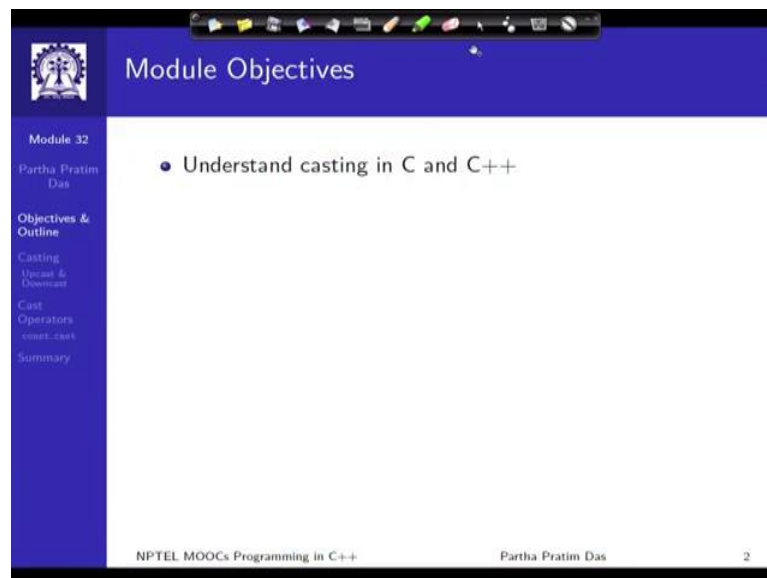


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture - 47
Type Casting and Cast Operators: Part I

Welcome to Module 32 of Programming in C++. In this module and the next couple of ones, we will discuss in depth about type casting and cast operators in C++.

(Refer Slide Time: 00:41)



The image shows a presentation slide with a blue header and a white main content area. The header contains the text 'Module Objectives' and a small logo on the left. The main content area has a single bullet point: '• Understand casting in C and C++'. On the left side, there is a vertical navigation menu with the following items: 'Module 32', 'Partha Pratim Das', 'Objectives & Outline', 'Casting', 'Type Casting & Operators', 'Cast Operators', 'cast.cpp', and 'Summary'. At the bottom of the slide, there is a footer with the text 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and the number '2'.

So, the objective is to understand casting in C and C++.

(Refer Slide Time: 00:48)

The screenshot shows a presentation slide with a blue header and a white main area. The header contains the text 'Module Outline'. The main area contains a bulleted list of topics. On the left side, there is a vertical navigation bar with a blue background and white text. At the bottom of the slide, there is a small circular portrait of a man and some text.

Module 32
Partha Pratim Das

Objectives & Outline

Casting
Upcast & Downcast

Cast Operators
const_cast

Summary

Module Outline

- 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

This will be the overall outline. We have already made some small introduction to casting while we were talking about polymorphism, primarily the C kind of casting. And the fact that on a hierarchy up cast that is casting from a specialized class object to a generalized class type is safe; and downcast is not safe, it can lead to really serious errors we observed all of that.

So, we will take a quick recap on that. And then we will introduce what is known as cast operator in C++, there are four different operators we will discuss that. Ending with the dynamic casting scenarios, where casting is done based on the runtime type; and for that we will discuss another additional type id operator. So, for these modules, we will carry the same outline, thus parts to be covered in the specific module will be shown in blue as we have done here. And for that module the blue part will be visible on the left of your screen in every slide.

(Refer Slide Time: 02:16)

Type Casting

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }
```

// compiler generates:
double f (int i, int j) {
 double temp_i = i, temp_j = j; // Conversion
 return temp_i / temp_j;
}

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, let us step back and ask for type casting, why should we cast. So, casting is primarily required to convert the type of an object type of an expression a function, function argument, a return value etcetera to another type. So, the basic concept of C++ is strongly rooted in type that is any variable, any constant, any object that we use has a well defined type or is expected to have a well-defined type. So, as we involve them in different expressions in different computations, there is a need that often for computing an expression, I need an object of a certain type, but at hand I have an object of a different type, and that is where I need to cast.

As we have seen in the context of C that a large number of type casting is done through implicit conversion or what can be said as silent conversion, these are standard C++ conversion; standard in the sense that conversions that are between known types and therefore, have already been enumerated by the language designer. So, there is a rule which dictates as to how I can implicitly convert an integer to a double or a double to an integer, whether I can convert a pointer to a certain type into a pointer to wide and so on.

And it will also have an implicit conversion, which is add on over C of user-defined conversions see you would recall that C did not exactly have user-defined type. So, the user-defined conversion was not a part of C, and that will come in here, and those can be

used as implicit conversions also.

The word implicit here particularly mean that when we do this kind of a conversion, we actually do not write anything extra in the source code to say that I am doing this kind of a conversion. But from the context of the expression, and the context of the object being used, you figure out that a conversion has happened. Naturally, in contrast, there is a whole lot of conversion, which is known as explicit conversion. Now of course, if we have the convenience of implicit conversion then why do we have explicit conversion? Explicit conversion is one where I explicitly write that I am making a conversion. So, if I x is of certain type T_1 , I would try to write out that I would like to take x to another type T_2 convert it to another type T_2 , and write it out explicitly in the source.

When I do that then I say I am doing explicit conversion and the reason I do explicit conversion could be many for example, it could be that implicit conversion is making a conversion which is not what I want. It is converting it to it into something which I am not comfortable with or it could be that implicit conversion finds that given an object or given an expression there are multiple possible conversions and it becomes ambiguous to which one should be taken.

And in those cases, and which are quite a few, you may need to make use of the explicit conversion. It is always usually, it is always a good practice to make conversions as explicit as possible, so that your intention in writing the code and what the compiler has understood for your intention becomes completely resolved, if you making an explicit conversion. Because you are writing it out in the part of the text if it is implicit then it depends on your understanding and the compilers understanding of that particular situation of conversion, and we will see how that may lead to different difficulties.

(Refer Slide Time: 06:50)

The slide is titled "Type Casting" and is part of "Module 32" by Partha Pratim Das. It contains a list of bullet points explaining why casting is used, including implicit conversions and explicit conversions. A code example shows a function `double f (int i,int j) { return (double) i / j; }` with handwritten annotations: a bracket under `(double) i` labeled "i: int -> double" and a bracket under `j` labeled "int". Below the code, it says "// compiler generates:" followed by `double f (int i, int j) { double temp_i = i, temp_j = j; // Conversion return temp_i / temp_j; }`. A small video inset of the speaker is visible in the bottom right corner.

Module 32
Partha Pratim Das
Objectives & Outline
Casting
Update & Download
Cast
Operators
const-cast
Summary

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }
```

// compiler generates:
double f (int i, int j) {
 double temp_i = i, temp_j = j; // Conversion
 return temp_i / temp_j;
}

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, to perform a conversion, to perform a type cast a compiler may now so far what the kind of things that you have seen, I have just try to illustrate one here that suppose we have written a function like this even in C. So, it is f is a function that takes two integers int i and int j and returns a double value and how does it return a double value by doing this. So, what all you have here is this is a nice illustration of the different conversions this is one where I make an explicit conversion. I say that i is an int, so i is of type int i want to take it to double.

Now what happens to j? j is also of type int. Now I do not say that actually I also need j to be taken to the type double, because I am doing a division where the first argument, the first operand is a double. And therefore, the second operand also needs to be double. So, there will be another conversion here which is implicit. So, this is a simple example where you have implicit as well as explicit conversion.

(Refer Slide Time: 08:21)

The slide is titled "Type Casting" and is part of "Module 32" by "Partha Pratim Das". It contains the following content:

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }
```

// compiler generates:
double f (int i, int j) {
 double temp_i = i, temp_j = j; // Conversion in temporary
 return temp_i / temp_j;
}

Handwritten annotations in red ink show "i = 2;" and "j = 3;" above the function definition. Below the function definition, "2.0" and "3.0" are written under the variables i and j respectively, with lines connecting them to the variables in the code.

And as I have seen with a large number of my students, in students mind the thinking always is that i is an integer possibly its value was 2; j is another integer possibly its value was 3. And when I just convert it to double then I say that it becomes 2.0 or it will become 3.0 and so on and nothing actually happens, but that is not the case.

(Refer Slide Time: 08:40)

The slide is titled "Type Casting" and is part of "Module 32" by "Partha Pratim Das". It contains the following content:

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }
```

// compiler generates:
double f (int i, int j) {
 double temp_i = i, temp_j = j; // Conversion in temporary
 return temp_i / temp_j;
}

Handwritten annotations in red ink show a memory diagram. It consists of two boxes labeled "int" and "double". The "int" box is divided into two smaller boxes, and the "double" box is divided into four smaller boxes. Arrows point from the "int" box to the "double" box, indicating the conversion process.

For the simple reason that thinks about a case of i being integer, it has a certain representation. So, this two has a representation which is an integer representation, which typically is a single field representation where you use some kind of a particular way to show the signed numbers. So, you use a particular style for doing that. And when you take it to double, then you get into a different representation which is a usually multipart, where you have a sign explicitly put in terms of whether it is positive or negative then you keep a certain characteristics of this and you keep a mantises part of it.

(Refer Slide Time: 09:36)

The slide is titled "Type Casting" and contains the following content:

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

Handwritten annotations on the slide include:

- $35.7 \rightarrow 0.357 \times 10^2$
- A diagram showing a box with "+ 0.357" and "2" inside, with an arrow pointing to the result "35.7".

Code examples shown on the slide:

```
double f (int i,int j) { return (double) i / j; }
```

```
// compiler generates:
double f (int i, int j) {
    double temp_i = i, temp_j = j; // Conversion in temporary
    return temp_i / temp_j;
}
```

So, all that you say that if my number is 35.7, then I do not represent it; let me clear that (Refer Time: 09:36) if my number is 35.7, then I do not represent it as 35.7 or anything like that. I represent it in some normalize form, typically what the systems do these days that you convert it to 0.357 into 10 to the power 2, may be you will not do 10 to the power 2, you will possibly do this power in binary. I am just explaining it in decimal. So, the representation then becomes that my number is positive my fractional part is 0.357 that is it is a fraction which is less than 1 and then my power is 2. So, this representation gives me 35.7.

Now, certainly if I want to take it to integer, this will become 35, which is your 10s complement, but or 2s complement representation which is very different. Even if this

were 35.0, this was 0.350, this representation and the representation of 35 is very different. So, what is the consequence of that it is that you cannot just wish that through casting the type has changed, you actually have a different kind of value when you have changed it.

(Refer Slide Time: 11:01)

Type Casting

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }
```

// compiler generates:
double f (int i, int j) {
 double temp_i = i, temp_j = j; // Conversion in temporary
 return temp_i / temp_j;
}

Diagram illustrating memory allocation for casting:
- Variable `i` is shown in a box.
- A new box labeled `double (i)` is shown, with an arrow pointing to it from the `i` box.
- Handwritten notes: `→ Memory` and `→ Computation` are written next to the `double (i)` box.

NPTEL MOOCs Programming in C++ Partha Pratim Das 4

So, if I have `i` represented somewhere, and once I have cast it to double that same location cannot actually carry that values. So, I have a different location some location `t1`, where I have the value of `i` equivalent value of `i` in the double representation. So, in many a times, we will see that compiler actually needs to allocate separate storage for this cast value and then initialize that with the casting process.

So, cast includes in many cases not always though, I will highlight where it will not need this it, but in many cases it needs to allocate memory where the temporary value, the cast value will be put, and it needs to perform computation to take the original value, and compute the value that have cast to. So, that is the basic idea of casting. And we will need to see how that happens well.

(Refer Slide Time: 12:05)

The slide is titled "Type Casting" and is part of "Module 32" by Partha Pratim Das. It contains the following content:

- Why casting?
 - Casts are used to convert the type of an object, expression, function argument, or return value to that of another type
- (Silent) Implicit conversions
 - The standard C++ conversions and user-defined conversions
- Explicit conversions
 - Type is needed for an expression that cannot be obtained through an implicit conversion more than one standard conversion creates an ambiguous situation
- To perform a type cast, the compiler
 - Allocates temporary storage
 - Initializes temporary with value being cast

```
double f (int i,int j) { return (double) i / j; }
```

```
// compiler generates:  
double f (int i, int j) {  
    double temp_i = i, temp_j = j; // Conversion in temporary  
    return temp_i / temp_j;  
}
```

The slide footer includes "NPTEL MOOCs Programming in C++", "Partha Pratim Das", and the number "4".

So, if I have a function like this then in all likelihood, the compiler actually is implemented a function something like this, this remains same. But it needs to do the conversion, so it defines a new variable temp i. And it is doing a conversion here, this is where the conversion is happening, this is like you can think of this as if like a constructor, so as if double as a constructor which is being initialized with an integer value i. So, temp i will become a new double variable having the value which is closest to the integer value i.

Similarly temp j, this will also need conversion will have a double corresponding to j and then you actually divide these two double numbers and give the result in double. So, this conversion, this conversion are what is required and these are the two fields which are required for the temporary storage. So, this is the scenario of casting that will happen in significantly in C, and that is what we keep on seeing when while we either use implicit casting, implicit conversion or when we use C style conversion in the process.

(Refer Slide Time: 13:12)

Casting: C-Style: RECAP (Module 26)

- 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 *p = &d;  
  
d = i; // implicit  
  
i = d; // implicit -- // warning C4244: '=' : conversion from 'double' to 'int',  
// possible loss of data  
i = (int)d; // explicit  
  
i = p; // error C2440: '=' : cannot convert from 'double *' to 'int'  
i = (int)p; // explicit
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, just to quickly recap we had seen a lot of C style conversion in module 26 before discussing getting into the dynamic binding. So, we have seen the usual you know integer double kind of conversion we have seen different other kinds of conversion also like we have seen that implicit conversion is not possible in certain cases like a pointed to double cannot be converted to int, but we explicit c style casting those things can be done.

(Refer Slide Time: 13:44)

Casting: C-Style: RECAP (Module 26)

- (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

We will see what are the consequences of that we saw different now the same rule if it is extended to C++ where classes come into the play, then we have seen that if there are two unrelated classes then mostly, no kind of implicit conversion between the objects are actually possible. But you can still use the C style of post casting to cast between the different pointers of these types.

(Refer Slide Time: 14:14)

Casting: C-Style: RECAP (Module 26)

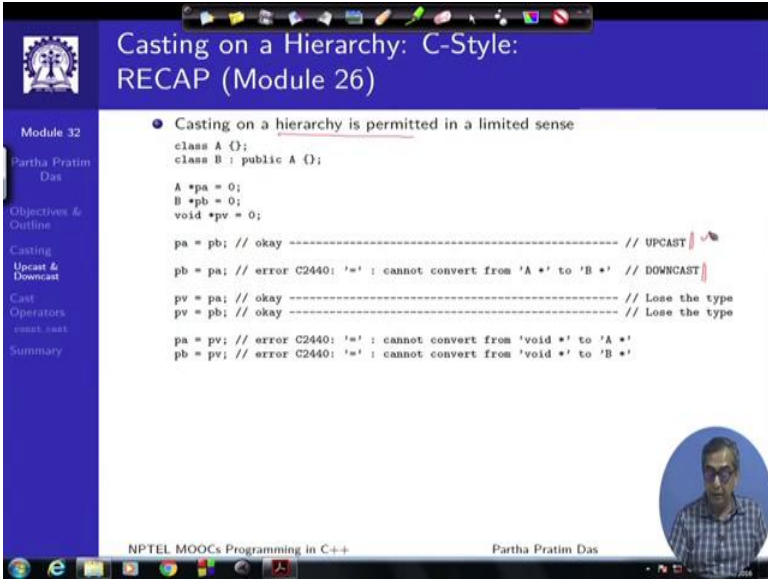
- Forced Casting between unrelated classes is dangerous

```
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 = (A*)&b;  
q = (B*)&a;  
  
cout << p->i << endl; // prints -858993459 ----- GARBAGE  
cout << q->d << endl; // prints -9.25596e+061 ----- GARBAGE
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

We have seen the danger of doing this kind of casting; and because we have seen in this example earlier below that it is possible that I have a forced a casting, which is not to be cast, because it is a two unrelated classes. And the compiler allowed us to do that and then gave out certain garbage output.

(Refer Slide Time: 14:38)



Casting on a Hierarchy: C-Style: RECAP (Module 26)

- Casting on a hierarchy is permitted in a limited sense

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

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, going further on that, we saw the casting still may make some sense, if the classes are on a hierarchy. And in that, we saw that there could be up-cast or down-cast; up-cast is specialization to generalization; downcast is generalization to specialization. And we saw that up-cast is usually a relatively safe thing to do.

(Refer Slide Time: 15:02)

The slide is titled "Casting on a Hierarchy: C-Style: RECAP (Module 26)". It features a blue header with the NPTEL logo and the title. A sidebar on the left lists navigation options: Module 32, Partha Pratim Das, Objectives & Outline, Casting, Upcast & Downcast, Cast Operators, view_text, and Summary. The main content area has a blue background and contains the following C++ code:

```
● 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 3
// cout << pa->dataB_ << endl; // error C2039: 'dataB_' : is not a member of 'A'
```

At the bottom right, there is a small circular portrait of Partha Pratim Das. The footer includes "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, you can up-cast because when you are up-casting, you are using less information than what actually exist. So, we saw that these is up-cast are usually, OK; but if you down-cast then you are going from generalization to specialization, so you actually have less information and you are trying to infer more information and that gets risky.

(Refer Slide Time: 15:26)

The slide is titled "Casting in C and C++". It features a blue header with the NPTEL logo and the title. A sidebar on the left lists navigation options: Module 32, Partha Pratim Das, Objectives & Outline, Casting, Upcast & Downcast, Cast Operators, view_text, and Summary. The main content area has a blue background and contains the following text:

- 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++

At the bottom right, there is a small circular portrait of Partha Pratim Das. The footer includes "NPTEL MOOCs Programming in C++" and "Partha Pratim Das".

So, coming to C++, the casting takes a completely different view significantly different view compared to C. In C, you had implicit cast and C style cast all of which know. And the two major factors is it often loses the type information in the several context, because you can force something into an expression into a certain type without really bothering about what happens to the representation, and the correctness of computation. And the casting really lacks the clarity of semantics. C++ has tried making this far more uniform and far more type consistent, so that your type always remains correct. So, in C++, you can perform fresh inference of types actually often without changing the value, because you often need to infer about type where you really do not want the value to change, you do not need the value to change, and we will show examples of that.

And in other cases, while you infer the type like you were doing in case of inferring an integer as a double; you need to change the value. And this can be done either through implicit computation, where you do not need to write how the change will happen. We have not written how integer is getting change to double or it could be situations where I will need to say the user will need to define how through the casting the change will happen. And with all that, in C++, the casting preserves type information in all context that is a very, very important factor which does not is not guaranteed in C. And it provides a clear semantics which is not also the case in C by making use of certain cast operators. So, there are fore cast operators, which you can explicitly write out to cast one value into another.

And the reason the cast operators are highly encouraged in comparison to implicit casting or explicit C style casting is a fact that if you have cast operators, you can see they have very peculiar kind of name const cast, static cast, reinterpret cast dynamic cast. So, if you have use them then you can easily do a grep kind of operation, grep is basically searching for a string in a file, those you have used Unix know this very well. So, you can use or you can use in your systems search operation, so if you just look for const cast, you will able to see exactly what are the points in your source code where you have used the constant casting mechanism. So, it has a lot of advantages.

And anything that you can cast by C style, you can cast more meaningfully using these operators. And therefore, once you have understood the casting in C++, you should not at

all use the C style casting at all in your code. If you need to use that, you will know that there is something wrong in your design or your understanding of coding.

(Refer Slide Time: 18:27)

Cast Operators

- A cast operator takes an expression of **source type** (implicit from the expression) and converts it to an expression of **target type** (explicit in the operator) following the semantics of the operator
- Use of cast operators increases robustness by generating errors in static or dynamic time

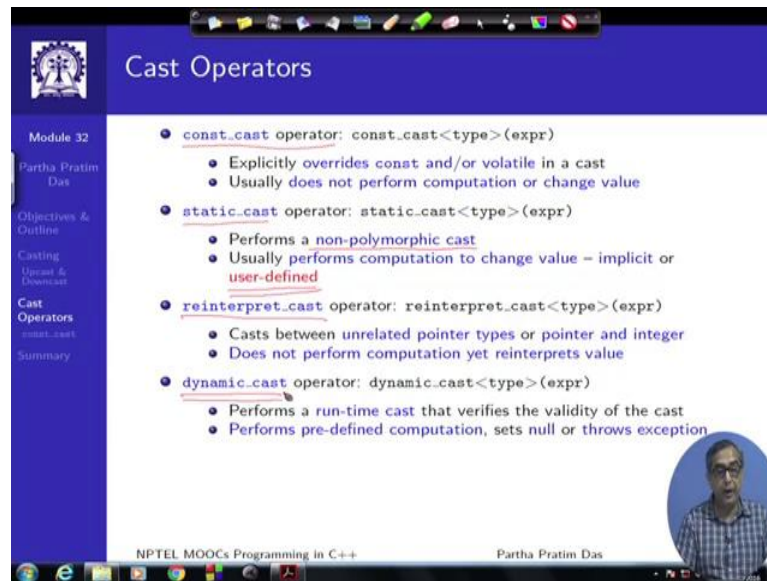
Handwritten diagram:
i : T1 (Source) → Cast < T2 > → T2 (Target)
C (circled) → T2

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now, usually a casting is as I said that a variable is of certain type T 1 and I want to take it to some type T 2. So, the way it works is a cast operator takes. So, this is what we say is a source type and this is what we say is a target type. So, a cast operator takes the expression and specifies what is the target type, and that is how this conversion will happen, now it does not need to specify the source type because in C++ all expressions, all variables have a fixed type at any point they have a type.

So, knowing i, I will know the source type T 1 because i is of type T 1, but I need to know - what is a destination type, what is a target type and that will specify like this. And C++ has operators which can do casting at the static time or at the dynamic time.

(Refer Slide Time: 19:29)



The slide is titled "Cast Operators" and is part of "Module 32" by Partha Pratim Das. It lists the following operators and their characteristics:

- const_cast operator:** `const_cast<type>(expr)`
 - Explicitly overrides `const` and/or `volatile` in a cast
 - Usually does not perform computation or change value
- static_cast operator:** `static_cast<type>(expr)`
 - Performs a non-polymorphic cast
 - Usually performs computation to change value – implicit or user-defined
- reinterpret_cast operator:** `reinterpret_cast<type>(expr)`
 - Casts between unrelated pointer types or pointer and integer
 - Does not perform computation yet reinterprets value
- dynamic_cast operator:** `dynamic_cast<type>(expr)`
 - Performs a run-time cast that verifies the validity of the cast
 - Performs pre-defined computation, sets null or throws exception

At the bottom of the slide, it says "NPTEL MOOCs Programming in C++" and "Partha Pratim Das". There is a small circular video inset of the presenter in the bottom right corner.

So, therefore, cast operators which will discuss one after another. One is a first is a `const` cast operator; this operator is used to basically deal with overwrite the `const`-ness or volatility that is the `c-v` qualification of an expression. So, it can convert take away the `const`-ness from a constant expression, it can add `const`-ness to a non-constant expression and so on. `Static cast` is done primarily through polymorphic is a polymorphic cast. So, if you are not on a polymorphic hierarchy then you can use `static cast`; and it often engages user defined casting, we will see that.

`Reinterpret cast` is a something between unrelated pointer type or between pointer and integer you can to `reinterpret cast` which is very, very risky; it is pretty much like the `c` style cast and should be very sparingly used. And the most important of the cast is the `dynamic cast`, where you do the casting at the runtime based on the runtime time. So, we will see through all of these starting from the first one.

(Refer Slide Time: 20:37)

The screenshot shows a presentation slide with a blue header and a white main area. The header contains the text 'const_cast Operator' and a small logo on the left. The main area contains a bulleted list of three points: 'const_cast converts between types with different cv-qualification', 'Only const_cast may be used to cast away (remove) const-ness or volatility', and 'Usually does not perform computation or change value'. Below the list, there is handwritten text 'const int i' and a diagram of a variable 'i' in a box with the number '5' inside. A small circular inset photo of the presenter is in the bottom right corner. The slide is part of a module titled 'Module 32' by 'Partha Pratim Das'.

Module 32
Partha Pratim Das
Objectives & Outline
Casting
History & Download
Cast Operators
const_cast
Summary

- `const_cast` converts between types with different cv-qualification
- Only `const_cast` may be used to cast away (remove) const-ness or volatility
- Usually does not perform computation or change value

const int i

NPTEL MOOCs Programming in C++ Partha Pratim Das

First is a const cast which is const cast converts between types of different c-v qualification c-v, you remember that is c stands for const v stands for volatile. So, if I have a variable declaration, I can qualify that by saying that it is either a constant that it will never change after it has been constructed or it could be volatile that is it can change at any time without our knowledge. So, whenever we have that we have the c-v qualification and const cast can change that is only const which can take away or remove the const-ness or volatility of an expression.

And usually does not perform any computation or change the value, because const-ness is more not in the value const-ness are in our understanding of the value. See if I have a variable `i`, and I know it is `int` now the variable at present the variable `i` at present may have value 5, now whether or not this value 5 can be changed in future is not a property of the value five. But it is understanding of or interpretation of this variable `i` as to whether it is const; if it is const then you are saying that this cannot be changed; if it is not const this can be changed.

(Refer Slide Time: 22:06)

The slide is titled "const_cast Operator" and is part of Module 32. It lists three bullet points: "const_cast converts between types with different cv-qualification", "Only const_cast may be used to cast away (remove) const-ness or volatility", and "Usually does not perform computation or change value". Handwritten notes in red ink show: "int i; const int & r = i; r = 5; i = 6;". A small video inset shows the presenter, Partha Pratim Das.

- `const_cast` converts between types with different cv-qualification
- Only `const_cast` may be used to cast away (remove) const-ness or volatility
- Usually does not perform computation or change value

Handwritten notes: `int i;`
`const int & r = i;`
`r = 5;`
`i = 6;`

And we have seen earlier also while discussing const-ness as a property that I may have a non-constant variable say `i` and I may have a constant reference to this variable. It is possible that I have a constant reference to this variable, which will mean that I cannot make assignments to this reference because this is a constant one while I can actually make changes to the variable itself. So, it can be looked at a multiple different ways.

(Refer Slide Time: 23:36)

The slide is titled "const_cast Operator" and shows C++ code with several error messages. The code includes `<iostream>` and `using namespace std;`. It defines a class `A` with a constructor `A(int i) : i_(i) {}`, a `get()` method returning `i_`, and a `set(int j)` method. In `main()`, it declares `const char * c = "sample text";` and attempts to call `print(c)`, `print(const_cast<char*>(c))`, `a.set(5)`, `const_cast<A*>(a).set(5)`, and `const_cast<A>(a).set(5)`. Error messages indicate that `print(char*)` cannot convert `const char*` to `char*`, `A::set(int)` cannot convert a `const A*` to `A*`, and `const_cast` cannot convert `const A*` to `A*`. A small video inset shows the presenter, Partha Pratim Das.

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

class A { int i_;
public: A(int i) : i_(i) {}
       int get() const { return i_; }
       void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
  const char * c = "sample text";
  print(c); // error: 'void print(char *)': cannot convert argument 1
           // from 'const char *' to 'char *'

  print(const_cast<char*>(c));

  const A a(1);
  a.get();

  // a.set(5); // error: 'void A::set(int)': cannot convert
             // 'this' pointer from 'const A*' to 'A*'

  const_cast<A*>(a).set(5);

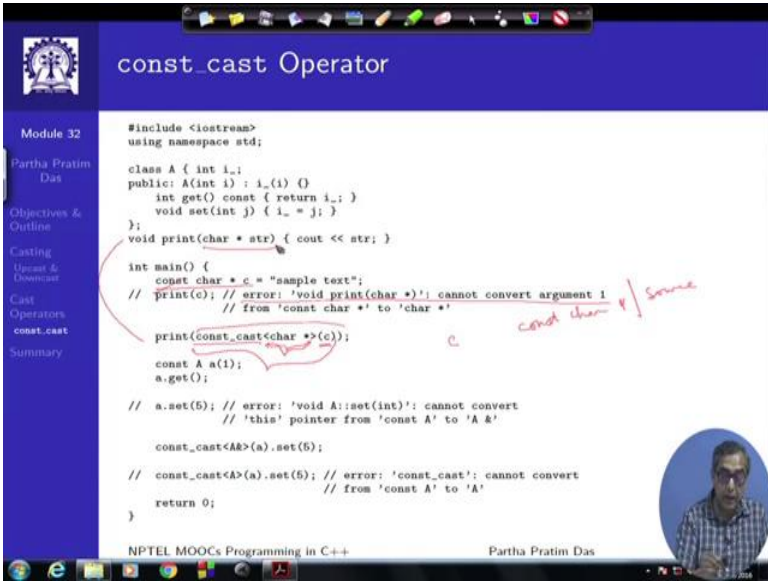
  // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
                             // from 'const A*' to 'A*'

  return 0;
}
```

Now, let us look at how does const cast work. So, I am taking a few simple situations for example, the first situation is having a print function. I mean do not worry about the functionality, it basically takes a char star pointer and prints that. And in the application, I have a char star pointer which is a const pointer because this is constant string; sample text is a constant string.

So, if I tried to do print c, you will get an error. Why will you get an error, because this c is a constant is a pointer to the constant data, you know the point constant is on this side, so the data cannot change, but the actual parameter the formal parameter in print is a non constant pointed to a non constant data, so it can change. So, this sees that if I pass c to this function then c might get changed; therefore, this call will not be allowed. So, you will get this is one kind of possible error that you can get.

(Refer Slide Time: 23:34)



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

class A { int i_;
public: A(int i) : i_(i) {}
    int get() const { return i_; }
    void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1
    // from 'const char*' to 'char*'
    print(const_cast<char*>(c));
    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert
    // 'this' pointer from 'const A' to 'A&'

    const_cast<A&>(a).set(5);

    // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
    // from 'const A' to 'A'

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

Now if you want to actually really want to call this function, then you need to strip c of its const-ness. So, you can do it in this way; const cast is a name of the cast, c is the expression that you want to cast and this is your target type, your target type is char star. So, c was of type const char star, this was your source type. And now you are made it char star, so the type of this whole expression const cast char star c is char star is not const char star. So, it takes the const char star type stripes of the const and gives up a

char star. Now once it becomes char star then it is same as the char star type you have. So, now, you can call this function.

(Refer Slide Time: 24:35)

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

class A { int i_;
public: A(int i) : i_(i) {}
       int get() const { return i_; }
       void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1
    // from 'const char*' to 'char*'

    print(const_cast<char*>(c));

    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert
    // 'this' pointer from 'const A' to 'A&'

    const_cast<A&>(a).set(5);

    // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
    // from 'const A' to 'A'

    return 0;
}
```

Handwritten annotations in red ink:

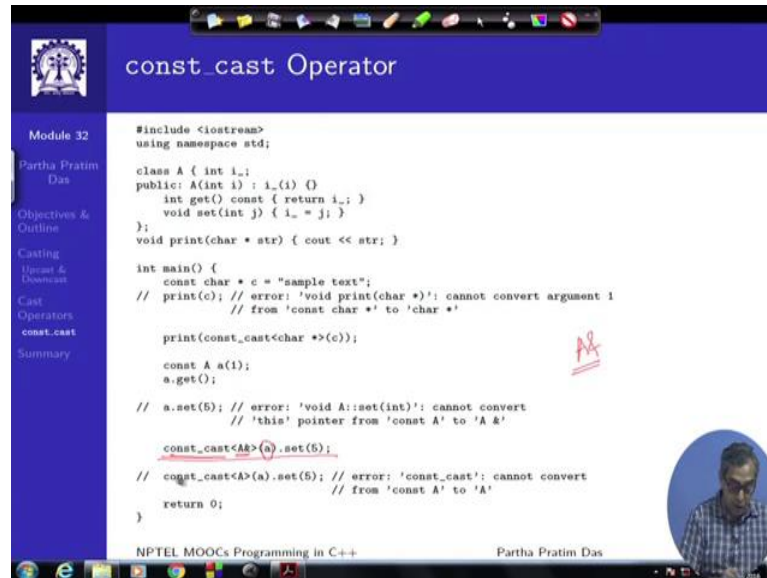
- A * const* (with an arrow pointing to the `const_cast<char*>` cast)
- const A * const* (with an arrow pointing to the `const_cast<A&>` cast)

With this trick you can call the function in this particular case, and because you are using const cast anybody who reads to this code will immediately understand that you needed to strip the const-ness from c, and therefore, you have used it. Thing about another situations suppose you have a class a, and that has a const member function and a non-const member const member function cannot change the contents of the class, non-const member function can. So, if you have a constant object a then calling a dot get is fine because a dot get is a const member function it can be called with const or non const object, because it guarantees that it will not change.

Now think about calling a dot set a dot set this is a non constant member function it actually changes the object, and you are said that a is a constant object. So, a dot set is an error, because you are you cannot be allowed to change the value of a constant object which you can if you are using a non-constant member function. In terms of type basically this pointer of a, has a type const, it has a type which is of this. So, it says that this point is a constant pointer it points to a constant a object. And therefore, but set this function is a non-constant one. So, it needs a pointer of type A star const, but you have a

pointer of this type. So, you cannot pass it there, because if you can pass it there, then it can violate change anything and go ahead.

(Refer Slide Time: 26:34)



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

class A { int i_;
public: A(int i) : i_(i) {}
       int get() const { return i_; }
       void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";
    // print(c); // error: 'void print(char *)': cannot convert argument 1
    // from 'const char *' to 'char *'

    print(const_cast<char *>(c));

    const A a(1);
    a.get();

    // a.set(5); // error: 'void A::set(int)': cannot convert
    // 'this' pointer from 'const A' to 'A &'

    const_cast<A>(a).set(5);

    // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
    // from 'const A' to 'A'

    return 0;
}
```

NPTEL MOOCs Programming in C++ Partha Pratim Das

So, if you still need to call that function, what you can do we can strip the const-ness of the object from A and cast it, you do const cast and now you make it A ampersand that is a taking a reference you are creating a reference of A which is a constant object. And you are creating a non constant reference to that.

So, this resultant expression is an object which has this pointer which does not point to a constant object it points to a non-constant object now and therefore, you can it is to call the set function on this. But certainly you cannot do a similar thing using, you cannot strip the const-ness of the whole object altogether that is not allowed because that will actually mean that you need to construct a new object and do something else. So, this will still continued to be an error, but you can create a non-constant reference to the same object by doing the const cast.

(Refer Slide Time: 27:45)

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

class A { int i_;
public: A(int i) : i_(i) {}
       int get() const { return i_; }
       void set(int j) { i_ = j; }
};

void print(char * str) { cout << str; }

int main() {
    const char * c = "sample text";

    // print(const_cast<char *>(c));
    print((char *)c); // C-Style Cast

    const A a(1);

    // const_cast<A*>(a).set(5);
    ((A*)a).set(5); // C-Style Cast

    // const_cast<A>(a).set(5); // error: 'const_cast': cannot convert
    ((A)a).set(5); // from 'const A' to 'A' // C-Style Cast

    return 0;
}
```

So, this is the basic reason you need const cast for I have put in several other examples for you to practice. For example, this one show that how does it looks like if you do a C style. For example, here we had shown that you can strip const-ness of the string and make the function call go through.

Alternatively you can use C style and do this, I will strongly advice against doing this, because if you do this then anybody reading this will understand that you are striping const-ness of. If you write this then somebody reading this does not know why you are doing this C could have been you are doing this possibly because you know want to remove const-ness, but you are doing this may be because you has a void star pointer. So, you just want to make it understand it as a char star and so on. So, C style of casting does not give you any information and should be avoided.

Similarly, you could do this by creating a non-const reference to the constant object, and make this call we just saw that. You could also do this by a C style casting of this. You can cast the object to a non-const reference and then use that. I will again strongly advice against using this, because here it is clear that you are just removing the const-ness, here it is not clear as to what we are trying to do, it is not possible to make out, it will have to really understand that.

And the most dangerous thing is even if you tried to const cast, the whole object cannot be const cast. We cannot take a constant object and const cast to a non-constant object, because that will mean a completely different object. So, this even with const cast is an error, but surprisingly with c style this is permitted. So, you are actually doing something which is kind of illegal and goes against the basic premise of const-ness that you have built up. So, these should be strongly discouraged. So, please do not use this kind of things and all.

(Refer Slide Time: 29:45)

The screenshot shows a presentation slide titled "const_cast Operator". On the left is a navigation menu with items like "Module 32", "Objectives & Outline", "Casting", "Issues & Demos", "Cast Operators", "const_cast", and "Summary". The main content area displays C++ code and its output. The code includes a constant member function, a non-constant member function, and a constant pointer. Handwritten red annotations highlight "const" and "non-const" in the code. The output shows the values of variables and the result of the const_cast operation.

```

#include <iostream>
using namespace std;
struct type { type() :i(3) {}
void m1(int v) const {
//this->i = v; // error C3490: 'i' cannot be modified because
// it is being accessed through a const object
const_cast<type*>(this)->i = v; // OK as long as the type object isn't const
}
int i;
};
int main() {
int i = 3; // i is not declared const
const int& cref_i = i;
const_cast<int&>(cref_i) = 4; // OK: modifies i
cout << "i = " << i << '\n';

type t; // note, if this is const type t;, then t.m1(4); is undefined behavior
t.m1(4);
cout << "type::i = " << t.i << '\n';

const int j = 3; // j is declared const
int* pj = const_cast<int*>(j);
*pj = 4; // undefined behavior! Value of j and *pj may differ
cout << j << " " << *pj << endl;

void (type::*mfp)(int) const = &type::m1; // pointer to member function
//const_cast<void(type::*)(int)>(mfp); // error C2440: 'const_cast' : cannot convert
// from 'void (__thiscall type::*)(int) const' to 'void (__thiscall type::*)(int)'
// const_cast does not work on function pointers
return 0;
}

```

Output:

```

i = 4
type::i = 4
3 4

```

Finally, I have given some I say set of other examples here, showing you that if you have a constant member function then you can still make changes within that constant member function by changing the const-ness of this pointer within that member function. You can still do some tricks like that. Of course, do not try this with c style casting, because it will become very dangerous.

And just read through this line by line later on. I will just point out to one small piece of the code here. Here, I have defined a constant integer and initialize it with 3. I have defined integer pointer, but what I have done is have taken the address of this variable, which will be a pointer to a constant integer, because it is a constant integer. And therefore, I have striped the const-ness by the const cast.

So, I have j here which is constant and I have p j here which points which is non-constant. So, this const-ness has allowed me to create this. And since this is non-constant, I can assign through this because is pointing to a non-constant integer. So, I can do start p j and assign a value. If you do this after all this you accept undefined behavior you do not know what is going to happen. For example, all that we know that pointer points to a variable then if I print the value of the variable and if I deference and print the value from the pointer, I should get the same value.

So, if you look at this carefully if I print j and star p j, j is a variable p j is the pointer to this variable, star p j is certainly has to be same variable value. If I print that this is output that gets generated that this is 3 and this is 4. So, you are infer a quite a bit of surprise, if you use the const-ness to strip of the const cast to strip of const-ness arbitrarily. The reason this happens is a pretty simple that the compiler particular compiler that have used knowing that j is a const and three has actually replace 3 in this place. So, it does not j anymore, because it knows that it is const, so it cannot change and this one is a changing value.

(Refer Slide Time: 32:24)

The slide displays the following C++ code and its output:

```

#include <iostream>
using namespace std;
struct type { type() :i(3) {}
void m(int v) const {
//this->i = v; // error C3490: 'i' cannot be modified because
// it is being accessed through a const object
const_cast<type*>(this)->i = v; // OK as long as the type object isn't const
}
int i;
};
int main() {
int i = 3; // i is not declared const
const int& cref_i = i;
const_cast<int&>(cref_i) = 4; // OK: modifies i
cout << "i = " << i << '\n';

type t; // note, if this is const type t, then t.m(4); is undefined behavior
t.m(4);
cout << "type::i = " << t.i << '\n';

const int j = 3; // j is declared const
int* pj = const_cast<int*>(&j);
*pj = 4; // undefined behavior! Value of j and *pj may differ
cout << "j = " << j << " *pj = " << *pj << endl;

void (type::*mf)(int) const = &type::m; // pointer to member function
//const_cast<void*(type::*)(int)>(mf); // error C2440: 'const_cast': cannot convert
// from 'void (__thiscall type::*)(int) const' to 'void (__thiscall type::*)(int)'
// const_cast does not work on function pointers
return 0;
}

```

Output:

```

i = 4
type::i = 4
3 4
j = 3 *pj = 4

```

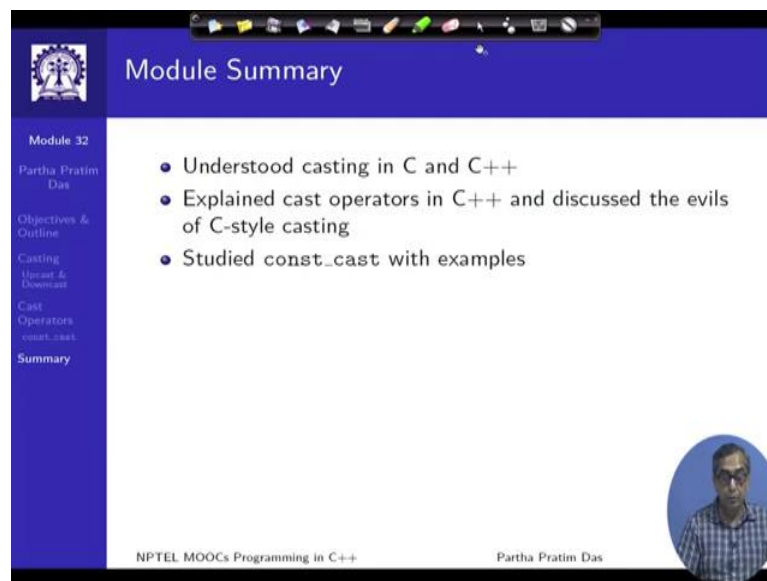
Handwritten red annotations on the slide include:

- A box around the line `*pj = 4;` with the text "undefined behavior! Value of j and *pj may differ".
- Arrows pointing from the `j` variable to the `*pj` expression, indicating that the compiler replaced `j` with its value (3) even though `*pj` is a pointer to `j`.

So, when it had to do this conversion it has silently created another location, so p j actually. So, j is here which has three when this conversion was done p j actually does

not point here, but it points to a new temporary location whose value is initialized with the value of three and then the changes happened within that. You can clearly see that they give you different results. So, go through this, you will get further details, you can learn that function pointers const cast cannot be removed from the function pointers and so on.

(Refer Slide Time: 33:00)



Module Summary

- Understood casting in C and C++
- Explained cast operators in C++ and discussed the evils of C-style casting
- Studied `const_cast` with examples

NPTEL MOOCs Programming in C++ Partha Pratim Das

To summarize, we have tried to understand the basic process of casting in C and C++. And particularly explained cast operators, a basic structure of the cast operator and discuss the evils of C-style casting in a number of examples. And particularly, we have taken a look into the const cast operator.

In the next module, we will take up the other cast operators and move on.