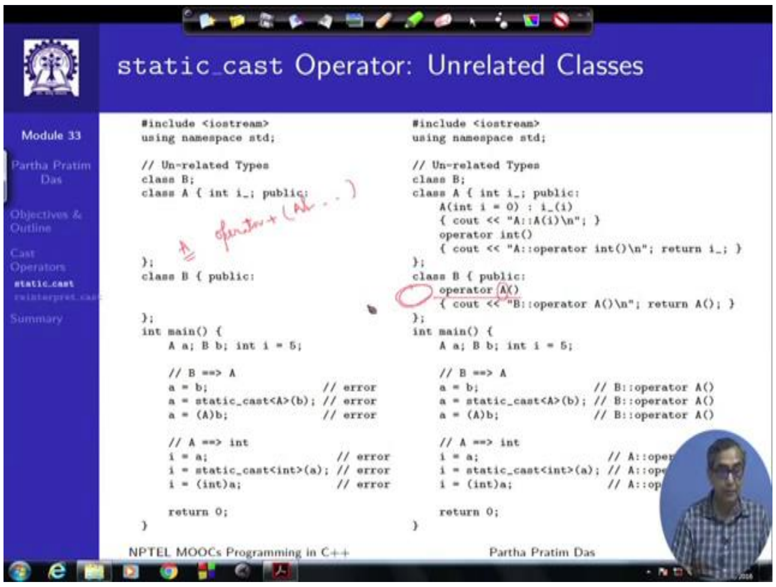
```
#include <iostream>
using namespace std;

// Un-related Types
class B;
class A {
public:
    A(int i = 0) { cout << "A::A(i)\n"; }
    A(const B&) { cout << "A::A(B&)\n"; }
};

class B { };

int main() {
    A a;
    B b;
    int i = 5;

    // B ==> A
    a = b;                  // Uses A::A(B&)
    a = static_cast<A>(b); // Uses A::A(B&)
    a = (A)b;               // Uses A::A(B&)

    // int ==> A
    a = i;                  // Uses
    a = static_cast<A>(i); // Use
    a = (A)i;               // Us

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, it basically creates an it takes the b creates b primed object of type A, and then actually assigns b prime to a using the free copy assignment operator that A class has, A class is not defined any copy assignment operator. So, it makes use of that in the next stage to actually achieve the task. So, it is the similar thing is done if you do static cast, if you use C style casting. If you look into int providing this, then what do you need by the same logic you need a constructor of A which takes int as a parameter, so you add a constructor here. And so this if you do this, it is simply turns out that it takes i, constructs an a object through this, and after construction it simply does a copy assignment through the free copy assignment operation.

So, I have shown here what are the functions that will get called if you just trace the output you will see that the actually those functions are getting called. So, all these three forms of implicit casting, casting with static cast, or C style casting, all of them will give the same behavior. So, this is where we see that if the classes are unrelated, and in several other contexts, we can actually have a user-defined conversion this is what is user-defined conversion taking place. And one way to make the user-defined conversion take place is to actually provide an appropriate constructor for the target type from the source type.

(Refer Slide Time: 19:25)



static_cast Operator: Unrelated Classes

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

// Un-related Types
class B;
class A { int i_; public:

};
class B { public:

};
int main() {
    A a; B b; int i = 5;

    // B ==> A
    a = b;                  // error
    a = static_cast<A>(b); // error
    a = (A)b;               // error

    // A ==> int
    i = a;                    // error
    i = static_cast<int>(a); // error
    i = (int)a;               // error

    return 0;
}
```

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

// Un-related Types
class B;
class A { int i_; public:
    A(int i = 0) : i_(i)
    { cout << "A::A(i)\n"; }
    operator int()
    { cout << "A::operator int()\n"; return i_; }
};
class B { public:
    operator A()
    { cout << "B::operator A()\n"; return A(); }
};
int main() {
    A a; B b; int i = 5;

    // B ==> A
    a = b;                  // B::operator A()
    a = static_cast<A>(b); // B::operator A()
    a = (A)b;               // B::operator A()

    // A ==> int
    i = a;                    // A::oper
    i = static_cast<int>(a); // A::ope
    i = (int)a;               // A::op

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

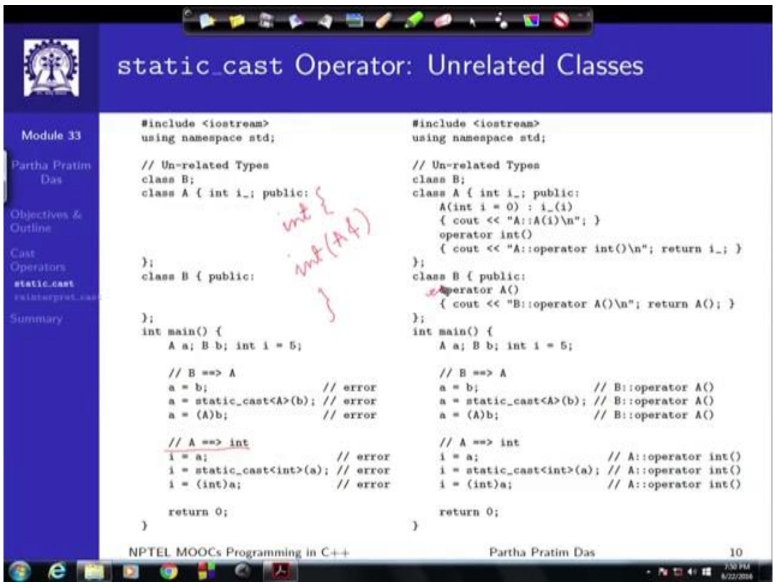
But that the story does not end there, there is something more interesting than this here in the next slide, again those two same classes. Again we are trying to take b object and cast it to an a type object there is nothing different. So, we will turn all of them will be error. But in the second case, I am trying to do something different earlier I try to take an int and construct an a object. Now, given any object I want to make it to an int, I am doing the other way round. Certainly, all of them are errors, because I have no way of knowing that given an a object how should I interpret it as an integer int i. So, all of these are errors.

How do we solve the problem, in case of this, we have seen that we could have used a constructor in A, which takes a B type parameter? And alternate way to do that is to the operator form, conversion operator this is known as a conversion operator. So, class B could write a conversion operator which you write as operator and the target type. So, this is a special operator, which will get invoke, if you are trying to do a casting of a b type object to an a type object. This operator is specifically written for that purpose and this is called the conversion operator. And the conversion operator has a very interesting style if you recall operator overloading like you wrote operator plus, and whatever those types, and then you say that this gives me an A object, this I am taking an A reference and these are return type and so on.

Here you see that in this the name of the operator is operator A, the type name, because it has to change A type. So, this does not take a symbol, it takes a type as the name of the operator, it changes to A. And then it does not have a return type, it does not have a return type here, why does not it have a return type, because you are defining a conversion operator to take a B object and convert it to an A object. So what it must return it must return an A type object, it cannot return anything else. You cannot write that the return type here is int that will be stupid. The return type will have to be A, because it is converting to A. And therefore, it is super flow us to write this, and the syntax does not allow it to be return. So, this is a conversion operator.

So, where I take… so you can do write the whole logic that will you need to construct an A object from the B object, and put it down here. I have done something very ((Refer Time: 22:13)) I have just constructed a default A object and return that, but you could use the data members of B and actually construct this. So, once you provide this, again equivalently all of these become valid, but now there is no constructor in a which takes a b object rather there is a conversion operator in B, for the A types which will get used. What is interesting is how do we solve this problem.

(Refer Slide Time: 22:51)



static_cast Operator: Unrelated Classes

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

Summary

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

// Un-related Types
class B;
class A { int i_; public: {
                            int {
                            int(A&)
                            }
};
class B { public:


};
int main() {
    A a; B b; int i = 5;

    // B ==> A
    a = b;                  // error
    a = static_cast<A>(b); // error
    a = (A)b;               // error

    // A ==> int
    i = a;                    // error
    i = static_cast<int>(a); // error
    i = (int)a;               // error

    return 0;
}
```

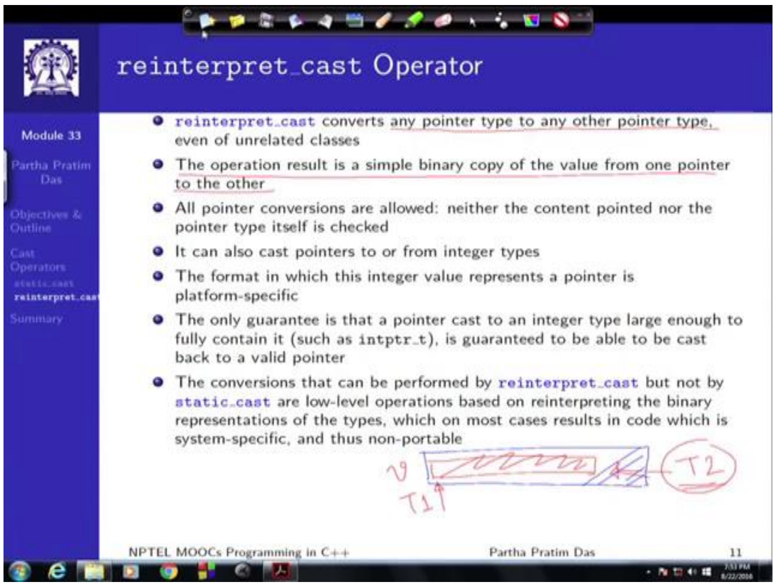
```
#include <iostream>
using namespace std;

// Un-related Types
class B;
class A { int i_; public:
    A(int i = 0) : i_(i)
    { cout << "A::A(i)\n"; }
    operator int()
    { cout << "A::operator int()\n"; return i_; }
};
class B { public:
    operator A()
    { cout << "B::operator A()\n"; return A(); }
};
int main() {
    A a; B b; int i = 5;

    // B ==> A
    a = b;                  // B::operator A()
    a = static_cast<A>(b); // B::operator A()
    a = (A)b;               // B::operator A()

    // A ==> int
    i = a;                    // A::operator int()
    i = static_cast<int>(a); // A::operator int()
    i = (int)a;               // A::operator int()

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das 10

Now think about, how could you have solve this problem using a previous style, one way could have been that if I have my int type, then I could have written a constructor of int which takes A type object that will convert A into int. Given A, it will give me int, exactly the way we did the previous case. Now certainly that option does not exist, because we do not have, we cannot write a constructor for int, it is a built in type. So, the only option remains is take the conversion operator root. So, is the only way you can write it is for class A you write an operator int you are writing this for class A.

So, what it means it means that given a class A object, I can convert it to an int using this operator int function, and that is what I have done, I have taken this, and I have been implemented as if it returns. So, this class has a int data members. So, I have return this data member that is a specific computation choice you can do anything else, but all that you will need to do is the since it is an operator int, it will necessarily always return an int. So, there have to be returns on int which is computed from the A object that you have. So, this is using the operator conversion operator is necessary if you want to a convert some user-defined type into a built in type value. For the other, you could still use the constructor; and for two user-defined types, you could either use the constructor or use the conversion operator, but certainly you cannot use both. So, once you have done this, then all these become valid; and all of them actually use this conversion operator to convert. So, this is a basic a way the static cast operator could allow you to invoke user-defined conversion code and work between unrelated classes.

(Refer Slide Time: 24:47)



reinterpret_cast Operator

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

Summary

- reinterpret_cast converts any pointer type to any other pointer type, even of unrelated classes
- The operation result is a simple binary copy of the value from one pointer to the other
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked
- It can also cast pointers to or from integer types
- The format in which this integer value represents a pointer is platform-specific
- The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as intptr_t), is guaranteed to be able to cast back to a valid pointer
- The conversions that can be performed by reinterpret_cast but not by static_cast are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable

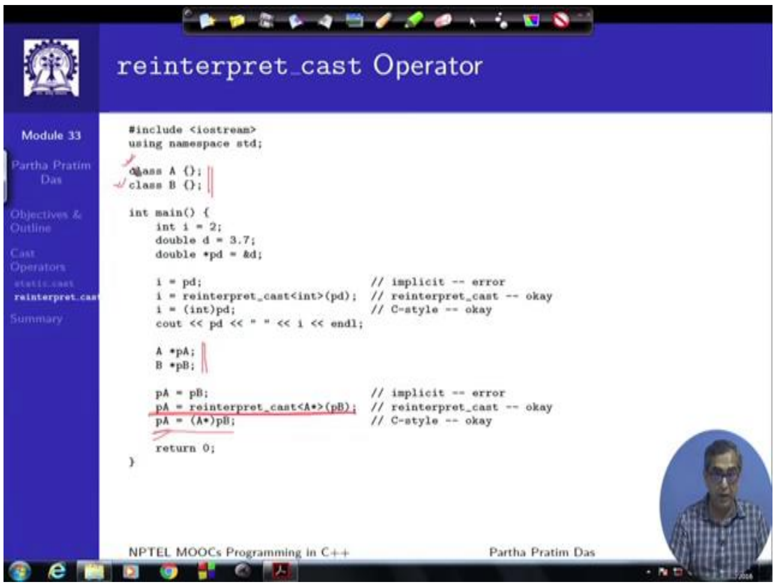
NPTEL MOOCs Programming in C++ Partha Pratim Das 11

The next is what is known as reinterpret cast operator, the third kind of. The reinterpret cast operator can convert pointer of any type to pointer of another type, even of the unrelated classes. This is the most important thing. The reinterpret cast, if you have you is doing reinterpret on something, so you have a variable v, which has some representation some value, if this is of type t 1, and you reinterpret cast to t 2, then you simply look at this as if it is a t 2 object. You do not do any computation, you do not try to do anything, you just take this address and start thinking as if it is a t 2 object that is all. So, that could give you really disastrous result, for example, t 2 could mean a bigger type. So, you are looking at here, and you could be just looking at this much. So, this is something which was not there in the variable v, which will think that is the part of variable v.

So, normally reinterpret cast is a given, so that you can convert between pointer types. As you know whatever is the system, the size of the pointer is same for all types of pointers. So, this will at least not have the size issue, but it can be used also to cast two integers as well as from integer. Now when you cast it that way, you will have to make sure that your integer type is large enough, so that the pointer address can be accommodated there. So, this is a platform specific, next point, you will should this is where platform specific feature, because C language does not guarantee that your integer size and your pointer size is related in any way. In a large number of systems, they are the same size like 32 bit machines, Intel x 86 both are 4 bytes, but it is not guaranteed by the language. So, you should be careful about the specific platform specific stuff.

So, the conversions that can be performed by reinterpret cast, but not by static cast or whole based on low-level operations of the machine. And the general understanding is that if you require reinterpret cast then you must be very, very sure of what you are trying to do. Because under normal circumstances, if you are done a good design in C++, you should not require reinterpret cast. I have read several books on C++, where in the whole 300, 400 page book, there may be only one or two instances of reinterpret cast as code examples and that to just to show how reinterpret cast can be used, so that is the kind of in a module that exist.

(Refer Slide Time: 27:54)



reinterpret_cast Operator

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

Summary

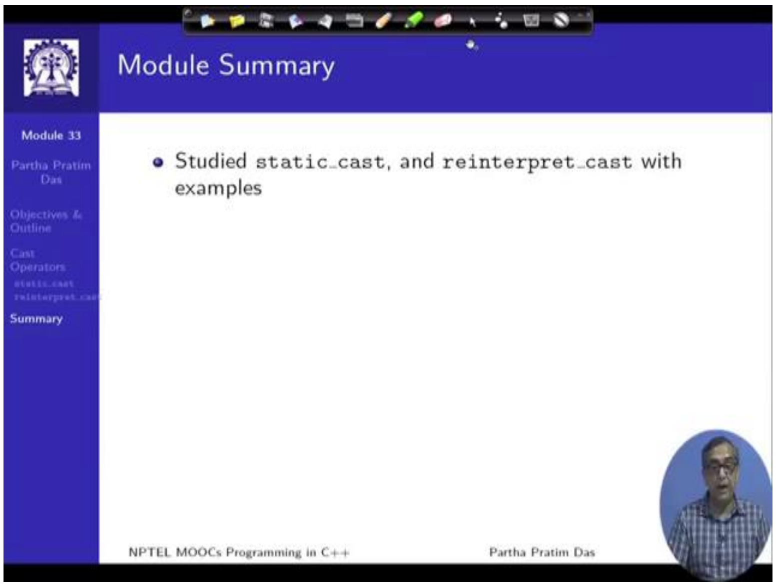
```
#include <iostream>
using namespace std;

class A {};
class B {};

int main() {
    int i = 2;
    double d = 3.7;
    double *pd = &d;

    i = pd;                         // implicit -- error
    i = reinterpret_cast<int>(pd);  // reinterpret_cast -- okay
    i = (int)pd;                    // C-style -- okay
    cout << pd << " " << i << endl;

    A *pA;
    B *pB;

    pA = pB;                        // implicit -- error
    pA = reinterpret_cast<A*>(pB);  // reinterpret_cast -- okay
    pA = (A*)pB;                    // C-style -- okay

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

But certainly having said that reinterpret cast still exists because there are certain C style cast you can do which you cannot do with any of the other three cast operators. Therefore, for example, you can convert a say if you look into this, we were looking at this little earlier also, so if you look into this and this is a pointer to double, and this is an integer this is implicitly an error, but in C style casting, you could still do that. So, reinterpret cast has been given, so that for such cases of C style casting, you have a formal cast operator to do the cast. So, all that you say that take a double pointer as you have here, and just think of it as an integer, do not try to do anything else. Now after that the risk lies with the programmer has to what is gone wrong. It can be used for a completely, these are two completely unrelated classes pointers to do unrelated classes as you can see here. And you can cast one into another through a reinterpret cast. So, this is different; this is not casting the objects which were doing by the static cast, where we could invoke the constructor or the conversion operator.

Here we are just trying to cast the pointer. So, certainly we cannot define any logic for casting the pointer is just the way to infer the type and reinterpret cast allows you to do that. And after this has being done then you can actually dereference p A and start using that objects. Of course, what is in store for you because this was a separate object, this is a separate object. What is in store for you, it is completely unpredictable. And therefore, I would strongly, strongly recommend that do not use reinterpret cast at all. If you are requiring reinterpret cast take a second look there must be some lacunae in the design, so that you are requiring it changing the design, you will find that you will able to manage with the other three types of cast operators.

(Refer Slide Time: 30:04)


Module Summary

Module 33

Partha Pratim Das

Objectives & Outline

Cast Operators

static_cast

reinterpret_cast

Summary

- Studied static_cast, and reinterpret_cast with examples

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, to summarize, we have continued on the discussion of type casting in C++. And specifically, we have studied the static cast and reinterpret cast operators in this module.