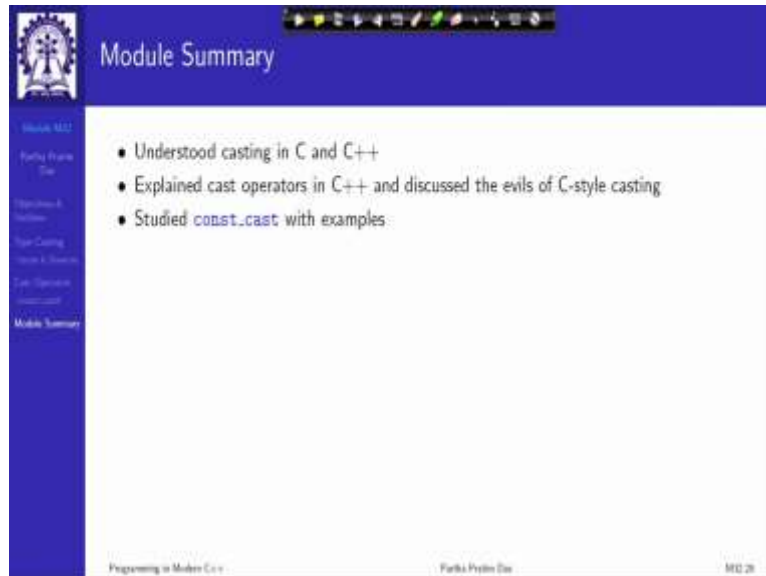
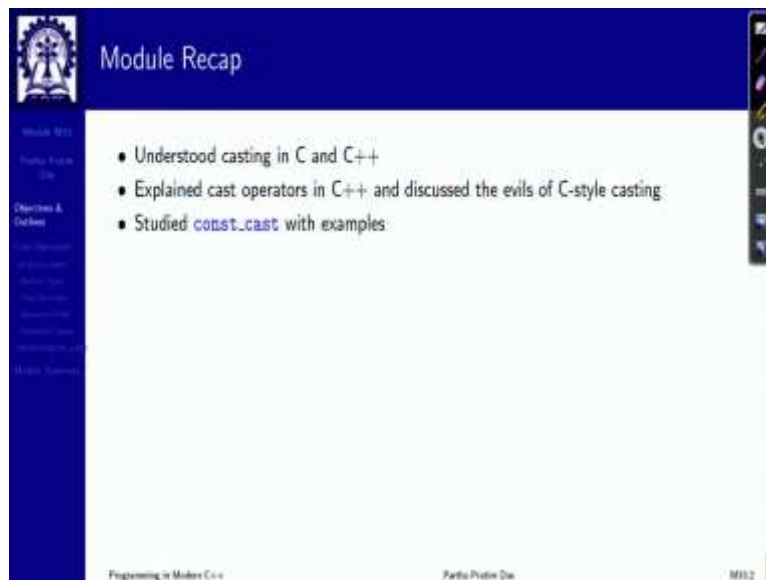


Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture 33
Type Casting & Cast Operators: Part 2

(Refer Slide Time: 0:36)



The slide titled "Module Summary" features a blue header with the IIT Kharagpur logo and navigation icons. A vertical sidebar on the left lists navigation options: "Module M02", "Partha Pratim Das", "Objectives & Outcomes", "Type Casting", "Cast Operators", "Introduction", and "Module Summary". The main content area contains a bulleted list: "Understood casting in C and C++", "Explained cast operators in C++ and discussed the evils of C-style casting", and "Studied `const_cast` with examples". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M02 28".

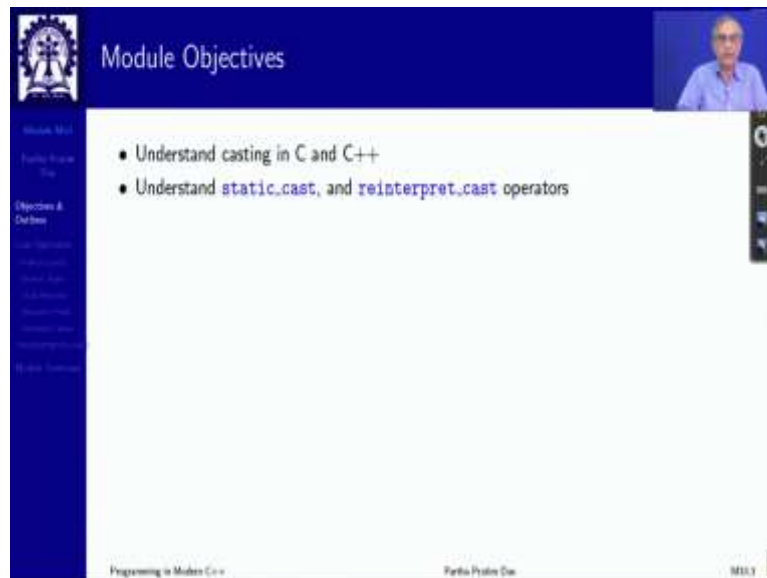


The slide titled "Module Recap" has a similar layout to the summary slide. The sidebar on the left lists: "Module M02", "Partha Pratim Das", "Objectives & Outcomes", "Type Casting", "Cast Operators", "Introduction", "Introduction to C++", "Introduction to C++", "Introduction to C++", "Introduction to C++", "Introduction to C++", and "Module Summary". The main content area contains the same bulleted list: "Understood casting in C and C++", "Explained cast operators in C++ and discussed the evils of C-style casting", and "Studied `const_cast` with examples". The footer includes "Programming in Modern C++", "Partha Pratim Das", and "M02 29".

Welcome to Programming in Modern C++. We are in week 7 and we are going to discuss module 33. From the last module, we have started discussing typecasting in C++. In module 26, we had discussed this for C. And what we are taking look at is, in C++ you have different cast operators for specific semantics of casting that you need. And in this context, we have seen one cast operator, `const_cast` with examples which can be used to remove const-ness,

volatility, cv-qualifiers, manipulate with the cv-qualifiers of different pointers, references and so on.

(Refer Slide Time: 1:28)



The slide is titled "Module Objectives" and features a dark blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white and contains two bullet points:

- Understand casting in C and C++
- Understand `static_cast`, and `reinterpret_cast` operators

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Part 6: Primitives", and "M11.1".



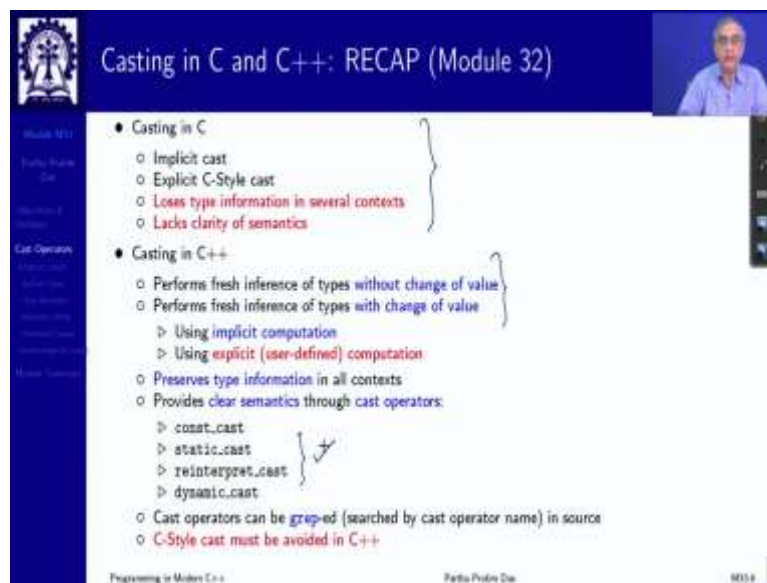
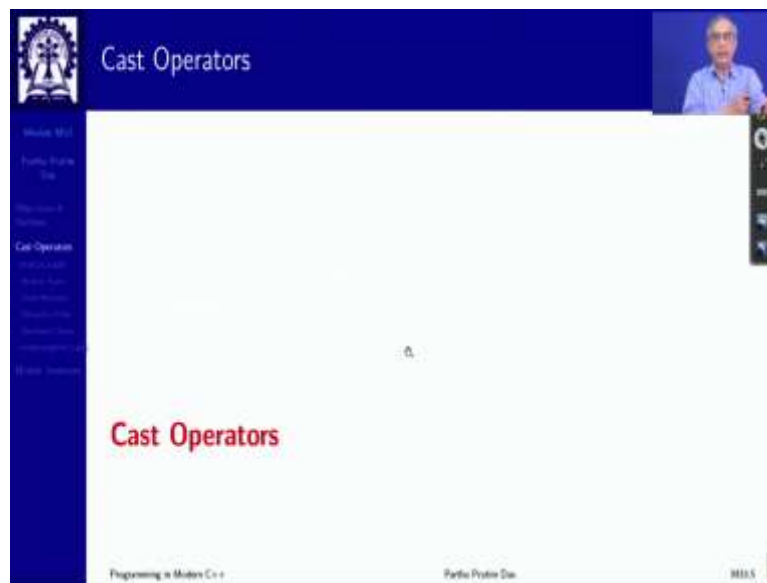
The slide is titled "Module Outline" and features a dark blue header with a logo on the left and a small video feed of the presenter on the right. The main content area is white and contains a numbered list of topics:

- 1 Cast Operators
 - `static_cast`
 - Built-in Types
 - Class Hierarchy
 - Hierarchy Pitfall
 - Unrelated Classes
 - `reinterpret_cast`
- 2 Module Summary

At the bottom of the slide, there is a footer with the text "Programming in Modern C++", "Part 6: Primitives", and "M11.1".

In the present module, we will continue on this and we will take a look at two other operators `static_cast` and `reinterpret_cast`. So, this is outline which will be on the left panel as usual.

(Refer Slide Time: 1:48)



So, before we get into the meat of discussion in this module, let us just quickly recap what we have seen in the last module. That what are the types of casting that C cast provider us and why do we need separate specific casting operators for C++ particularly to have fresh inference of types without or with change of value.

I will remind you again that C++ is a strongly typed language. Therefore, it is always necessary for the developer as well as the compiler to understand the type of every literal, every variable, every expression and deal with their conversions casting with a lot of care. So, we have seen the `const_cast`. Today we are going to discuss about `static_cast` and `reinterpret_cast` with a specific context in which they are useful.

(Refer Slide Time: 3:01)

static_cast Operator

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

Programming in Modern C++ Part 6: Pointers and References 10:17

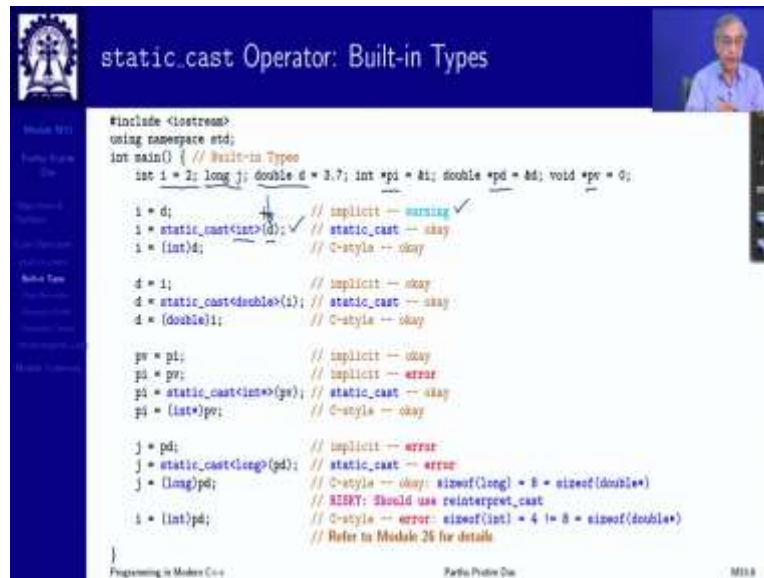
So, we start with the `static_cast` operator which is which in a way is a common kind of C equivalent casting in C++. With this operator which can perform all conversions that are allowed implicitly. So, if we if something is allowed implicitly, you can write specific typecast operator in that place. We will see examples. It also allows us to do some of the you know opposite things like by implicit conversion you can convert any pointer type to `void*` but certainly you cannot convert `void*` to pointer type which you can do by `static_cast` in case you need to really reassign type to a pointer.

You can convert integer, floating point values to enum types. You can convert one enum type to another enum type and so on. You have to be very careful always. Because when you are doing these things, you must know that what you really mean and what you are really doing. `static_cast` can perform conversions of pointers, pointer types to related classes. Certainly, it can do upcast because it is implicit. But it can also do downcast though that is not the proper way to do downcast. We will see in the next module; we will discuss about the downcast operator specifically.

But `static_cast` also allows you to force a downcast if you really need to do that. But the caveat in that is when it does it downcast, it does not check at the runtime. Because `static_cast` as a name suggests is at the compiled time, is static type. So, it does not check any real existence of the downcast object at the runtime. And therefore, downcast perform using `static_cast` could lead you to some problems. We will see those examples.

Additionally, `static_cast` can do perform conversion based on a single argument constructor or a conversion operator. This will be these will be new concepts which are called user-defined cast. It can convert to rvalue reference. It can convert enum to integer or floating point. It can convert any type to void that is do the evaluation and just forget about, discard the value. right? So, these are the different kinds of things that `static_cast` is capable of giving us.

(Refer Slide Time: 5:57)



```
#include <iostream>
using namespace std;
int main() { // Built-in Types
    int i = 2; long j; double d = 3.7; int *pi = &i; double *pd = &d; void *pv = 0;

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

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

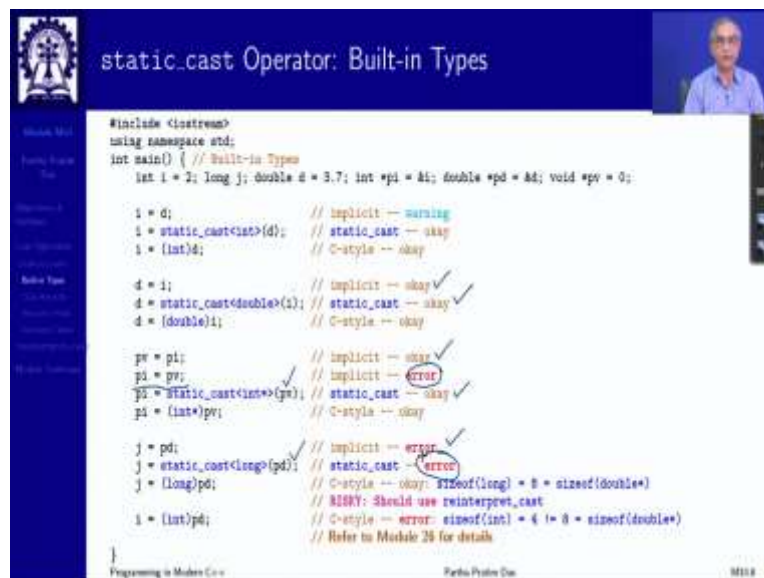
    pv = pi; // implicit -- okay
    pi = pv; // implicit -- error
    pi = static_cast<int*>(pv); // static_cast -- okay
    pi = (int*)pv; // C-style -- okay

    j = pd; // implicit -- error
    j = static_cast<long>(pd); // static_cast -- error
    j = (long)pd; // C-style -- okay: sizeof(long) = 8 = sizeof(double*)
    // ERROR: Should use reinterpret_cast
    i = (int)pd; // C-style -- error: sizeof(int) = 4 != 8 = sizeof(double*)
    // Refer to Module 26 for details
}
```

So, we will start with extending the example of built-in typecasting that we had seen in C. This is, note `i` is an integer, `j` is a long, `d` is a double and I have `pi` and `pd` and `pv` as three different pointer types. Now, if I cast a double to integer, implicitly it is allowed we have seen but it gives a warning. But if I do that with `static_cast`, it will the compiler will be silent because compiler knows what you are doing. So, you can again get reminded of the style of doing this. Since we are converting, we need to know the expression and its type that is there in the source. And we need to know the target type.

So, the type of source does not need to be specified because once you put the expression within this parenthesis the parameter of the `static_cast`, you will automatically get to know the type. Because it is strongly typed, so it has a type. So, that is a source type which is double here. And this is the target type which is int. So, this style continues.

(Refer Slide Time: 7:22)



```
#include <iostream>
using namespace std;
int main() { // Built-in Types
    int i = 2; long j; double d = 3.7; int *pi = &i; double *pd = &d; void *pv = 0;

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

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

    pv = pi; // implicit -- okay ✓
    pi = pv; // implicit -- error
    pi = static_cast<int*>(pv); // static_cast -- okay ✓
    pi = (int*)pv; // C-style -- okay

    j = pd; // implicit -- error
    j = static_cast<long>(pd); // static_cast -- error
    j = (long)pd; // C-style -- okay: sizeof(long) = 8 = sizeof(double*)
    // RISKY: Should use reinterpret_cast

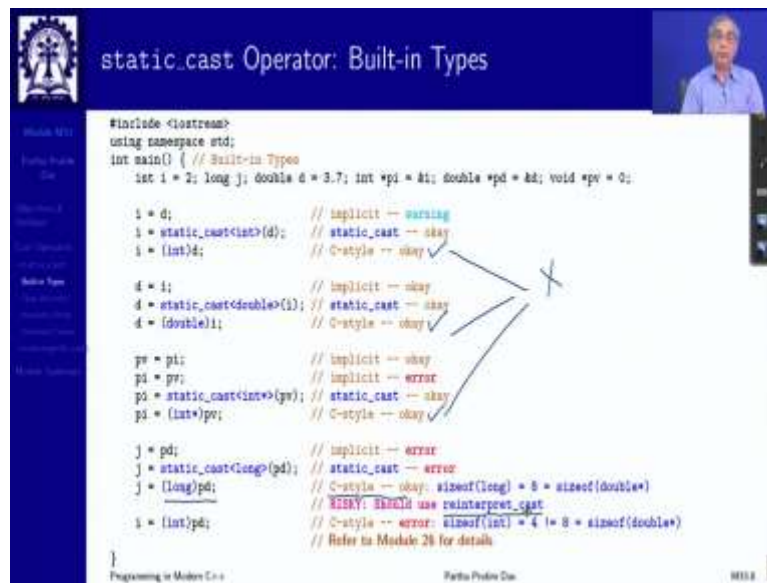
    i = (int)pd; // C-style -- error: sizeof(int) = 4 != 8 = sizeof(double*)
    // Refer to Module 26 for details
}
```

Now, this was from int to double. I am sorry. This was from double to int. The next is from int to double which is implicit silent even for that because it is going to a bigger type. And certainly you can do a static_cast for that. You can convert a pointer to a pointer to void implicit. Obviously, you can do that by static_cast.

But you cannot do convert a pointer to void to a specifically type pointer like pointer to integer here. But using static_cast, you can do that. Using static_cast you can do that and the compiler will be silent. It is to do that. Which means that compiler understands that if you are converting a pointer to void to say a pointer to integer, then you know exactly what you are doing.

You can now finally let us look at the conversion between pointer and integral values. We discussed this at length in Module 26. So, if you try to do an implicit cast from say a pointer to double to long, it is an error. If you try to do it using static_cast, also you will get an error. Because static cast is does not have a semantics to convert a pointer to a long. Converting pointer to a long is just a bit level reinterpretation. It is not a type level conversion because these two types are in no way related; no way they can be used. So, even with a static_cast, this is not permitted.

(Refer Slide Time: 9:17)



```
#include <iostream>
using namespace std;
int main() { // Built-in Types
    int i = 2; long j; double d = 3.7; int *pi = &i; double *pd = &d; void *pv = 0;

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

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

    pv = pi; // implicit -- okay
    pv = pv; // implicit -- error
    pv = static_cast<int*>(pv); // static_cast -- okay
    pv = (int*)pv; // C-style -- okay ✓

    j = pd; // implicit -- error
    j = static_cast<long>(pd); // static_cast -- error
    j = (long)pd; // C-style -- okay: sizeof(long) = 8 = sizeof(double*)
    // RISKY: Should use reinterpret_cast
    i = (int)pd; // C-style -- error: sizeof(int) = 4 != 8 = sizeof(double*)
    // Refer to Module 26 for details
}
```

Now, in parallel to this, I have also put the C style of casting in every place. In all of these three if you look at the C style is possible but our recommendation is never use them because they cannot be. They are not safe. They are not cannot be easily searched and so on. Now, in case of pointer to integral type or vice versa, if you use a C style casting as you are doing here, then this will be okay. But obviously, that does not mean that we are promoting C style casting for this case. We will see there is a separate specific cast operator given known as reinterpret_cast which will make it possible to do these things in a proper way.

(Refer Slide Time: 10:17)



```
#include <iostream>
using namespace std;
int main() { // Built-in Types
    int i = 2; long j; double d = 3.7; int *pi = &i; double *pd = &d; void *pv = 0;

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

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

    pv = pi; // implicit -- okay
    pv = pv; // implicit -- error
    pv = static_cast<int*>(pv); // static_cast -- okay
    pv = (int*)pv; // C-style -- okay ✓

    j = pd; // implicit -- error
    j = static_cast<long>(pd); // static_cast -- error
    j = (long)pd; // C-style -- okay: sizeof(long) = 8 = sizeof(double*) ✓
    // RISKY: Should use reinterpret_cast
    i = (int)pd; // C-style -- error: sizeof(int) = 4 != 8 = sizeof(double*) ✓
    // Refer to Module 26 for details
}
```

While you do this conversion again be reminded, we discussed in Module 26 at length that if you try to cast this in C style to long. It will be. But if you try to cast this in C style to int, you

will get an error. This is for a machine where the long and the pointer are of size 8 bytes. But int is of size 4 bytes.

So, this is a necessity that when we convert pointer to integral type, the integral type must be large enough to hold the pointer. So, that same error will continue. And just the take back here is static_cast cannot be used for these conversions and we need a separate cast operator for this. And never never use the C style casting.

(Refer Slide Time: 11:08)

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

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

int main() {
    A a;
    B b;

    // UPCAST
    A* p = 0;
    p = &b; // implicit -- okay ✓
    p = static_cast<A*>(&b); // static_cast -- okay ✓
    p = (A*)&b; // C-style -- okay ✓

    // DOWNCAST
    B* q = 0;
    q = &a; // implicit -- error ✓
    q = static_cast<B*>(&a); // static_cast -- RISKY: Should use dynamic_cast ✓
    q = (B*)&a; // C-style -- okay ✓
}
```

Again, relook at the class hierarchy. A is a base class and B is a derived class here. I have a A object and a B object. I have an A pointer p and I am setting the address of b which is a derived class object. Which means that I am going from specialization to generalization and that is implicitly. That is okay with the static_cast. Obviously, with the C style cast but we would not recommend it.

If I try to do the reverse, if I try to do a downcast that is take a pointer to the specialized class, derived class B and try to convert a A object, the generalized object to that. We have seen several reasons as to why this is not advisable. This is not possible in general. So, implicit cast has an error which is the perfect way to go.

But by static_cast, by doing a static_cast as in here, you can force this downcast. Again, this is not recommended because really when you are forcing this down, you do not know whether there is enough information available in this source object, source expression to evaluate that specialized object, the target expression. For this we will discuss in the next module another special cast operator called dynamic_cast. Obviously, C-style casting will

work but not recommended to be used. So, to conclude for related classes on the hierarchy, static_cast should be used for doing upcast.

(Refer Slide Time: 13:08)

```
class Window { public:
    virtual void onResize(); ...
}
class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        static_cast<Window>(*this).onResize(); // cast *this to Window, then call its onResize;
        // this doesn't work!
        ... // do SpecialWindow-specific stuff
    }
    ...
};

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

class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        Window::onResize(); // Direct call works
        ... // do SpecialWindow-specific stuff
    }
    ...
};
```

```
class Window { public:
    virtual void onResize(); ...
}
class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        static_cast<Window>(*this).onResize(); // cast *this to Window, then call its onResize;
        // this doesn't work!
        ... // do SpecialWindow-specific stuff
    }
    ...
};

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

class SpecialWindow: public Window { // derived class
public:
    virtual void onResize() { // derived onResize impl;
        Window::onResize(); // Direct call works
        ... // do SpecialWindow-specific stuff
    }
    ...
};
```

Now, here I will just point you to a little pitfall in terms of a very common mistake that many of us do. Just think about there is a class Window which has a resizing method, onResize. If you try to drag and resize the window, this function is called and the resizing drawings are done. Now, from that we have specialized to a SpecialWindow which may have some additional requirement when you resize it. So, when you resize what you would want? That the basic functionality of the resize of the Window, the parent class should be first called. So, that the basic resizing is done and then the extra work that you need for this SpecialWindow is carried out.

Now, naturally for this you want to call this function. So, one common mistake we do is we take the object which is the SpecialWindow object that is *this. SpecialWindow object and static cast it to the Window object. So, we say, "Okay we will static_cast". We are doing upcast because we are going up. But if you do that recall that this will actually lead to slicing because you are trying to copy a specialized object as a generalized object.

So, when you do that, you will have only the base part going. And how will the compiler handle that? Because compiler cannot lose original object. So, compiler creates a temporary and slices the base part of the SpecialWindow object as a temporary Window object and on that it calls the method.

It does not call the method on the current object. It is calling it on a temporary which you have created because of this static_casting and therefore you will see no effect. So, please remember that whenever you work on a hierarchy and you have to call the function of the base or base of base and or whatever, then you cannot you should never use the way to copy the larger, the derived class object as the base class object by casting.

What you should do is very simple. Just directly call the base class function, Window::OnResize. Just directly call it so that it is called on the current object. Effects also happens on the current object and everything works fine. So, please be careful about this pitfall. I have shown it in in one context of Windows programming but it is a very very general issue to happen.

(Refer Slide Time: 16:03)

The slide is titled "static_cast Operator: Unrelated Classes" and features a small video inset of a speaker in the top right corner. It displays two code snippets side-by-side, illustrating the behavior of static_cast with unrelated classes.

Left Snippet (Errors):

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

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

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

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

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

Right Snippet (Successful Casts):

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

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

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

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

    // int ==> B
    a = i; // Use A::A(int)
    a = static_cast<A>(i); // Use A::A(int)
    a = (A)i; // Use A::A(int)
}
```

A diagram on the right side of the slide shows a circle representing a temporary object. Arrows point from the temporary object to the successful casts in the right snippet, indicating that the compiler creates a temporary of the target type (A) when casting from an unrelated source (B or int).

Now, I will introduce you to what you can define as your own casting operator. For example, you have two unrelated classes, A and B. They are not related. They are not on the hierarchy. So, as we have seen any kind of conversion from A to B is error whether you do it implicitly, by C-style casting, by `static_casting`, it is an error. Similarly, if I try to convert say `int` to A, all of these are error. It is as it should be because they are unrelated. There is no semantics on that. So, what you can do is we can define the semantics for it. Because you know why is this an error? Why converting B to A is an error?

Because I have an B object. I have a B object. And what do I need as a target output? I need an A object. Now, A object has to be constructed. I cannot get any object just you know out of nothing. So, in C++ as we have understood is I really need an instance of that A object to be created from the B object only then the conversion has a meaning. Otherwise, what is the result of it? In the built-in type, this is not an issue. Because built-in types do not have constructors. They have built-in processes to create the data pattern from `int` to `double` or from `double` to `int` and so on.

But no such thing exists for types that we have defined, the class A and class B. So, very simply, what we need is if we need to convert a B type object to an A type, I need to provide a constructor for that. That is the simplest way of doing it. Just provide a constructor which takes a B object and gives you an A object.

As you provide that, all of these will become valid. Even the implicit cast will become valid because it gets a implicit cast sees that there is a B object. It sees that it needs an A object. And it finds that there is a constructor to allow you to do that. So, it will invoke that constructor. Similarly, you can do that by `static_cast` which is what I will recommend. C-style casting also allows it though we will not recommend that.

Using the same style, I can also do a conversion from `int` to A. Because again what it means? If I am converting an integer to the A object, it means that given an integer, I want to construct an A object which means a constructor of this form. An A object will be created by this constructor. So, all of these again will be correct. Even the implicit. Obviously, the C type and the `static_cast` which is what I will recommend. So, this is one way of so if you are able to edit or able to you know put code in the target type. Here A is the target type. If you are able to edit that, then the way to give a conversion from any source type is to provide a constructor for it.

Obviously, these constructors have to be single parameter constructor because it is to be used for the cast operator. If it has more than one parameter, then the cast operator will not be able to do that, quite obviously, because the other parameters will not be known to the cast operator. So, here you see a very nice thing that using the static_cast operator, you can actually control the casting logic depending on how you define these constructors.

(Refer Slide Time: 20:18)

```

#include <iostream>
using namespace std;

// Unrelated Types
class B;
class A { int i; public:
};
class B { public:
};
int main() { A a; B b; int i = 6;

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

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

#include <iostream>
using namespace std;

// Unrelated Types
class B;
class A { int i; public:
A(int i = 0) : i(i) { cout << "A::A(i)\n"; }
operator int() { cout << "A::operator int()\n"; return i;
};
class B { public:
operator A() { cout << "B::operator A()\n"; return A(); }
};
int main() { A a; B b; int i = 6;

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

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

```

Let us look at this example in a different way. First part of this; again, A and B are the same unrelated classes. i is from B to A which we have already I mean discussed it in a certain way. In the second part, what we want to look at is A being converted to int. Now, when you convert into A, you know how to do it. By providing a constructor which takes an integer gives you an A. What you do if you have to convert from A to int? You cannot provide a constructor in int type which takes A object and gives you an integer. You do not have that option. It is a built-in type.

Similarly, if you have a situation that where you are not allowed to edit the target type. B to A, how did we do? We just added a constructor in A which takes B object as a parameter which means I need to edit the class A. Now, if I am not allowed to do that, how do I do it? So, there are two situations. In one case it is not possible to edit the target class and in the other case, we are assuming that we are not allowed to do that. So, it boils down to the same thing that if you are not allowed to edit the target class, how do you do that? The way of the path of doing constructors, putting constructors does not work.

(Refer Slide Time: 22:00)

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

// Un-related Types
class B;
class A { int i; public:
};
class B { public:
};
int main() { A a; B b; int i = 5;

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

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

#include <iostream>
using namespace std;

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

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

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

This is where C++ has given another new, I mean another novel operator called the casting operator wherein in class B we write operator A. The name of the operator is A itself. You can see there is no return type. Why there is no return type? The operator A will operate on a B object and will return you an A type object. That is the purpose.

So, you do not need to specify any return type because the return type has to be an A object. So, this kind of user-defined cast operator written like operator, the keyword followed by the target type, then you just have a parenthesis; no parameter. This is the fixed style will be used by the static cast or for casting.

So, now given this if you do if you want to do this conversion, even the implicit conversion would be allowed because it knows that you want to take a B object to an A object. So, what are the choices? One choice is the class A has a constructor which takes B as a parameter. It does not exist here. So, that part is ruled out. Second it checks does Class B, has Class B provided A as a target type to be converted. The first one constructor path is based on the source type. This is based on the target type.

So, given B, it has given an operator, user-defined operator for the target type A. So, the types match and it will invoke that operated. Implicitly it will happen. By static_cast it will happen. By C-style casting it will happen. We will again always recommend that you use the static_cast on this.

(Refer Slide Time: 24:16)

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

// Un-related Types
class B;
class A { int i; public:
};
class B { public:
};
int main() { A a; B b; int i = 5;

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

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

#include <iostream>
using namespace std;

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

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

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

```

Similarly, if we want to convert say A to int. Since you cannot do it based on the target type, you have to do it based on the source type. So, on A you provide a casting operator, operator int where the target type is int. So, he will, if you provide that, then this conversion will be allowed in all of these cases. Again, you do not need to give a return type because it is known that the return type has to be an integer.

So, you may know note that in both of these operator cases the return will always have to return the object of the target type. So, here if you have to; now here I have just directly shown that as if the this constructor is getting called. You can have a whole lot of logic to construct that object but the end of it, it has to return an object of type A. Here it has to return an integer because you are converting to int.

So, two pathways. Now, if both are available for a conversion between two classes, two types, then obviously, the compiler will get confused and the compiler will not know which one to use and you will have error. So, you have to choose which one you want to do and write the expression in a different form.

(Refer Slide Time: 25:45)

reinterpret_cast Operator

- `reinterpret_cast` converts any pointer type to any other pointer type, even of unrelated classes ✓
- The operation result is a simple binary copy of the value from one pointer to the other
- All pointer conversions are allowed: neither the content pointed nor the pointer type itself is checked
- It can also cast pointers to or from integer types ✓
- The format in which this integer value represents a pointer is platform-specific
- The only guarantee is that a pointer cast to an integer type large enough to fully contain it (such as `intptr_t`), is guaranteed to be able to be cast back to a valid pointer (Refer to Module 26)
- The conversions that can be performed by `reinterpret_cast` but not by `static_cast` are low-level operations based on reinterpreting the binary representations of the types, which on most cases results in code which is system-specific, and thus non-portable

Programming in Modern C++ Partha Prasad Das 30/11/17

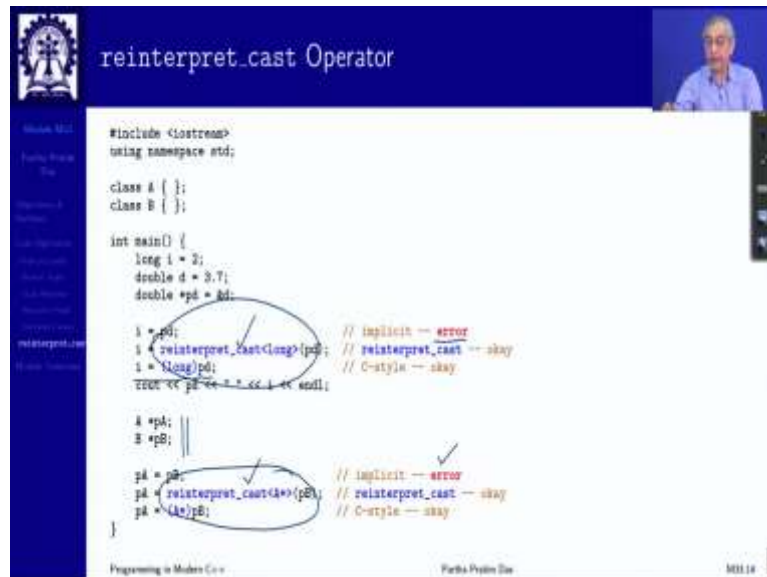
Next, we look at the `reinterpret_cast`. We already mentioned that `reinterpret_cast` is a very special type of cast operator which is given where `static_cast` cannot work. So, `reinterpret_cast` basically works in two contexts. One is to convert one pointer type to another where there is no relationship at all or between a pointer type and an integer type. So, these are the two contexts in which `reinterpret_cast` will work. Unlike `static_cast` where you can see that actually, computations are being done, functions are getting called, you can implement logic internally, `reinterpret_cast` is not like that.

`Reinterpret_cast` simply takes the binary copy of the source value and thinks it is the target value. It is just that. So, it just erases the source type, puts the target type right. So, all pointer conversions are allowed in terms of `reinterpret_cast`. But it does not take cognizance of any of the pointer content or any checking on the pointer type or anything of that.

Naturally, given this since it is at that low binary level, the format in which say integers will represent a pointer particularly if you are doing between integer and pointer is platform-specific. We have already seen examples in Module 26 that I could convert between a pointer and long but could not convert between pointer and int.

But if I have a machine where the pointer and int have the same size, I will be able to do that. And that is what is being emphasized by this point that there is a type defined in the library called `intptr_t` which gives you the size of the integer that must be provided to fit the pointer type. So, for our earlier case in Module 26, this size was 8 bytes. So, these conversions can be performed only by the `reinterpret_cast` not by the `static_cast` at all.

(Refer Slide Time: 28:24)



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

class A { };
class B { };

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

    i = *pd; // implicit -- error
    i = reinterpret_cast<long>(pd); // reinterpret_cast -- okay
    i = (long)pd; // C-style -- okay
    return 0;
}

A *pA;
B *pB;

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

Even as you can do them using C-style casting. But never never never do that because that is going to be very very risky. And you because this is a C-style casting like this one taking a pointer to an integer is a significant deviation in terms of the type system. And therefore, it is very important that it stands out boldly in your code.

So, the way to do that will be to use `reinterpret_cast`. Naturally the implicit casting does not work. Similarly, for two unrelated type pointers, the implicit cast does not work but `reinterpret_cast` will work. So, these are the two contexts where you should use `reinterpret_cast`.

And a word of caution. A lot of survey has been done on community code basis to check do we really need `reinterpret_cast`? The answer is no. It is usually, this is the only case where if you are doing that kind of programming of serialization-deserializing stuff like is the only case where you might need `reinterpret_cast`.

So, maybe in a very, very huge program, there may be one instance of `reinterpret_cast`. But if usually this will never be needed if you have done your design correctly. So, if you are requiring `reinterpret_cast` often, then ask yourself there must be something wrong in the design. It is not good to erase types and put a different type to fool the compiler because in turn compiler is your greatest friend in programming. So, if you fool the compiler, you are actually fooling yourself. So, avoid using `reinterpret_cast` as much as possible. And always question if you are using it and make sure that you are design and the context are correct to use that.

(Refer Slide Time: 30:34)



The image shows a presentation slide titled "Module Summary" with a dark blue header. In the top right corner, there is a small video feed of a man in a light blue shirt. The main content area is white and contains a single bullet point: "Studied `static_cast`, and `reinterpret_cast` with examples". On the left side, there is a vertical navigation menu with several items, including "Module Summary" which is highlighted. At the bottom of the slide, there is a footer with the text "Programming in Modern C++" on the left, "Part 6: Polymorphism" in the center, and "M11.15" on the right.

So, in this module, we have studied `static_cast` with all its different nuances. Particularly we have taken a look at user-defined cast operators in terms of constructor, single parameter constructor as well as user-defined cast operator-operator type. And we have seen how `reinterpret_cast` can be used in specific contexts of type erasing and redefinition. Thank you very much for your attention. We will meet in the next module.