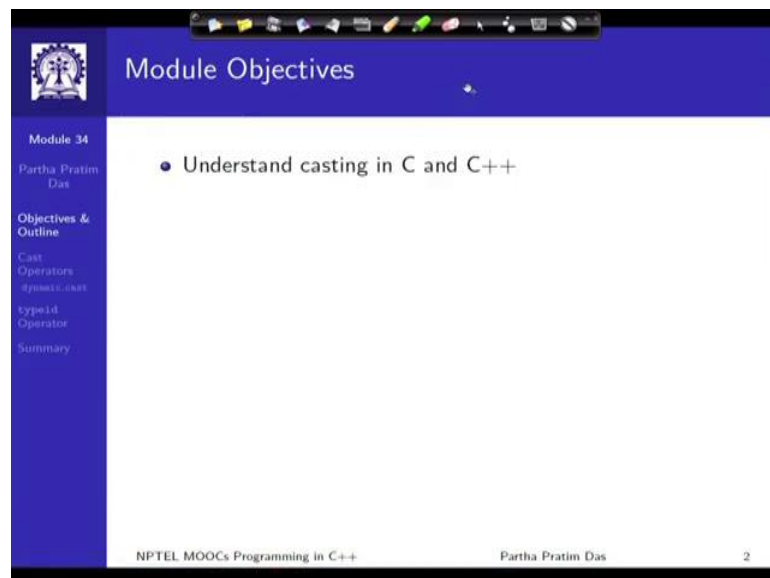


Programming in C++
Prof. Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

Lecture – 49
Type Casting and Cast Operators: Part – III

Welcome to Module 34 of programming in C++. We have been discussing about Type Casting and Cast Operators in C++.

(Refer Slide Time: 00:26)



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 34', 'Partha Pratim Das', 'Objectives & Outline', 'Cast Operators', 'typeid Operator', 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, our objective continued to be understanding casting in C and C++.

(Refer Slide Time: 00:37)

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 3

As I had mentioned two modules before that this is our total outline of discussions on casting. And the blue once here the dynamic cast operator and the typeid operator is what we discussed. In the current module, we have already discussed the basic premise of casting and the three cast operators const cast, static cast, and reinterpret cast operators in C++. Our discussion on cast operators and casting will conclude with this module.

(Refer Slide Time: 01:06)

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 in source
 - C-Style cast must be avoided in C++

NPTEL MOOCs Programming in C++ Partha Pratim Das

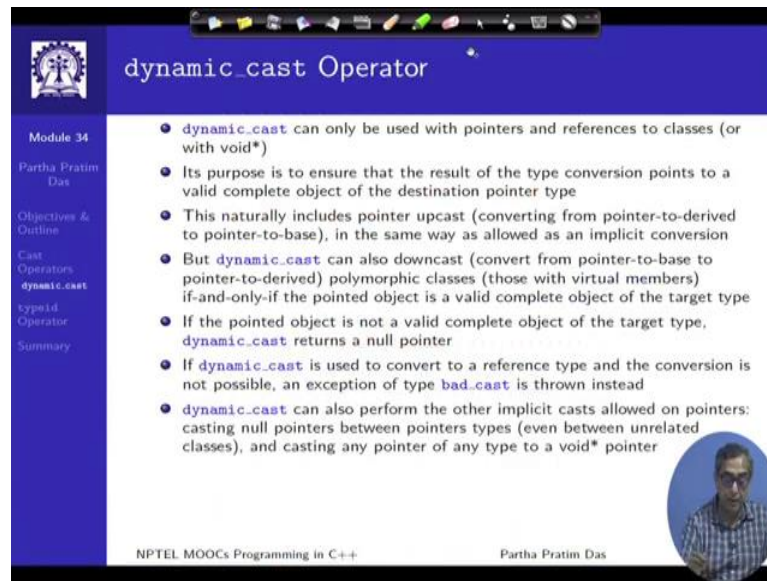
So, this is what we had seen that C has implicit casting, and explicit style of casting. And these are the lacunae in C casting; and based on this, the cast operators are the saviors that we have. Now if we look into the cast operators, this cast operator has told given us the ability to change the c v qualification.

So, the consciousness volatility view of the objects can be manipulated through const cast, particularly by using const or non-const reference to non-const or const objects or pointers constants and so on. Static cast takes care of all different kinds of implicit cast that C allows, you can explicitly write them using static cast. It has allowed us to actually do up cast as well as downcast on a hierarchy. It has also allowed us to cast between unrelated classes by using user defined constructor or conversion operators.

This is the widest form of static cast is a widest form of cast which will be used. And static cast is specifically so named, because it can do the whole inference of the casting at the compile time it does everything at the compile time. So, does const cast, but you specifically talk of static cast, as static because it is a wide variety of casting that you can do, but all of that you do based on the compile time.

Reinterpret cast as we saw is basically looking at the data through the classes of a different type and it is something which can be used to cast a pointer of any type to a pointer of another type or to cast between a pointer type and an integer type. And we have reason that we would normally avoid using the reinterpret cast.

(Refer Slide Time: 03:30)



The slide is titled "dynamic_cast Operator" and is part of Module 34, "Cast Operators", specifically focusing on "dynamic_cast". It lists several key points about the operator:

- `dynamic_cast` can only be used with pointers and references to classes (or with `void*`)
- Its purpose is to ensure that the result of the type conversion points to a valid complete object of the destination pointer type
- This naturally includes pointer upcast (converting from pointer-to-derived to pointer-to-base), in the same way as allowed as an implicit conversion
- But `dynamic_cast` can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if-and-only-if the pointed object is a valid complete object of the target type
- If the pointed object is not a valid complete object of the target type, `dynamic_cast` returns a null pointer
- If `dynamic_cast` is used to convert to a reference type and the conversion is not possible, an exception of type `bad_cast` is thrown instead
- `dynamic_cast` can also perform the other implicit casts allowed on pointers: casting null pointers between pointers types (even between unrelated classes), and casting any pointer of any type to a `void*` pointer

The slide also includes a small portrait of the presenter, Partha Pratim Das, and the NPTEL MOOCs logo.

In this context, the dynamic cast is a very unique one. Dynamic cast is the only cast operator which is based on the run time behavior of the program, which is actually based on the objects that can be that will exist at the run time. Dynamic cast does not work with objects; it is used only with pointers and references to classes. It does not work with object directly; it has to either have a pointer that you can cast or it has to have a reference that you can cast.

The purpose is to ensure that the result of type conversion points to a valid complete object of the destination pointer type; this is probably not still making much sense. So, it will come up when we go through examples, but basic point is that if the dynamic cast presumption is that is a context where you bring in dynamic cast is you have polymorphic hierarchy. It is not defined, if you do not have a hierarchy; and it is ill defined on a non-polymorphic hierarchy, it has hardly any use its use is always on a polymorphic hierarchy.

So, on a hierarchy we can up cast, for which also you can use dynamic cast, but what is important is you can downcast using dynamic cast and downcast with the run time information. So, that you can actually guaranty correctness of converting a pointer to the base class to a pointer to the derived class, and really know after this conversion as to

whether you are pointing to a valid object or you are not pointing to a valid object.

Now what happens is if this conversion is valid that after you have come from the base class pointer to a derived class pointer, you are actually pointing to the derived class object through this pointer. Then your dynamic cast operator gives you the same value of the pointer as of the base class pointer, because certainly the address of the object cannot change. But if this conversion is not valid that is if when you have downcast, you are not actually having a derived class object, for which you want to point to then the dynamic cast will set the pointer to null, so that by checking if the pointer is null, you can figure out if it is properly downcast or not. Dynamic cast can also be used with reference type we will see the example.

(Refer Slide Time: 06:03)

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

class A { public: virtual "A() () };
class B: public A { };
class C { public: virtual "C() () };

int main() {
    A a; B b; C c; A *pA; B *pB; C *pC; void *pV;

    pB = &b; pA = dynamic_cast<A*>(pB);
    cout << pB << " casts to " << pA << " : Up-cast: Valid" << endl;

    pA = &b; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << " : Down-cast: Valid" << endl;

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << " : Down-cast: Invalid" << endl;

    pA = (&a)&c; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << " : Unrelated-cast: Invalid" << endl;

    pA = 0; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << " : Unrelated-cast: Valid for null" << endl;

    pA = &a; pV = dynamic_cast<void*>(pA);
    cout << pA << " casts to " << pV << " : Cast-to-void: Valid" << endl;

    //pA = dynamic_cast<A*>(pV); // error: 'void *': invalid expression type for dynamic_cast

    return 0;
}
```

Output:

```
00E9FCAB casts to 00E9FCAB: Up-cast: Valid
00E9FCAB casts to 00E9FCAB: Down-cast: Valid
00E9FCB4 casts to 00000000: Down-cast: Invalid
00E9FC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00E9FCB4 casts to 00E9FCB4: Cast-to-void: Valid
```

Hand-drawn diagrams on the slide show a hierarchy where A is the base class, B is a derived class, and C is an unrelated class. Arrows indicate the direction of casting: A to B (Up-cast), B to A (Down-cast), and A to C (Unrelated-cast). A diagram also shows pointer pA pointing to object a, pB pointing to object b, and pC pointing to null.

Let us just go ahead and start taking some example. So, what I we will do is I have three classes here; class A is a base class; class B is specialized from class A. So, I have a polymorphic hierarchy. Class C is a third class, which is not related. So, we will try to illustrate what happens if you deal with unrelated class by using class C. Point to note on this hierarchy is A has a virtual destructor, which means that the hierarchy is polymorphic. This hierarchy is polymorphic, because it has a virtual function; therefore, the whole hierarchy is polymorphic. I construct three objects; I have three pointers of

three types, and pv is a void type pointer.

Now, let us try to start doing some tricks. So, this is A here, and B is A. So, if I say p B contains the address of B. So, this is a B object, and p B is a base, B type pointer. So, it is I have a B type pointer, I have pointer here, and that is what holds the object. Now if I dynamic cast p B to A star that is the pointer to A type object then what am I doing, what is the direction my direction is upwards, I am basically doing an up cast.

So, there should not be any problem in terms of this So, if I do that and keep it in a pointer of a class then I have done an Up-cast, so it should be valid. So, the result of this is if I after that I print p B and I print p A, this is a value of p B casts to p A, you can see the values are same that is basically the address of the B object. This is Up-cast and this is valid, first case, so that was straight forward. This is just like a base line check showing that the dynamic cast will able to cast Up-cast.

(Refer Slide Time: 08:41)

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

class A { public: virtual ~A() {} };
class B : public A { };
class C { public: virtual ~C() {} };

int main() {
    A a; B b; C c; A *pA; B *pB; C *pC; void *pV;

    pB = &b; pA = dynamic_cast<A*>(pB);
    cout << pB << " casts to " << pA << " : Up-cast: Valid" << endl;

    pA = &b; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << " : Down-cast: Valid" << endl;

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << " : Down-cast: Invalid" << endl;

    pA = (A*)&c; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << " : Unrelated-cast: Invalid" << endl;

    pA = 0; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << " : Unrelated-cast: Valid for null" << endl;

    pA = &a; pV = dynamic_cast<void*>(pA);
    cout << pA << " casts to " << pV << " : Cast-to-void: Valid" << endl;

    //pA = dynamic_cast<A*>(pV); // error: 'void *': invalid expression type for dynamic_cast

    return 0;
}
```

Output:

```
00EFFC48 casts to 00EFFC48: Up-cast: Valid
00EFFC48 casts to 00EFFC48: Down-cast: Valid
00EFFC44 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFC44 casts to 00EFFC44: Cast-to-void: Valid
```

Diagram illustrating pointer relationships:

- Object A (address 00EFFC48) is pointed to by pA (address 00EFFC48).
- Object B (address 00EFFC44) is pointed to by pB (address 00EFFC44).
- Object C (address 00EFFC9C) is pointed to by pC (address 00EFFC9C).
- Object A is also pointed to by pB (address 00EFFC44).
- Object A is pointed to by pV (address 00000000).

Let us now try to downcast. So, p A has address of B. So, I have a address of b and I have a b object, and I have a p A which is A type pointer, and this holds this object here. This is possible because this is possible so Up-cast. So, I can always use a base class pointer to hold a derived class object. I do a dynamic cast of p A, here into B star into of

this type. So, I am doing a dynamic cast from of p A in this direction; it was of A type and now I am trying to take it to B star. So, I am doing a downcast; I am pulling it down. And the resultant I keep in p B, which is a B type pointer. What is the result?

So, after that if I print p A and p B, I find p A and p B have the same values, and the downcast is valid. What is meant by the downcast is valid, because the after I have converted p B now points to this object, p B is a B type of pointer and this object is of b type, so there is no violation there is everything is in place. So, we say this downcast is a valid downcast earlier I had a type pointer and a b type object, I was holding it there. So, from p B from looking at p A, I could not have said whether the object is a b type object or is a type object. But now I have done this conversion, so I know that it is a it is an object where a valid b type object can be found. I print that all these are equal, the last part of the story.

(Refer Slide Time: 10:53)

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

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

class A { public: virtual ~A() {} };
class B : public A { };
class C { public: virtual ~C() {} };

int main() {
    A a; B b; C c; A *pA; B *pB; C *pC; void *pV;

    pB = &b; pA = dynamic_cast<A*>(pB);
    cout << pB << " casts to " << pA << " : Up-cast: Valid" << endl;

    pA = &b; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << " : Down-cast: Valid" << endl;

    pA = &a; pB = dynamic_cast<B*>(pA);
    cout << pA << " casts to " << pB << " : Down-cast: Invalid" << endl;

    pA = (A*)&c; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << " : Unrelated-cast: Invalid" << endl;

    pA = 0; pC = dynamic_cast<C*>(pA);
    cout << pA << " casts to " << pC << " : Unrelated-cast: Valid for null" << endl;

    pA = &a; pV = dynamic_cast<void*>(pA);
    cout << pA << " casts to " << pV << " : Cast-to-void: Valid" << endl;

    //pA = dynamic_cast<A*>(pV); // error: 'void *': invalid expression type for dynamic_cast

    return 0;
}
```

Output:

```
00EFFC68 casts to 00EFFC68: Up-cast: Valid
00EFFC68 casts to 00EFFC68: Down-cast: Valid
00EFFC64 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFC64 casts to 00EFFC64: Cast-to-void: Valid
```

Hand-drawn diagrams on the slide illustrate pointer relationships:

- A box labeled 'A' has an arrow pointing to 'pA'.
- A box labeled 'B' has an arrow pointing to 'pB'.
- A box labeled 'C' has an arrow pointing to 'pC'.
- A box labeled 'pA' has an arrow pointing to 'pB'.
- A box labeled 'pB' has an arrow pointing to 'pA'.
- A box labeled 'pA' has an arrow pointing to 'pC'.
- A box labeled 'pC' has an arrow pointing to 'pA'.
- A box labeled 'pA' has an arrow pointing to 'pV'.

Let us do the same thing, look at here, let us do the same thing, but now let me start with p A again, but now unlike earlier case, where p A was holding the address of a b type object here now it is holding the address of a type object. So, this is your p A is, it holds the address of a type object. I again try to do the same dynamic cast as I had done here exactly the same expression. So, p A being, so I am again doing a downcast, I am again

bringing it down and I will have p B now try to point to this A. So, I do this downcast here and put that pointer value to p v.

See what is the result is the third line is p A, which is address of A. And what is p B, p B is 0 – null. Why p B is 0, because it is dangerous if I can do this pointing. Because if I can do this pointing then the fact that this pointer is a p B that is of B type, I will expect that what it points to is at least a b type object, but it is not b type object it is generalized then that. So, b type object may have this much it actually has only the base part of it. So, that if you do I mean this is the basic difference between dynamic cast and static cast.

So, in this context here instead of dynamic cast, if I had done a static cast of the same expression b star and then p A, the result will be the value of p A. Because if I am doing statically, I do not know what p A is pointing to, but now that this is a dynamic cast this will able to figure out as to whether it is actually B object in that case it gives copies the value of p A to p B. Or if it is A object in which case it does not copy this value, it is a 0 to this.

So, after the dynamic cast, so and you have you have seen that in these two cases, in this case you have seen this is a second line in this case you have seen this is a third line in this case, you get a copy of that address; in this case, you get a null value. So, by simply checking if p B is null or not you know whether the dynamic cast has gone through or whether actually the pointed object is a b type object or not, so that is a basic value of the dynamic cast.

And rest of it simply rests on that, for example, I could use I can again take p A take the C type of object, and keep it is address in p A certainly these two are unrelated. Since these two are unrelated, I cannot actually put the address of the C object into p A therefore, I have forced through a C style casting there is only way I can put this address and then I tried to do this conversion of p A to C star. Certainly this conversion is not possible, so what I get is a value of this is a fourth line, what I get is a value of after the conversion into p C is a null value. So, if it is unrelated then dynamic cast will always reject. These are the special case if p A is 0, and then I try to do this cast, it will the null value null pointers, it will always convert to null pointers.

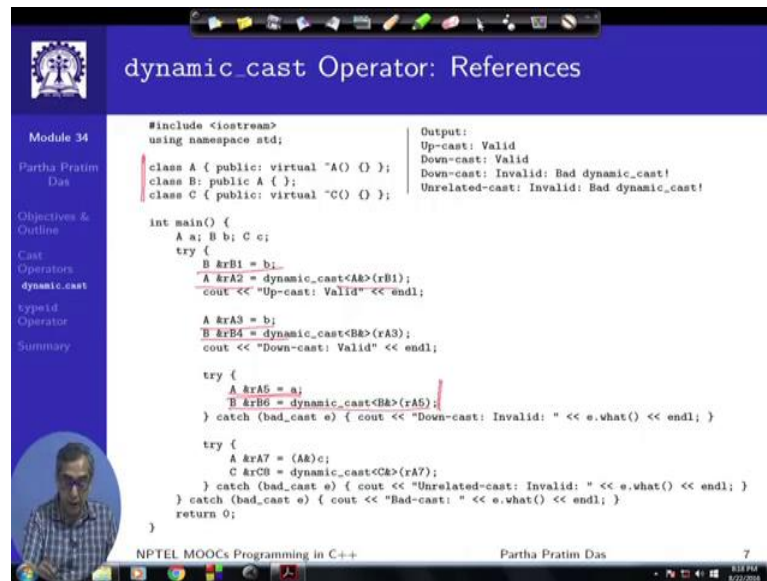
Look at this, if p A carries the address of A object, and I try to do a dynamic cast used to void star that is the void star, and this is the my void star pointer then this is a permitted cast this is a last line you can see values are same. Because it is allowed to convert from a pointer type to void, but if you try to do the reverse here you can see, I try to do the reverse, I take try to take that p v and bring it back to A star, this is compilation error.

Now in the process of doing this, in the process of doing this, certainly you will wonder as to how dynamic cast is figuring this (Refer Time: 15:37) is very simple, is the restriction is the argument to the dynamic cast the source expression to the dynamic cast must be a pointer to a polymorphic hierarchy. What is a characteristic of polymorphic hierarchy, it is a virtual function. What is a consequence of having a virtual function, it has a virtual function pointer in that object and that pointer is specific to the type of the class as we have seen.

If I have object of a different type, I have a different virtual function table, and I have a different pointer value, but all objects of the same class same polymorphic class have the same virtual function table, and therefore, the same function pointer value. So, it basically internally take looks at this, checks the corresponding, what is the virtual function pointer table of that object. And from that it knows that what is the type, it is dealing with.

And therefore, it can always say that whether that true dynamic type matches the static type into which you are trying to bring that address. And for that reason expressions like this cannot be expression like this that is doing a dynamic cast on p V is not possible, because void star does not is not a polymorphic type. And since void star is not a polymorphic type there is no virtual function pointer table in that. So, the dynamic cast itself, the operator itself cannot overcome this that is the reason you get invalid expression type for dynamic cast that is the basic. So, that is about the dynamic cast which can cast your pointers on a polymorphic hierarchy at the run time.

(Refer Slide Time: 17:30)



The slide displays a C++ program demonstrating the use of the `dynamic_cast` operator with references. The code defines three classes: `A`, `B`, and `C`. `A` is the base class, `B` inherits from `A`, and `C` inherits from `B`. The `main` function creates objects `a`, `b`, and `c`. It then performs several dynamic casts:

- `A& rA1 = b;` (Valid up-cast)
- `A& rA2 = dynamic_cast<A&>(rB1);` (Valid down-cast)
- `B& rB1 = b;` (Valid down-cast)
- `B& rB2 = dynamic_cast<B&>(rA3);` (Valid down-cast)
- `A& rA3 = b;` (Valid up-cast)
- `B& rB4 = dynamic_cast<B&>(rA3);` (Valid down-cast)
- `A& rA5 = a;` (Valid up-cast)
- `B& rB6 = dynamic_cast<B&>(rA5);` (Invalid down-cast, caught by `catch (bad_cast e)`)
- `A& rA7 = (A&)c;` (Valid up-cast)
- `C& rC8 = dynamic_cast<C&>(rA7);` (Invalid down-cast, caught by `catch (bad_cast e)`)

The output of the program is:

```
Output:
Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: Bad dynamic_cast!
Unrelated-cast: Invalid: Bad dynamic_cast!
```

The slide also includes a sidebar with navigation options and a small video inset of the presenter, Partha Pratim Das.

You can use that similarly with reference. So, the same hierarchy the whole everything same is such that you could now use references, and use the dynamic cast in the same way. So, this is an example of doing the up-cast, this is the example of doing the down-cast which is valid. Now, what happens if you try to do this; that you are actually having a reference of the base class object and you are trying to downcast it to the specialized class.

(Refer Slide Time: 18:25)

dynamic_cast Operator: References

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

class A { public: virtual ~A() {} };
class B: public A { };
class C { public: virtual ~C() {} };

int main() {
    A a; B b; C c;
    try {
        B &rB1 = b;
        A &rA2 = dynamic_cast<A*>(rB1);
        cout << "Up-cast: Valid" << endl;

        A &rA3 = b;
        B &rB4 = dynamic_cast<B*>(rA3);
        cout << "Down-cast: Valid" << endl;

        try {
            A &rA5 = a;
            B &rB6 = dynamic_cast<B*>(rA5);
        } catch (bad_cast e) { cout << "Down-cast: Invalid: " << e.what() << endl; }

        try {
            A &rA7 = (A)c;
            C &rC8 = dynamic_cast<C*>(rA7);
        } catch (bad_cast e) { cout << "Unrelated-cast: Invalid: " << e.what() << endl; }
        catch (bad_cast e) { cout << "Bad-cast: " << e.what() << endl; }
        return 0;
    }
}
```

Output:

```
Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: Bad dynamic_cast!
Unrelated-cast: Invalid: Bad dynamic_cast!
```

Module 34
Partha Pratim Das
Objectives & Outline
Cast Operators
dynamic_cast
typeid Operator
Summary

NPTEL MOOCs Programming in C++ Partha Pratim Das 7

Now what happened when you what happened when you ah when we try to do this in the earlier case when we tried to do something similar here then the resultant was that the pointer turned out to be null. Now if I am doing dealing with references of objects then certainly null reference is not possible in C++. So, you cannot have a. So, this in this case you cannot have this is an invalid case, because you cannot have a reference because that object is not of the B type it is of the generalized A type. So, this conversion reference cannot be done.

What it does? It is what is known as it throws a bad cast exception, now we have not yet discussed about exception. So, it is one way of saying that is something has gone wrong. So, when we do exceptions next you will understand how what exactly is happening, but all that it is doing it is same that something has gone wrong and we cannot proceed further. And it comes out and prints this separate message which you can see here. Similarly, here I have tried to do a casting of the reference based on the unrelated class C which also should be invalid. So, it gives you unrelated class here. So, this is the way to use the dynamic cast operator.

(Refer Slide Time: 19:29)

typeid Operator

- `typeid` operator is used where the **dynamic type** of a polymorphic object must be known and for static type identification
- `typeid` operator can be applied on a type or an expression
- `typeid` operator returns `const std::type_info`. The major members are:
 - `operator==`, `operator!=`: checks whether the objects refer to the same type
 - `name`: implementation-defined name of the type
- `typeid` operator works for polymorphic type only (as it uses RTTI – virtual function table)
- If the polymorphic object is bad, the `typeid` throws `bad_typeid` exception

NPTEL MOOCs Programming in C++ Partha Pratim Das 8

Along with the dynamic cast operator another operator which is available is known as a `typeid` operator which simply tries to find out that. For example, if you have a class A or something I mean whatever that class A is and you have a variable of type A or rather let me just do it you have a pointer of this type `p A`. And of `p A` could be pointing to something, you do not know, what it is pointing to; it could be pointing to anything.

Now the question is, is it possible that from this pointer value I can say what is the type of pointed object, what is the type of the object that is pointing to. It would be great if we can say this is dynamic type, if we can say what is the type that, because it is a pointers which can be of own static type, but can point to number of different types.

So, `typeid` operator is tries to find out the dynamic type of a polymorphic object, which can exist in different forms. It should be able to compare the dynamic type of two such objects and say if there are of the same type, there of the different types. So, this is useful in that way and it can again be used only with polymorphic types not with non-polymorphic types that are you cannot use it on a non-polymorphic hierarchy or things like that.

(Refer Slide Time: 21:13)

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

// Polymorphic Hierarchy
class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A a;
    cout << typeid(a).name() << " : " << typeid(&a).name() << endl; // Static
    A *p = &a;
    cout << typeid(p).name() << " : " << typeid(*p).name() << endl; // Dynamic

    B b;
    cout << typeid(b).name() << " : " << typeid(&b).name() << endl; // Static
    p = &b;
    cout << typeid(p).name() << " : " << typeid(*p).name() << endl; // Dynamic

    A &r1 = a; A &r2 = b;
    cout << typeid(r1).name() << " : " << typeid(r2).name() << endl;

    return 0;
}

-----
class A: class A *
class A *: class A
class B: class B *
class A *: class B
class A: class B
```

So, it works something like this. This is simple hierarchy B is A, this is what you have. You have an A object and you apply typeid on that. If you apply typeid, it gives you a structure back it is called a type info structure. So, the typing for structure as I showed has different fields; and the most useful of that is a name field, which it gives some in some form the name of the class not the way we write it, but in some form.

(Refer Slide Time: 22:20)

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

// Polymorphic Hierarchy
class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A a;
    cout << typeid(a).name() << " : " << typeid(&a).name() << endl; // Static
    A *p = &a;
    cout << typeid(p).name() << " : " << typeid(*p).name() << endl; // Dynamic

    B b;
    cout << typeid(b).name() << " : " << typeid(&b).name() << endl; // Static
    p = &b;
    cout << typeid(p).name() << " : " << typeid(*p).name() << endl; // Dynamic

    A &r1 = a; A &r2 = b;
    cout << typeid(r1).name() << " : " << typeid(r2).name() << endl;

    return 0;
}

-----
class A: class A *
class A *: class A
class B: class B *
class A *: class B
class A: class B
```

So, typeid a dot name will give the name of the type. Similarly, if I do typeid ampersand a dot name it will give the name of that type. So, if I do this it prints class A, for this class A, for this it prints class A star, so that is basically the typeid output. If I take this pointer to this address, and print the pointer or the pointed object I get A star, I get class A. Simple not a problem nothing interesting as such, it is just taking the object type and the pointer type pointer type and the object type. Think about you have an object B, you do the same thing here print the typeid on the object and typeid on it is address, you get class B and class B star.

Interesting thing starts happening here you assign the address of B to p, p is of type A. Again in this assignment itself, I have an up-cast because p is of type A, and this is a B type address, you have up-cast. So, you are holding the B object using A type pointer. Now try to do typeid p, typeid p is it is static type which is A star because it is A type. Typeid star p, what is a type of star p what the type of object it is pointing to that is class p, it is a dynamic type. So, type id is a dynamic type operator it tells you what is the dynamic type of that. So, earlier you did this while p was pointing to an A object, you got class A. Now it is pointing to a B type object you do here you get class.

You can do the same thing with reference, you have a reference to A, and you have another reference of type A to object B. You just look at this, this is one reference of type A to object A. Another reference of type a, to object b if you print r 1, it is class A; if you print the typeid of r 2, it is class B. Because the reference is, but the reference is of type a, but what does it print typeid what does it give, it gives you the typeid of the object it is referring to which is type B. So, that is basic typeid operator.

In some cases, you could use this operator you can actually compare do not one word of caution is do not take this strings very strictly, because compilers do not guarantee that my compiler is giving telling this class A, your compiler might write it in a very different form, it might be you write it in a different case or in some other notation. So, do not take these names very strictly, what you should do is rather use the equality and inequality operator that the type info class offers, so that you can check if two objects are of the same type or they are of different types.

(Refer Slide Time: 25:23)

The slide shows a C++ program demonstrating a polymorphic hierarchy. It defines three classes: `Engineer`, `Manager`, and `Director`, all inheriting from `Engineer`. The `ProcessSalary` method is overridden in each class. The `main` function creates objects of these classes and stores them in an array of `Engineer*` pointers. The `typeid` operator is used to print the type of the pointer and the object it points to. The output shows that pointers are always of type `Engineer*`, while the objects are of their respective class types.

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

class Engineer { protected: string name_;
public: Engineer(const string& name) : name_(name) {}
virtual void ProcessSalary() { cout << name_ << " : Process Salary for Engineer" << endl; }
};

class Manager : public Engineer { Engineer *reports_[10];
public: Manager(const string& name) : Engineer(name) {}
void ProcessSalary() { cout << name_ << " : Process Salary for Manager" << endl; }
};

class Director : public Manager { Manager *reports_[10];
public: Director(const string& name) : Manager(name) {}
void ProcessSalary() { cout << name_ << " : Process Salary for Director" << endl; }
};

int main() {
    Engineer e("Rohit"); Manager m("Kanala"); Director d("Ranjana");
    Engineer *staff[] = { &e, &m, &d };
    for (int i = 0; i < sizeof(staff) / sizeof(Engineer*); ++i) {
        cout << typeid(staff[i]).name() << " : " << typeid(*staff[i]).name() << endl;
    }

    return 0;
}

-----
class Engineer *: class Engineer
class Engineer *: class Manager
class Engineer *: class Director
```

So, this is for your workout, this is our staff salary application in the polymorphic hierarchy. I have just tried to print the typeids of the pointer type and pointed object type. So, you can say the pointers are always engineer star because our original array was of engineer star, but pointed objects are of different types as we are pointing to. You can read through this again, and understand it better.

(Refer Slide Time: 25:57)

The slide shows a C++ program demonstrating a non-polymorphic hierarchy. It defines two classes: `X` and `Y`, where `Y` inherits from `X`. The `main` function creates objects of these classes and stores them in arrays of `X*` pointers. The `typeid` operator is used to print the type of the pointer and the object it points to. The output shows that pointers are always of type `X*`, and the objects are of their respective class types. A comment indicates that a dynamic cast from `X*` to `Y*` fails.

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

// Non-Polymorphic Hierarchy
class X {};
class Y : public X {};

int main() {
    X x;
    cout << typeid(x).name() << " : " << typeid(&x).name() << endl; // Static
    X *q = &x;
    cout << typeid(q).name() << " : " << typeid(*q).name() << endl; // Dynamic

    Y y;
    cout << typeid(y).name() << " : " << typeid(&y).name() << endl; // Static
    q = &y;
    cout << typeid(q).name() << " : " << typeid(*q).name() << endl; // Dynamic -- FAILS

    X &r1 = x; X &r2 = y;
    cout << typeid(r1).name() << " : " << typeid(r2).name() << endl;

    return 0;
}

-----
class X: class X *
class X *: class X
class Y: class Y *
class X *: class X
class X: class X
```

For a change, if you look into a case, where we have non-polymorphic types again I have a hierarchy, but the only difference is non-polymorphic in the sense that there is no virtual function in this. So, what is the consequence, the consequence is if the no virtual functions then the objects do not have a virtual function pointer to the table. So, they are now it is now not possible at the run time to say, what is the class from where that object actually came. So, we can demonstrate that here. So, we see x object and it is address class x x star.

Again through this, through a pointer we set a pointer class x, another object y class, y y star. Now, you say q on this class x x star. So, basically now you do not you cannot track that you have actually a different dynamic object residing there because all that you get is static information alone. So, similar thing will get if you do the references as well. So, if the hierarchy is non-polymorphic, then the typeid mechanism will simply not work.

(Refer Slide Time: 27:26)

The slide content is as follows:

```

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

class A { public: virtual ~A() {} };
class B : public A {};

int main() {
    A *pA = new A;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    delete pA;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    pA = 0;
    try {
        cout << typeid(pA).name() << endl;
        cout << typeid(*pA).name() << endl;
    } catch (const bad_typeid& e) { cout << "caught " << e.what() << endl; }

    return 0;
}
    
```

Output:

```

class A *
class A *
class A *
caught Access violation - no RTTI data!
class A *
caught Attempted a typeid of NULL pointer!
    
```

The slide also includes a sidebar with 'Module 34', 'Partha Pratim Das', and 'typeid Operator'.

There are some more cases where you could try this, for example, again this is a polymorphic hierarchy; I am just I will just illustrate something interesting. So, I have created a new object of the base type. I have done the typeids, these are the two outputs, and it is fine, nothing new. I have deleted this object through p A, so it does not exist. I try to typeid again or in the pointer, I get A star which is valid because this is static type.

I tried to do typeid of star p A; I get a message caught access violation no RTTI data.

See normally, what you will expect is that that if we try to do this is the system will crash, but here that you will not do that. It will give you certain bad typeid exception. I will tell you that you are trying to find typeid for something which cannot be found, and on that bad typeid there is some message that. So, if you can actually use this on a on a polymorphic hierarchy, you can actually used to find out if a pointer is dangling or it is pointing to a valid object. So, this gives you lot of other additional you know advantages also

Last I just make this pointer null. I just made p a as 0; assigned it to 0. Again I tried to find out the type this is the static type which is fine. I tried to do star p A, I got an exception attempted a typeid of null pointer. So, the typeid computation on a polymorphic hierarchy will always tell you about what is the actual dynamic type, what is a actual dynamic status of the object that is being pointed to.

Of course, I mean as a practice I would not advice that you use the typeid operator very frequently in terms of your program, for the simple reason that since it uses the virtual function table and all that information in next to make use of what is known as RTTI - run time type information, which needs to regularly track what are where the virtual function tables are and what their addresses and so on.

So, usually it is quite a lot of you know time taking to compute this at the run time. So, it is much better to be able to manage with just the virtual functions themselves which are pretty efficient than to explicitly use dynamic cast or certainly the typeid operator. But you do have them in your armory, so that as and when required if you are in the extreme case you could use them and make a good benefit out of that.

(Refer Slide Time: 30:46)

The image shows a presentation slide with a blue header and a white main area. The header contains the text 'Module Summary' and a small logo on the left. The main area contains a bulleted list of three items. The footer contains the text 'NPTEL MOOCs Programming in C++', 'Partha Pratim Das', and '13'. The slide is presented in a window with a standard operating system toolbar at the top.

Module Summary

- Understood casting at run-time
- Studied dynamic cast with examples
- Understood RTTI and typeid operator

NPTEL MOOCs Programming in C++ Partha Pratim Das 13

To summarize, we have understood casting at run time in this module. And we have started studied dynamic cast as a last type of cast operator and associated with it. We have seen that in the presence of RTTI, the typeid operator can actually find out the dynamic type of a pointer an object and if that is on a polymorphic hierarchy. So, with this module, we conclude our discussions on type casting and cast operators in C++. And we have shown that they are four operators of const cast, static cast, reinterpret cast and dynamic cast are sufficient to solve any of your cast requirements.

And with that you should minimize on, if actually I would advice that you completely remove any use of C style casting in your code, just rely on these cast operators. Primarily, rely on the static cast and const cast operator, and dynamic cast when you are on a polymorphic hierarchy.