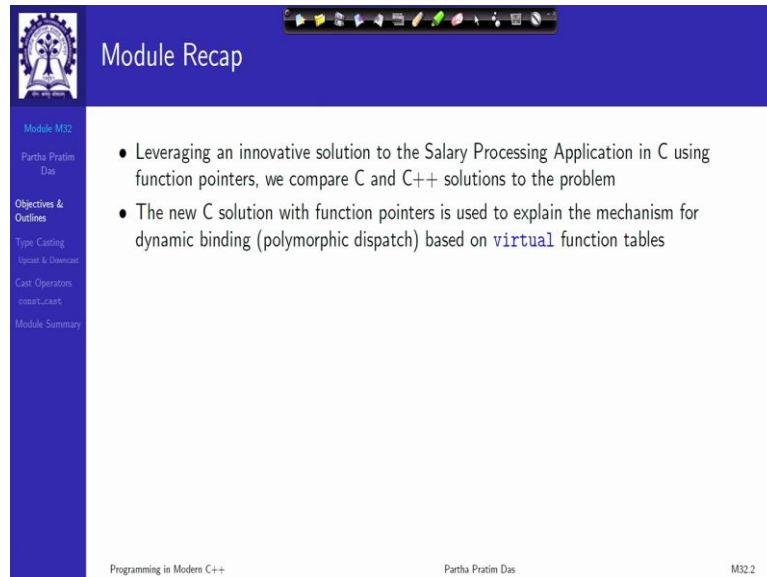


**Programming in Modern C++**  
**Professor Partha Pratim Das**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture 32**  
**Type Casting & Cast Operators: Part 1**

(Refer Slide Time: 0:34)



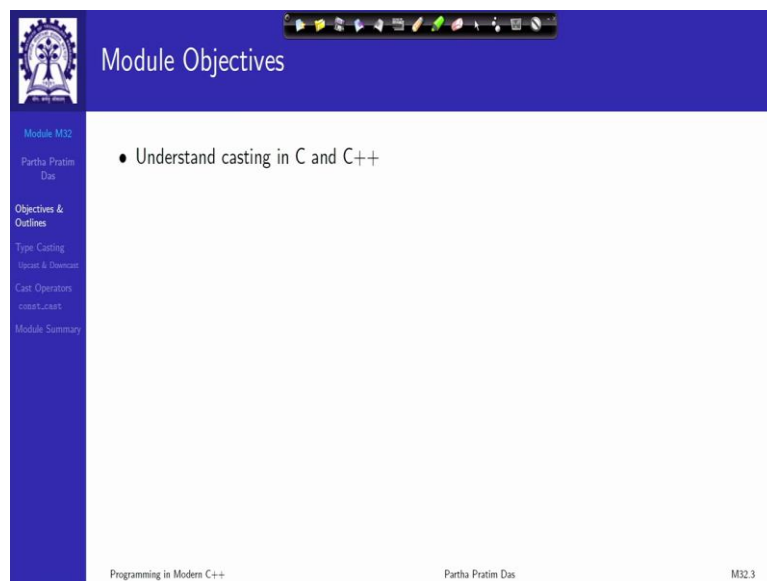
The slide is titled "Module Recap" and features a blue header with the IIT Kharagpur logo on the left. A vertical sidebar on the left contains navigation links: "Module M32", "Partha Pratim Das", "Objectives & Outlines", "Type Casting", "Upcast & Downcast", "Cast Operators", "const\_cast", and "Module Summary". The main content area has a white background with two bullet points:

- Leveraging an innovative solution to the Salary Processing Application in C using function pointers, we compare C and C++ solutions to the problem
- The new C solution with function pointers is used to explain the mechanism for dynamic binding (polymorphic dispatch) based on `virtual` function tables

At the bottom, the footer includes "Programming in Modern C++", "Partha Pratim Das", and "M32.2".

Welcome to Programming in Modern C++. We are in Week 7. And I am going to discuss Module 32. In the last module, we have talked about virtual functions and virtual function table that is an implementation of virtual functions as table of function pointers and how does it parallel with a nearest C application.

(Refer Slide Time: 0:52)



The slide is titled "Module Objectives" and features a blue header with the IIT Kharagpur logo on the left. A vertical sidebar on the left contains navigation links: "Module M32", "Partha Pratim Das", "Objectives & Outlines", "Type Casting", "Upcast & Downcast", "Cast Operators", "const\_cast", and "Module Summary". The main content area has a white background with one bullet point:

- Understand casting in C and C++

At the bottom, the footer includes "Programming in Modern C++", "Partha Pratim Das", and "M32.3".

Now, in this module we will go back to discussing the typecasting again. In fact, this will be a 3-part discussion covering this module and next two modules. You recall that in the module 26, we talked about casting and particularly in the context of C. Now, we will try to look at how does casting behave in C++? What is a very very significant difference between typecasting in C and in C++ and how to do it rightly in C++?

(Refer Slide Time: 1:29)

The slide is titled "Module Outline" and features a blue header with a logo on the left and a video feed of the speaker on the right. A navigation bar at the top contains various icons. The main content area lists three items:

- 1 Type Casting
  - Upcast & Downcast
- 2 Cast Operators
  - `const_cast`
- 3 Module Summary

At the bottom, the text "Programming in Modern C++" is on the left, "Partha Pratim Das" is in the center, and "M32.4" is on the right.

The slide is titled "Type Casting" and features a blue header with a logo on the left and a video feed of the speaker on the right. A navigation bar at the top contains various icons. The main content area has the text "Type Casting" written in red.

At the bottom, the text "Programming in Modern C++" is on the left, "Partha Pratim Das" is in the center, and "M32.5" is on the right.

**Type Casting**

- Why type casting?
  - Type 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
  - Often the type needed for an expression that cannot be obtained through an implicit conversion. There may be more than one standard conversion that may create an ambiguous situation or there may be disallowed conversion. We need explicit conversion in such cases
- 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; // Explicit conversion by (double) in temporary
  double temp_j = j; // Implicit conversion in temporary to support mixed mode
  return temp_i / temp_j;
}
  
```

*Handwritten notes on the slide:*  
 - "conversion" with an arrow pointing to the cast in the code.  
 - "mixed mode" with an arrow pointing to the division operation.  
 - "double" and "int" with arrows pointing to the variables in the code.

Programming in Modern C++ | Partha Pratim Das | M32.7

Okay, so this is the sample outline. So, I will start naturally with a quick recap of what we have learned in the earlier modules about casting. So, the first thing we learned is why do we have typecasting? Typecasting is used to convert the type of an object or expression or function argument or return value to another type. And we have seen that we often need this.

We have also seen that compilers often do this silently either because of the way we have written the code or because it needs to put things together which are not of the same type but can be made to be of the same type. And if it does that silently, then it is called the implicit conversion. So, that includes standard C++ conversions. Earlier we talked about C conversion. So, now we are talking about C++. So, we will say C++ conversions. And what will additionally coming which was not there is user-defined conversion.

So, we do not still know what user-defined conversions are. We will come to that. The other that we have seen is explicit conversion where; why do we need explicit conversion? Because often the type needed for an expression cannot be obtained from the given type through an implicit conversion. Why can might that happen? That might happen due to multiple reasons. One it might happen, because suppose, to do this conversion, I need more than one, a chain of implicit conversion.

Now, or there may be two ways to do conversion. Now, if those kinds of things exist, then the compiler feels that it is getting confused. Because suddenly the compiler does not want to assume something which the programmer did not really mean. So, it is much better to refuse to compile that than to compile something which has you know, unpredictable behaviour.

So, the compiler will refuse to do implicit conversion and you will need to explicitly say that I want this conversion to be done. Only then the compiler will do that conversion. And that is the basic notion of explicit conversion. In fact, explicit conversion maybe, maybe put in place in cases where it is not only that standard conversion is not being not unique but it may also happen that it does not exist. It is disallowed but you still want to do it. You will still want to take a pointer and treat it as an integer or treat it as a long. So, that is the reason we need explicit conversion there.

Now, the question is what actually happens in the conversion? Now, when you say conversion, we have kind of a mental set that well, a conversion is like, I have an integer value 2. I convert it to double. It becomes 2.0. I have an integer value 2.0. I am sorry double value 2.0. I treat it as integer, it becomes 2.

It feels like that as if conversion is kind of rewriting. But often that may not be the case. That may be either case. But often that may not be the case. The conversion or casting may involve actually a lot of computation. It may involve compiler generating specific separate code to do this conversion.

I illustrate here with a very simple example that I started the discussion with that is mixed more division. So, here is a function `f()` which takes two integer variables `i` and `j`. What it does? It casts `i` to double (into double). This is an explicit conversion. Does not do anything with `j`. Now, how will the compiler deal with it?

So, the two things. One is there is an explicit cast. Now, obviously the variable `i` which is storing the value of `i` is of type integer. It has certain size and certain format to store integer. The value that results by converting it or casting it to double cannot be stored in that location. It does not fit. Does not follow the format. Is not semantically consistent.

So, what the compiler has to do? Compiler has to define a new temporary variable. So, say I mean this may not be the name that the compiler will use but it will use something equivalent. It will as if create a temporary variable of type double which is say `temp_i`. And take the value of `i`, rewrite it in the form of a double number, double literal. So, it will do an explicit conversion based on this instruction of explicit double conversion. And `temp_i` now becomes a double representation of the value carried by `i`. It may be exactly the same. It may be a little bit different or it may be substantially different. who knows?

Second, what it does? Having seen that it is this part of the expression has a type double. This part of the expression has a type int. In C terms it is mixed mode. In other words, it is being asked to perform a double, a division using a double value and an int value which does not exist. So, it decides to do a promotion. It decides to promote this on to double.

And this promotion is not mandated by an explicit casting to double. So, it has to perform an implicit cast, implicit conversion. But the same game is again involved that it cannot keep that converted value in j. So, it needs another temporary variable. It has another temporary value where it converts and keep that value. Now, it has the original values explicitly converted and implicitly converted, 2 to double temporary variables, temp\_i and temp\_j. So, it will jolly well go ahead and do a division and return that result.

(Refer Slide Time: 8:25)

**Type Casting**

- Why type casting?
  - Type 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
  - Often the type needed for an expression that cannot be obtained through an implicit conversion. There may be more than one standard conversion that may create an ambiguous situation or there may be disallowed conversion. We need explicit conversion in such cases
- 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; }
```

*double = temp\_i / temp\_j*

```
int t; t = (int) t;
```

*int*

```
// compiler generates
double f (int i, int j) {
    double temp_i = i; // Explicit conversion by (double) in temporary
    double temp_j = j; // Implicit conversion in temporary to support mixed mode
    return temp_i / temp_j;
}
```

Programming in Modern C++ Partha Pratim Das M32.7

The things would be even more complicated, interesting if instead of return type being double if this were int. If this were int, then this will not work because this will give a double value. So, what would you again have to do? Possibly t gets say double t gets temp\_i divided by temp\_j. Then it has to do an int t\_t which has to take the value of t, internally explicitly cast it to an integer. But to you it will look like implicit because it has to convert; because it has to give you back the int. Some more code will be there. So, my whole idea is to show you that when I do this kind of conversion, actually there may be code that are generated. There may be computations that are generated which are involved and we have to be careful about those.

(Refer Slide Time: 9:21)

Module M32  
Partha Pratim Das  
Objectives & Outlines  
Type Casting  
Update & Download  
Cast Operators  
const\_cast  
Module Summary

- Various type castings are possible between built-in types

```
int i = 3;
double d = 2.5;

double result = d / i; // i is cast to double and used
```

- Casting rules are defined between numerical types, between numerical types and pointers, and between pointers to different numerical types and void
- Casting can be implicit or explicit

```
int i = 3;
double d = 2.5, *p = &d;

d = i; // implicit: int to double
i = d; // implicit: warning: '=' : conversion from 'double' to 'int': possible loss of data

d = (double)i; // explicit: int to double
i = (int)d; // explicit: double to int

i = p; // error: '=' : cannot convert from 'double*' to 'int'
i = (int)p; // explicit: double* to int
```

Programming in Modern C++ Partha Pratim Das M32.7

So, these are the typical castings we have seen. We have seen that it could be implicit or explicit. Just quickly recapitulating what happens in C which will also most often happen in C++ with some deviations though. So, int can be implicitly cast to double without any complaint. double can be cast to int with warning because there is possible loss of data. If you do explicit everything will be silent. We have seen the comparisons. But with the pointer, the implicit cast of a pointer to an integer will be refused. But explicit cast will be acceptable. We saw nuances of that as well.

(Refer Slide Time: 10:16)

Module M32  
Partha Pratim Das  
Objectives & Outlines  
Type Casting  
Update & Download  
Cast Operators  
const\_cast  
Module Summary

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

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

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

p = (A*)&b; // explicit on pointer: type cast is okay for the compiler
q = (B*)&a; // explicit on pointer: type cast is okay for the compiler
```

Programming in Modern C++ Partha Pratim Das M32.8

That was about the built-in types. If we came to the unrelated types, so we saw several things that cannot be done. You cannot convert an object implicitly or explicitly by casting from one type to another. These are none of these are allowed. You cannot convert their pointers implicitly but you can (using the C style) you can explicitly cast the pointer of one type to another type even though they are unrelated. Very dangerous but you can still do it.

We will see refinements of this. The reason I am just brushing up your memory on this is that in coming to C++, we will have finer rules for doing this and finer control of doing this. In C, it was only implicit or you know, C style explicit. Here we will have semantic minute differences coming in.

(Refer Slide Time: 11:14)

**• 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; // Forced casting on pointer: Dangerous
q = (B*)&a; // Forced casting on pointer: Dangerous

cout << p->i << endl; // prints -858993459: GARBAGE
cout << q->d << endl; // prints -9.25596e+061: GARBAGE
```

Programming in Modern C++ Partha Pratim Das M32.9

We also saw that we can do forced casting between unrelated classes as we did here. Forced casting of pointers between two unrelated classes and we saw that what kind of error it could give rise to because one class has an integer and the other class has a double. So, when I cast the pointer of the type of one class into the pointer of type of the other class, then I am actually interpreting an int and trying to print data as a double or otherwise. So, I get all sorts of garbage values. This kind of things will have to be avoided.

And in C++ we will try to you know anything that can that might lead to runtime error, we will try to build mechanisms in casting so that it does not wait up to the runtime error. It can give me the error earlier in compiled time. So, that I do not get surprised because this could be this kind of, you know, erroneous value could be hidden in a lot of deep computation and

debugging that would be a practical nightmare. So, I want to avoid that. And that is the reason we are reminding of the places where things can go wrong.

(Refer Slide Time: 12:26)

The slide displays the following content:

- Header:** Casting on a Hierarchy: C-Style: RECAP (Module 26)
- Text:** Casting on a hierarchy is permitted in a limited sense
- Code:**

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

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

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

pv = pa; // Okay, but lose the type for A * to void *
pv = pb; // Okay, but lose the type for B * to void *

pa = pv; // error: '=' : cannot convert from 'void *' to 'A *'
pb = pv; // error: '=' : cannot convert from 'void *' to 'B *'
```
- Diagram:** A class hierarchy diagram with 'A' at the top and 'B' below it. A solid arrow points from 'A' to 'B', indicating inheritance. Hand-drawn blue arrows point from 'B' back to 'A', representing upcast.

The third type that we had seen are relating to hierarchies. That is when we have an inheritance hierarchy, one class is derived from the other and we saw the pointers of this and we saw upcast is safe. That is at any point of time, I can take a pointer to a more specialized class and assign it to the pointer of a less specialized or more generalized class. That is because specialization keeps on growing the object details or the concept details.

So, if I take if I really have one and treat it as more generalized, then I do not lose information. But if I do the other way around which is downcast. That is trying to take an object of class A and treat it as if as an object of class B, I will have severe consequences. We have seen examples of that reproduced here. So, this is something which will not be implicitly allowed.

With void\*, you are saying that is a pointer to I do not know what. I can take any pointer and put it to void\* implicitly. I will lose the type information but there is nothing wrong that is going on. But I obviously cannot do the reverse. I cannot take a void\* pointer and say that it is a pointer of type A or pointer of type B. Because I certainly I am trying to assume something which the compiler has no way to verify.



(Refer Slide Time: 14:19)

**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: 'dataB_' : is not a member of 'A'
```

So, this is this is nothing. None of these are new. I am just you know reminding of what you what we did. And we concluded that up-casting is safe. So, with the up-casting, what we have is when we have done the up-casting then if we use the right pointers, we can print everything.

If we are using the up-casting if we are trying to use like pa is pointer to class A, pointer of type class A and holding your object of class B. So, using pa I will never be able to print this data member. I get a compile time error which is fine. So, up-casting always is safe. Down-casting will lead to problems.

(Refer Slide Time: 15:12)

**Cast Operators**

**Casting in C and C++**

- 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` (searched by cast operator name) in source
  - **C-Style cast must be avoided in C++**

*Handwritten annotations:*  
 - A circle around the C++ list with arrows pointing to 'double d, int i;' and 'd = i;'.  
 - 'd = i;' is underlined.  
 - 'static\_cast' is underlined.  
 - 'double' is written below the circle.  
 - 'C-Style cast' is written in red above the circle.  
 - 'double (i)' is written in blue next to the circle.

So, that was about again the quick recap of what we have in terms of casting in C++ as inherited from C. Now, C++ deals with casting differently from C. So, in summary, in C we have implicit cast. We have explicit C style casting. We might lose type information in several context and there is complete lack of clarity in terms of the semantics. What do we mean? Everything is, take type one, make it type two is all that we can say. But under what context? Under what context is this treatment of one type as another is valid? C does not allow you to say that.

Now, in C++, firstly what we will have to understand is there is casting which does fresh inference about that object without actually changing anywhere. It is not changing anywhere. But it is just making new inferences about the properties of that object. There are castings of that type.

And of course, there are castings of the original form that we were saying where they actually make fresh inferences with changing the value. So, if you change int to double or double to int, you are making fresh inferences about the type but with change of value. But C++ also allows you to do similar things without changing the value. We will see the example.

Now, this can be done second can be done using implicit conversion or explicit user-defined conversion which we will have to learn. What is user-defined conversion? Does not exist in C, so you do not know. The target is to preserve type information in all contexts. Do not lose the type information. C++ is strongly typed. You cannot lose the type information. So, that is the basic objective.

So, with that, C++ provide clear semantics through 4 different cast operators, `const_cast`, `static_cast`, `reinterpret_cast` and `dynamic_cast`. Now, other than anything else, the big advantage of (you know) using C++ style cast is, they can be searched by the name of the operator. Let me, let me tell you what I what they mean.

Suppose I say I said d. This example we have been using, `double d; int i`. This is valid in C, implicit. In C++, writing it in the C++ way, I might write this as `static_cast`. A little bit of writing I agree. But what does it say? It says that take this value i. i is defined as int so you know at this point that it is of type int and cast it to a double value which is possible here because you know that the value of the type of d is double.

Now, what is the difference between these two? In terms of actual computation, there will be no difference. But the core difference is the fact that when you do this, you can search for `static_cast<double>` in the code. You cannot search for this. There is no textual, you know kind of fingerprint to say that this is where the conversion has happened. It is all implicit. It is all inside. But using this you can get that.

Even it is, it is better than simply writing double because this can happen in multiple different contexts. You do not know in what semantic context you are changing something into double. But here the name of the cast operator will tell you that. Those are the nuances that make the C++ cast operators really great to work with and that is what we are going to learn. Having learned that C++ type, C style; C style of casting must be avoided in C++ altogether. You do not need them.

(Refer Slide Time: 20:09)

**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

Programming in Modern C++ Partha Pratim Das M32.14

Now, how does the cast operator look? You have already seen that. It has a name and is an expression which is the source type. The type of the source expression is not specified because you already know it implicitly and the specification of the target type to which it is going. So, three things as we have seen here, `static_cast`, this is the name. This is the target type, `double` and this is the expression `i`. It could have been an expression `i + j * 3`. If `i, j` are integer. I know `i + j` times 3 is of type `int`. So, the source type is `int`. The target type is `double` and the semantics of the cast is `static_cast`. This is a basic form of the expression of every cast operator in C++.

(Refer Slide Time: 21:22)

**Cast Operators**

- **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

Programming in Modern C++ | Partha Pratim Das | M32.15

So, we have 4 of them. The `const_cast` operator; and all 4 of them have the same structure. The name within corner brackets, the target type; within parenthesis, the source expression. The `const_cast` as the name suggests is to deal with the const-ness or volatility. It kind of overrides the const-ness of the expression that was already there. Like it changes a const expression to a non-const expression and so on.

`Static_cast` which is the which is a non-polymorphic cast which converts the expression of one type to another type. And there are certain context in which this conversion is allowed. If you try to `static_cast` in any other context, you get a compilation error which is great. And this static cast could happen implicitly or it can be defined by the user, `reinterpret_cast` is very interesting. If you cannot cast something, if you cannot cast something by the normal rules rather you should not be doing that cast, then you can still do that cast in many cases using the `reinterpret_cast`.

For example, you can cast unrelated pointer types. Most importantly you can cast pointer and integer by using `reinterpret_cast`. As the name suggests, `reinterpret`. It does not know how to think of as an integer as a pointer? A pointer is an address or a pointer as an integer, the other way. So, you have as if you take the bit pattern and give a new interpretation, `reinterpret_cast`.

And unlike `static_cast` it does not perform any computation. So, you can see I said that it could just, casting could just give a different meaning, different inference but not change the value. `const_cast` does not change value. `reinterpret_cast` does not change value but gives it a different meaning. Whereas `static_cast` may actually change value.

The last but the most interesting is a `dynamic_cast`. All these are compiled time. Whereas, this one is runtime. That is, it performs a runtime casting that verifies whether a casting is valid or not. At runtime you cannot do that because you do not know the actual object. But at the runtime you can do that.

So, `dynamic_casting` takes an object at the runtime and casts it to some other type, if that casting is valid. And for that, it may do some, it may perform some predefined computation. `dynamic_cast` is allowed only on pointers and references. So, if you feel the validity of a `dynamic_cast`, then what the `dynamic_cast` operator do is, it gives you a null pointer and you know that you have failed.

If you have invoked a `dynamic_cast` on a reference, then it throws an exception because there is no null reference. These are the ways at the runtime to tell you that something wrong has happened. Whereas, for these, if you have unacceptable things going on, conversions going on, then the compiler will give you error. That is the basic summary story of cast operators in C++.

(Refer Slide Time: 25:29)

const\_cast Operator

- `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

Programming in Modern C++ Partha Pratim Das M32.16

So, having said that, let us talk specifically about `const_cast` which converts types of different cv-qualification, `constvolatile`-qualification. And `const_cast` is the only way to move, I mean remove the const-ness or volatility of certain object. But it usually does not perform any computation or change the value.

(Refer Slide Time: 25:54)

const\_cast Operator

```
#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)); // Okay

  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); // Okay

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

Programming in Modern C++ Partha Pratim Das M32.17

Let us look at an example. So, just here is a class which has a data member `i`, the constructor. There is a const member function `get()`, const member function `get()`. There is a member function `set()`. So, all that what can you infer from here? You can infer that this const member function can be invoked only on const object whereas the non-const member function can be invoked on const as well as non-const objects. We have seen this. Then I have a global print

function. So, here is a pointer to a constant string. This side is constant. So, the pointer to a constant string, so I cannot change the string.

(Refer Slide Time: 26:50)

```
#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)); // Okay

    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); // Okay

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

Now, if I try to pass this as print as a parameter to print, I will get an error. Why should I get an error? Because I am passing a pointer to a constant object, constant string to a pointer value where it is a non-constant object. So, what will happen? It is a call by value. So, if I allow this, then this address will be copied here in terms of str. And by changing str, I am not changing it here in this particular case. But the compilers interpretation is by changing str, I can actually change the original string. So, it will not allow me. It will say that the conversion from const char\* to char\* is not allowed.

(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(c); // error: 'void print(char *)': cannot convert argument 1 from 'const char *' to 'char *'
    print(const_cast<char *>(c)); // Okay

    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); // Okay

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

So, the question is, if I have that, how do I call that function? Now, I know from the manual of this library, from the documentation of this library that, this print function does not do any harm to the string which is passed. So, it is okay to pass a constant string. So, what I do I stripped the const-ness. What was the type? It was const char\*. So, the source type of C is const char\*. What is the target type? I have given char\*. So, the const is gone. So, the resultant expression is only char\* which matches the formal parameter type of print and everything is okay. That is the basic purpose.

(Refer Slide Time: 28:38)

The slide shows the following C++ code:

```
#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)); // Okay

    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); // Okay

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

```

Handwritten annotations on the slide include:

- A blue circle around `const A a(1);` with an arrow pointing to `const A&>(a)` in the line below, with the note "const A&".
- A blue circle around `const_cast<char *>(c)` with an arrow pointing to `const_cast<A&>(a)` in the line below, with the note "const A\*".

Suppose I have defined a constant object a. I do a.get. That will work because it is a const member function. If I do a.set, it will give me an error. It is supposed to because set is a non-constant member function. It can change my object. So, validly I will get an error. But if I want to force that well, it may be the case but I want this non-const member function to be called. What I need to do? I need to take away the const-ness. So, what I do? I this object is const A. So, its type is const A&.

So, what I do is? I take away that const and I just give a target type which is A& which makes it a non-constant object. That is - it takes away the const, the fact that this pointer was pointing to a constant A object. It can now point to a non-constant A object and therefore calling a set is perfectly.



(Refer Slide Time: 29:58)

The slide displays the following C++ code with annotations:

```
#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)); // Okay

    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); // Okay

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

Programming in Modern C++ Partha Pratim Das M32.17
```

Annotations include blue circles around `const_cast<A&>(a)` and `const_cast<A>(a)`, and a blue arrow pointing from the error message for `const_cast<A>(a)` to the `const A` type in the class definition.

Mind you cannot do this. You cannot do this. You cannot convert const A to A. That is not allowed because that will mean changing the object. When you are doing A&, all that you are doing you are actually temporally creating another reference which is a non-constant reference which is allowing you to go through the call. But you cannot inherently change the object. So, it is not necessarily that the cast operators will also always succeed. This is a compilation error.

(Refer Slide Time: 30:36)

The slide displays the following C++ code with annotations:

```
#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 from 'const A' to 'A'
    ((A)a).set(5); // C-Style Cast ✗
}
}

Programming in Modern C++ Partha Pratim Das M32.18
```

Annotations include blue brackets grouping the class definition and the `const A a(1);` line. Blue checkmarks and crosses are placed next to the C-style cast lines. Blue circles highlight the `const_cast` and `(A&)` / `(A)` casts.

Now, look carefully. We want to; the same example. Here nothing changed. Nothing changed. Nothing changed. Nothing changed. Now, all that we are showing is what happens

with the C-style cast. If I do this instead, I can do this. This will work. C-style casting. Just force it without saying what you are doing.

So, you are missing out on two things. So, you are missing out on two things. One is, as I said it is not easy to find out where the casting has happened. Your char\* could be everywhere. Second, even if you find out you do not know what you are casting away. Are you converting c which was an object type into char\* or something else. This clearly tells you that you are ripping off the const-ness of the pointer. Nothing else. So, makes good sense but C-style cast will force this.

You could do this as you saw. You can C-style, you can do this. Again, the same complain that finding out A& and knowing that there is a casting is difficult. And second, you do not know the meaning for which it is done. The third dangerous part is, this is semantically not allowed. But if you do it in C, it will allow you. C-style will allow it. So, you can see that there is not only type conversion but there is a semantic difference between C looks at it and C++ looks at it. C++ is lot more strict.

Say if an object is constant, it is constant. You can not make it non-constant. You can have a different reference to it which treat it as a non-constant. Invoking function. You could have a pointer that treats it that way. But the object by itself does not become non-constant which C, using C-style will violate. That is so it is a disaster. Basic recommendation: do not use any one of these.

(Refer Slide Time: 32:47)

**const\_cast Operator**

```
#include <iostream>
struct type { type(): i(3) { }
void m1(int v) const {
    this->i = v; // error C3490: 'i' cannot be modified -- accessed through a const object
    const_cast<type*>(this)->i = v; // Okay 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; // Okay: modifies i
std::cout << "i = " << i << '\n';

type t; // note, if this is const type t, then t.m1(4); may be undefined behavior
t.m1(4);
std::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
std::cout << j << " " << *pj << std::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
}
```

**Output:**  
i = 4  
type::i = 4  
3 4

Programming in Modern C++ Partha Pratim Das M32.19

Finally, here there are some more example for you to understand. Here is a struct type, a constant member function. If you have a constant member function, naturally do can not do it, do this in it because using this pointer you cannot change i. So, you cannot do that. But you can strip off the const-ness of the this pointer and do this. So, you can force that by stripping off the const-ness here.

Here is an integer variable and a constant reference to that. You cannot certainly assign anything to this cref\_i but you can strip off the const-ness of the reference and make an assignment to it. If you will print, you will get the value 4. You have a type t object. You can invoke t.m1. m1 is the const member function. So, non-const object can always call const member function. If you do that, it will get changed to 4 and 4 will be printed.

Mind you. If you have type t as const, then also you will be able to invoke this. t is const. On a const object you can always do non-const, you can always invoke const member function. But if you invoke that, then the results will be unpredictable. Because you have a constant object by definition and you are forcibly changed something inside. Do not do that. Use the path of mutable and all those.

(Refer Slide Time: 34:41)

The slide is titled "const\_cast Operator" and features a video feed of the presenter in the top right corner. The main content is a code editor showing several examples of const\_cast usage with annotations and output results.

```

#include <iostream>
struct type { type(): i(3) { }
    void m1(int v) const {
        //this->i = v; // error C3490: 'i' cannot be modified -- accessed through a const object
        const_cast<type*>(this)->i = v; // Okay 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; // Okay: modifies i
    std::cout << "i = " << i << '\n';

    type t; // note, if this is const type t;, then t.m1(4); may be undefined behavior
    t.m1(4);
    std::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
    std::cout << j << " " << *pj << std::endl;

    void (type::*mf)(int) const = &type::m1; // 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
}

```

Output:

```

i = 4
type::i = 4

```

Annotations on the slide include blue arrows pointing to the const\_cast operations and underlines under the output values. The slide footer contains "Programming in Modern C++", "Partha Pratim Das", and "M32 19".

The screenshot shows a slide titled "const\_cast Operator" with a video feed of the presenter in the top right. The slide content includes:

```

#include <iostream>
struct type { type(): i(3) { }
    void m1(int v) const {
        //this->i = v; // error C3490: 'i' cannot be modified -- accessed through a const object
        const_cast<type*>(this)->i = v; // Okay 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; // Okay: modifies i
    std::cout << "i = " << i << '\n';

    type t; // note, if this is const type t, then t.m1(4); may be undefined behavior
    t.m1(4);
    std::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
    std::cout << j << " " << *pj << std::endl;

    void (type::*mf)(int) const = &type::m1; // 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
}

```

Output:

```

i = 4
type::i = 4
3 4

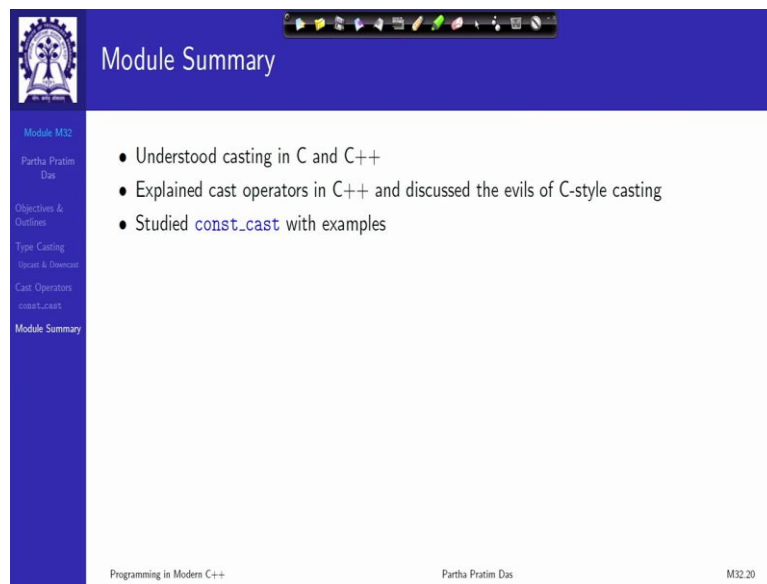
```

Navigation icons are visible at the top, and a sidebar on the left lists "Module M32", "Partha Pratim Das", "Objectives & Outlines", "Type Casting", "Unsafe & Downcast", "Cast Operators", "const\_cast", and "Module Summary". The footer contains "Programming in Modern C++", "Partha Pratim Das", and "M32.19".

Similarly, we have here. These are just different examples of const-ness. Here is a const variable `j` and I have created a pointer stripping off the const-ness. So, looking from this pointer it is; so I can now change this. Now, the interesting thing is, `pj` points to `j` but `j` and `pj` are not the same values. Print `j` and `pj`. This `j` remains to be 3; `pj` is 4. So, that is what I was meaning. You cannot change the const-ness of the object. You can have a non-const view of the object created by putting this separate reference or pointer.

And this can be risky. This is an undefined behaviour. Your basic assumption that the pointer and the object it points to are same is destroyed. Why does that happen? Because to be able to do this, compiler creates a temporary to put the value of `i` and let `pj` point to it. And it is making changes to that temporary, not to your `j`. Very risky. Here is one more example with function pointer. You can see that even with casting, `const_cast`, you cannot change the constant member function to a non-constant member function. Simply because `const_cast` is not allowed to work on the function pointers.

(Refer Slide Time: 36:15)



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

Module M32  
Partha Pratim Das  
Objectives & Outlines  
Type Casting  
llocast & Downcast  
Cast Operators  
const\_cast  
Module Summary

Programming in Modern C++ Partha Pratim Das M32.20

So, these were the basic high points about the `const_cast`. To summarise, we have understood casting in C and C++. I am trying to bring out the different nuances and differences. And having seen the summary of different C++ operators, we have just studied `const_cast` with examples. And thank you very much for your attention. In the next module we will talk about the other cast operators.