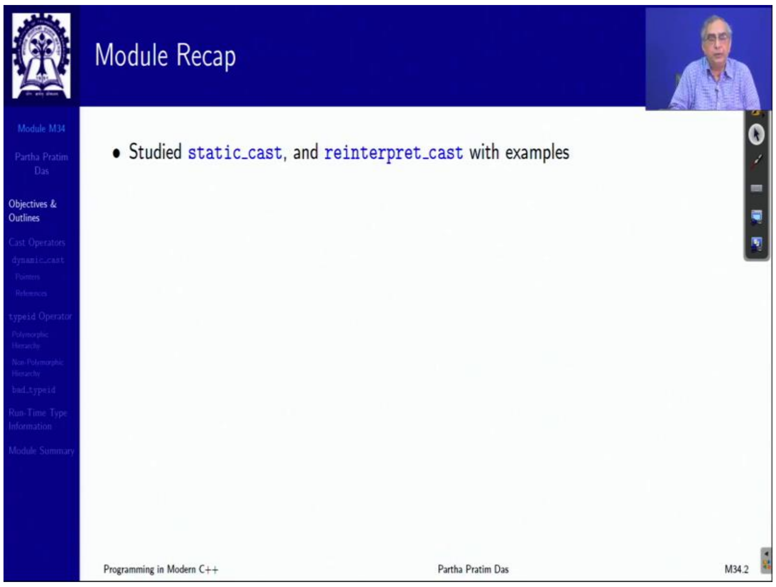
Programming in Modern C++
Professor Partha Pratim Das
Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur
Lecture – 34
Type Casting & Cast Operators: Part 3

(Refer Slide Time: 00:32)



Module Recap

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

Module Summary

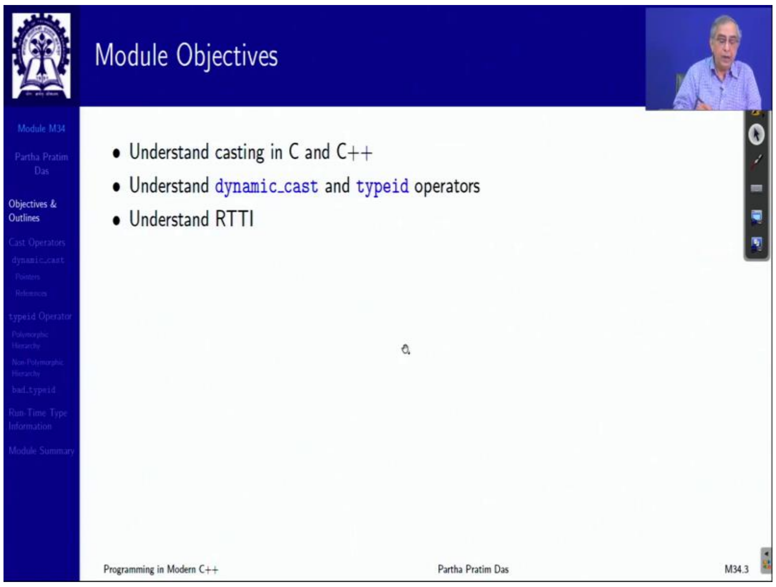
- Studied static_cast, and reinterpret_cast with examples

Programming in Modern C++ Partha Pratim Das M34.2

Welcome to Programming in Modern C++, we are in week 7 and we are going to discuss module 34. In the last two modules, we have been discussing about cast operators in C++ and how to use different cast operators in different semantic context. So, in the last module specifically we have discussed about static_cast which can be used in place of any implicit casting conversion on or also additionally for upcast also for some of the reverse conversions which are not implicitly allowed, risky but can be used for down casting as well and particularly is useful to invoke user defined cast operations in terms of constructor or explicitly written cast operator.

And we have also studied about reinterpret_cast which is basically reinterpreting a value which in simple terms mean that erase the type on the source and put the type of the target, not recommended not advised possibly the only context you use is when you have to convert a pointer to an integral type of the reverse.

(Refer Slide Time: 1:50)



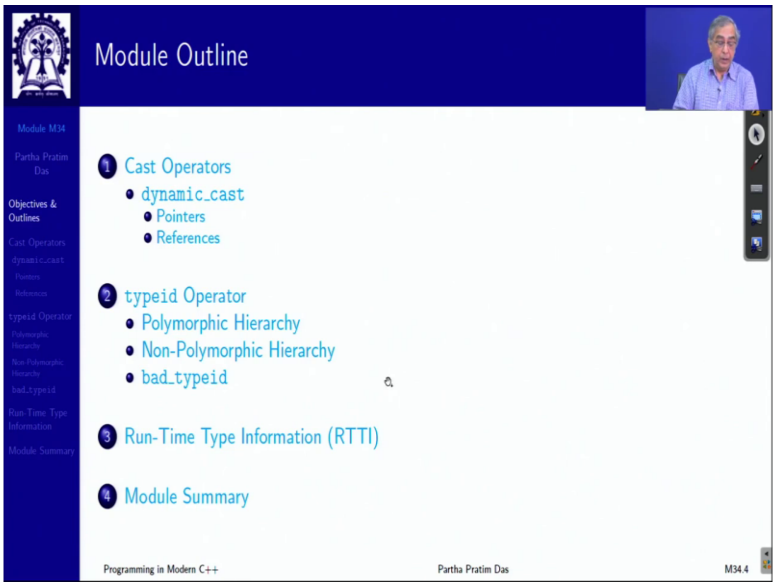
Module Objectives

- Understand casting in C and C++
- Understand dynamic_cast and typeid operators
- Understand RTTI

Programming in Modern C++ Partha Pratim Das M34.3

In the present module, we are going to continue on this discussion and understand the dynamic_cast and typeid operator. Dynamic_cast as we will see is a run time casting operator which is very very unique and different for the other three that we have discussed. So, in this context we will also take a look at what is known as RTTI, Run Time Type Information.

(Refer Slide Time: 02:22)



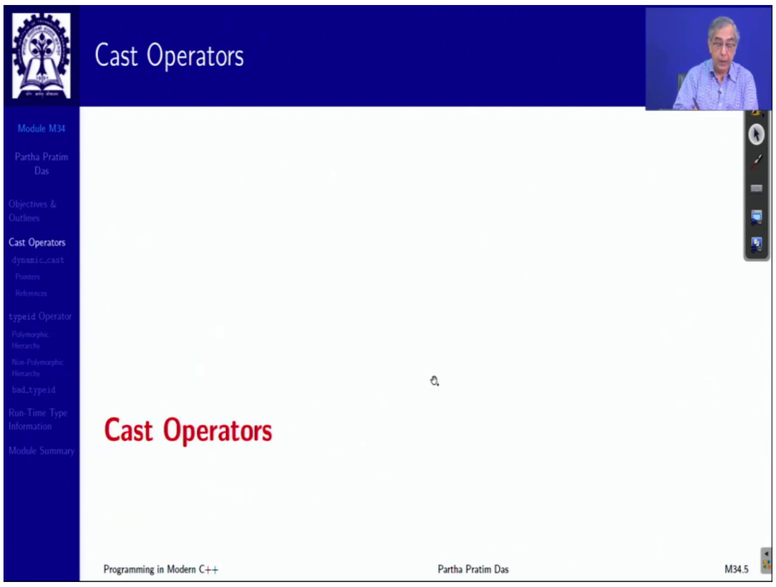
Module Outline

1. Cast Operators
   - dynamic_cast
     - Pointers
     - References
2. typeid Operator
   - Polymorphic Hierarchy
   - Non-Polymorphic Hierarchy
   - bad_typeid
3. Run-Time Type Information (RTTI)
4. Module Summary

Programming in Modern C++ Partha Pratim Das M34.4

This is the outline which will be available on your left.

(Refer Slide Time: 02:26)



Cast Operators

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

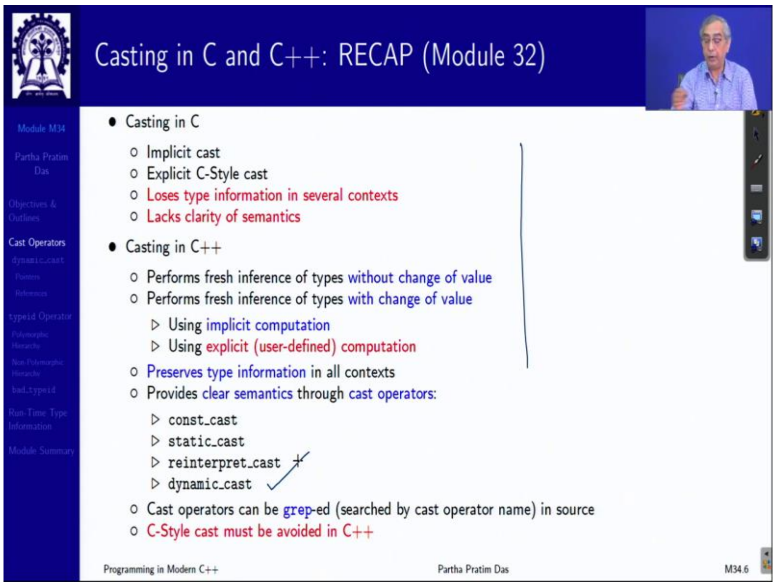
bad_typeid

Run-Time Type Information

Module Summary

Cast Operators

Programming in Modern C++ Partha Pratim Das M34.5



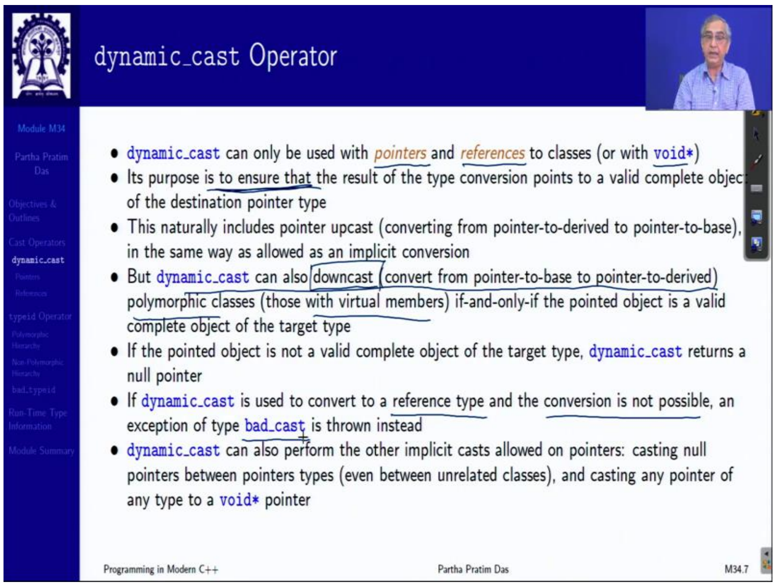
Casting in C and C++: RECAP (Module 32)

- 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++

Programming in Modern C++ Partha Pratim Das M34.6

Quick recap from module 32 of what we have been doing. We have gone over this number of times. We are now here to talk about the dynamic_casting operator.

(Refer Slide Time: 2:38)



dynamic_cast 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 objec 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

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

Module Summary

Programming in Modern C++ Partha Pratim Das M34.7

So, what is dynamic_casting? Let us start with a few, let us define the playing field. It can be used only with pointers and references, dynamic_casting cannot be used on objects directly. You can use it only on pointers or on references everything else is a compilation error. It can be used with void* also.

The purpose is to ensure that when you dynamically cast one pointer to another, the source pointer is pointing to an object, it has a type. The destination pointer, target pointer will continue to point to that same object but with a different type. So, dynamic_cast ensures that this conversion gives you a valid object.

If I am, I was pointing by A type pointer and I dynamically cast it to a B type pointer, then dynamic_cast ensures that the pointer actually is pointing to an object which can behave as B type, that is, this guarantee is the most important. Naturally, this will include the conversion which are upcast, the upcast will always satisfy that though it is not advisable to use dynamic_cast for upcast operation because dynamic_cast is far more expensive, for upcast you always have the static_cast.

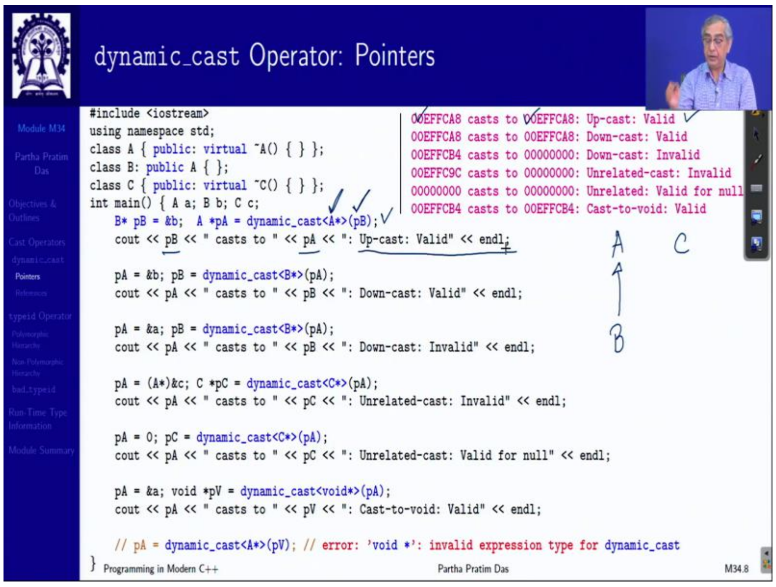
dynamic_cast can actually be useful or is the only context where it is useful, is when you down cast. When you down cast, you are coming from a generalized class to a specialized class. So, you do not know standing with a pointer of generalized class as to whether the object has enough information of the specialized class. If it does then this going down is valid, if it does not, then this going down is not valid this casting cannot be done.

This is valid for polymorphic classes, that is those with virtual member functions at least one virtual member function so which could be the destructor as we have seen and it is valid if and only if the pointed object is a valid complete object of the target type. So, that computation is the most important thing in the dynamic_cast. If it is not, then the dynamic_cast will return a null pointer.

Otherwise it will return the valid pointer value. It can be used similarly as I said with reference type and it gives you a reference of the new specialized class type and if it is disallowed, if the conversion is not possible because you do not have a correct object, then certainly since there is no concept of a null reference, it will through an exception. There is a system defined exception called badcast which is thrown.

We have not yet discussed about exception so maybe you will not find all of the code here easy to digest but you can come back to it after you have done exceptions in the next week. dynamic_cast can also perform other implicit casts on pointer for example, it can cast null pointers between pointed types. Casting any pointer of any type to void* and so on. So, these kinds of tasks can also be done with dynamic_casting.

(Refer Slide Time: 7:09)



dynamic_cast Operator: Pointers

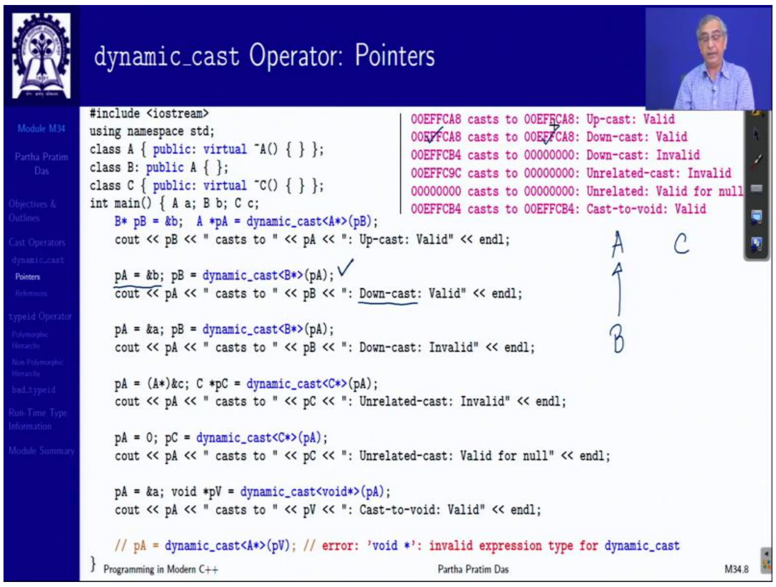
```
#include <iostream>
using namespace std;
class A { public: virtual ~A() { } };
class B: public A { };
class C { public: virtual ~C() { } };
int main() { A a; B b; C c;
    B* pB = &b;  A *pA = dynamic_cast<A*>(pB);
    cout << pB << " casts to " << pA << ": Up-cast: Valid" << endl;

    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; 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; void *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
}
```

```
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```

$$\begin{tikzcd} A & C \\ B \arrow[u] \end{tikzcd}$$

Module M34 | Partha Pratim Das | Objectives & Outlines | Cast Operators | dynamic_cast | Pointers | References | typeid Operator | Polymorphic Hierarchy | Non-Polymorphic Hierarchy | bad_typeid | Run-Time Type Information | Module Summary

Programming in Modern C++ | Partha Pratim Das | M34.8



dynamic_cast Operator: Pointers

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

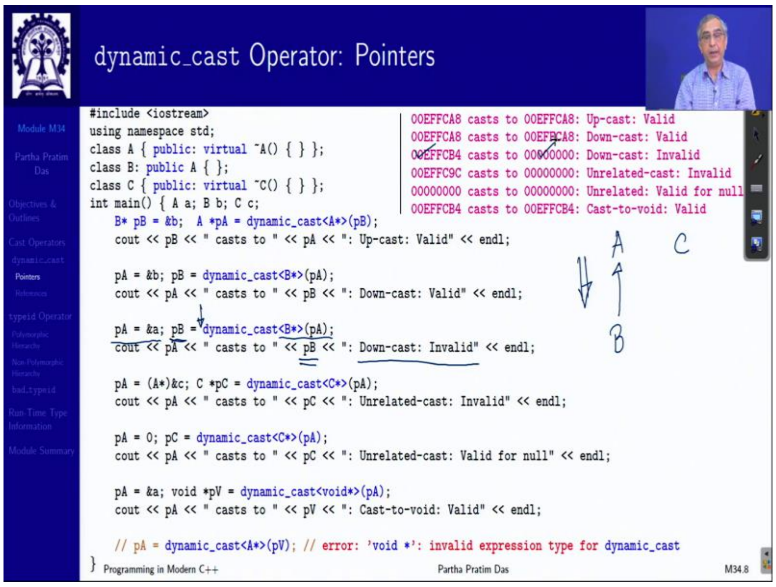
Run-Time Type Information

Module Summary

```
#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;
    B* pB = &b;  A *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; 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; void *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
}
```

```
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```

$$\begin{tikzcd} A & C \\ B \arrow[u] \end{tikzcd}$$



dynamic_cast Operator: Pointers

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

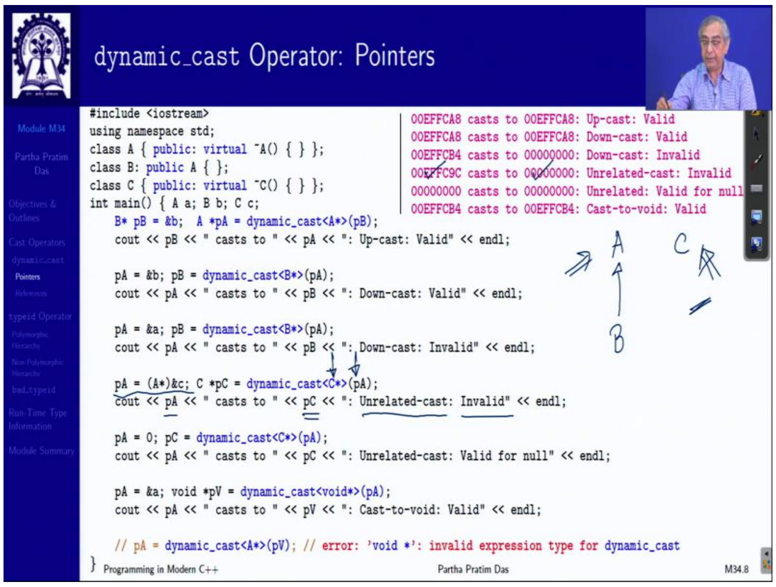
Run-Time Type Information

Module Summary

```
#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;
    B* pB = &b;  A *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; 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; void *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
}
```

```
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```

$$\begin{tikzcd} A & C \\ B \arrow[u] \end{tikzcd}$$



dynamic_cast Operator: Pointers

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

Module Summary

```
#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;
    B* pB = &b;  A *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; 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; void *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
}
```

```
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```

$$\begin{tikzcd} A & C \\ B \arrow[u] \end{tikzcd}$$

I think example is what will make things better. So, what do I have? I have a class A, specialized from this I have class B, class A, class B. Members and all that I am not considering because that is not important, it is just the type which is important. And, this is, all of the classes are polymorphic because I have a virtual destructor here and since B is a specialization of A, naturally it has a virtual distractor so these are all polymorphic and I have a class C, class C is not related to this. This context is important, so now let us think, let us say this is, I have a pointer pB which is holding the address of a B. I have kept the naming consistent so that you do not have to look up what is the type. If it is small b, it is a B object, if it is small c it is a C object and so on.

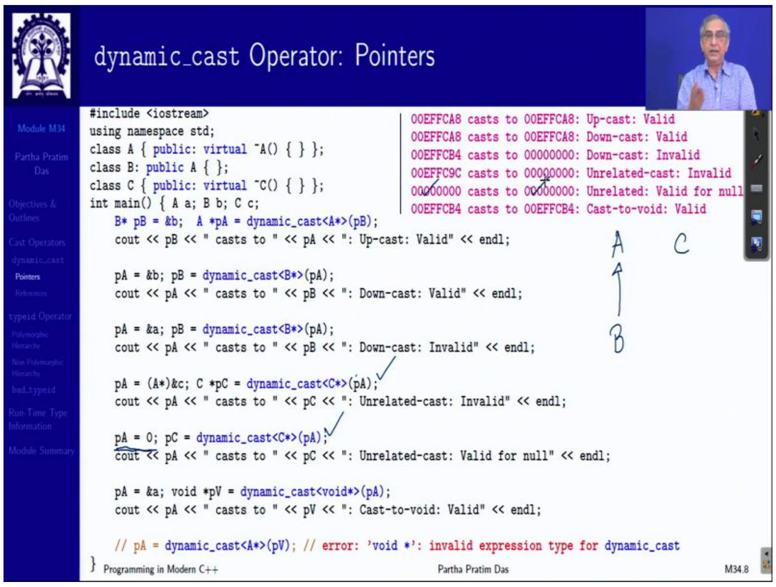
Now, I dynamically cast, now I am dynamically casting (this to) this pB to A*, it is a B type of pointer but it is actually holding a B type object and I want to see whether I can hold this object by A type pointer or in other words whether I can think of this object as an A type object which is basically the task of upcast. So, if I do that, it will be valid. So, here as we go with each cout, you can see the output written here. So, this is the first line of the output, this is your original pointer value in some machine very we ran and you get the same pointer. So, this was the value of pB, this is the value of pA and it prints upcasting valid.

So, first thing is done. I can do upcast, if I really need to. Second is, I hold, I have done an implicit upcast by taking the address of the B object and putting it as a type pointer. Now, I want to come back. So, the pointer type is A, the actual object it is pointing to is B and I want to see whether I can hold it as a B type pointer, only if it is actually a B type object which it is.

So, if I do this, this is the case of down cast and again you will see that I get the same pointer value the down cast is valid in this case. Now, this is in pA, I have A object and I want to downcast to B, come back, come from here to here down cast in B. So, if I try to hold it by pB, which is a B type pointer then I expect a B object but it actually is an A object which is a generalized object. It will be wrong, so now this down cast will be invalid. How do you see it?

This was your original pointer of A object and your result of downcast, the result that the downcast returns, the return value of down cast is zero, is a null pointer. This null pointer tells me that the down cast is not right. I do not have a correct, so by checking if the return pointer is null or not I can ascertain whether the down cast has correctly happened or it is not possible to down cast. So, this is the beauty if you had used static_cast here, you would have forced the casting without knowing that you are actually doing something wrong.

(Refer Slide Time: 11:54)



dynamic_cast Operator: Pointers

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

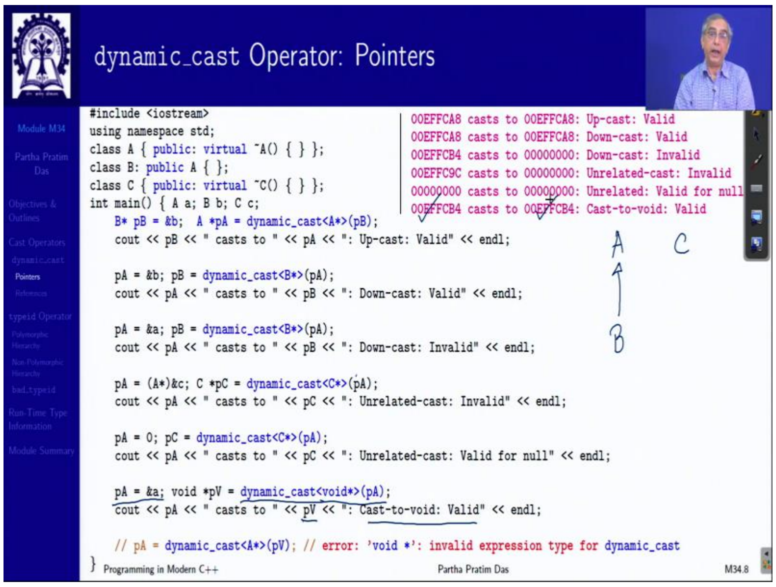
Module Summary

```
#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;
    B* pB = &b;  A *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; 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; void *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
}
```

```
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```

$$\begin{tikzcd} A & C \\ B \arrow[u] & \end{tikzcd}$$

Programming in Modern C++ Partha Pratim Das M34.8



dynamic_cast Operator: Pointers

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

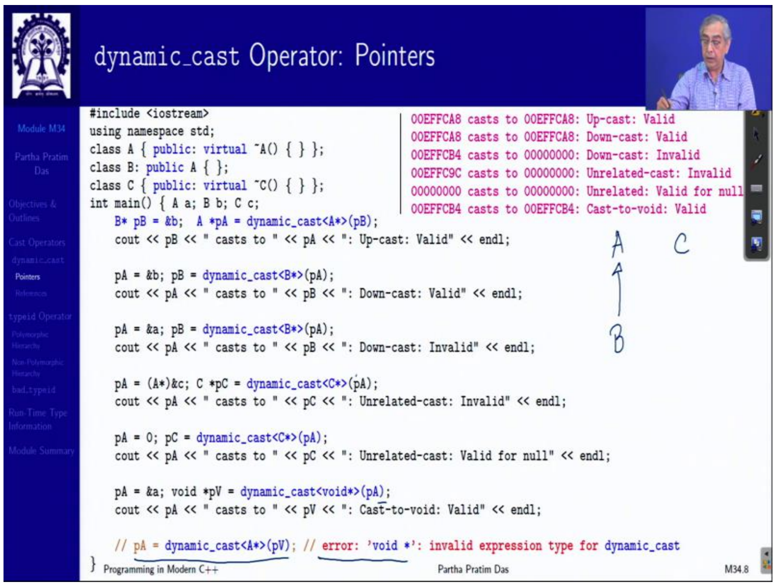
Module Summary

```
#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;
    B* pB = &b;  A *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; 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; void *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
}
```

```
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```

$$\begin{tikzcd} A & C \\ B \arrow[u] & \end{tikzcd}$$

Programming in Modern C++ Partha Pratim Das M34.8



dynamic_cast Operator: Pointers

```
#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;
    B* pB = &b;  A *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; 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; void *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
}
```

```
00EFFCA8 casts to 00EFFCA8: Up-cast: Valid
00EFFCA8 casts to 00EFFCA8: Down-cast: Valid
00EFFCB4 casts to 00000000: Down-cast: Invalid
00EFFC9C casts to 00000000: Unrelated-cast: Invalid
00000000 casts to 00000000: Unrelated: Valid for null
00EFFCB4 casts to 00EFFCB4: Cast-to-void: Valid
```

Programming in Modern C++ Partha Pratim Das M34.8

The third, the next example fourth is, I have taken an object of the unrelated class C and put it as a A pointer. Now, from this a pointer I want to check whether I have a C type object but they are unrelated as you can see. These are unrelated, A and C, these are unrelated classes.

Naturally there is no upcast, down cast nothing can be defined in these terms and the dynamic_cast operator will rightly identify and give a null pointer as returned. So, this is a case of unrelated caste which is invalid. So, pA was given, pC will be null. You can see the dynamic_cast is how powerful it is. It can really act and all these are happening at the run time because you have no way of knowing these things at the static time at the compile time.

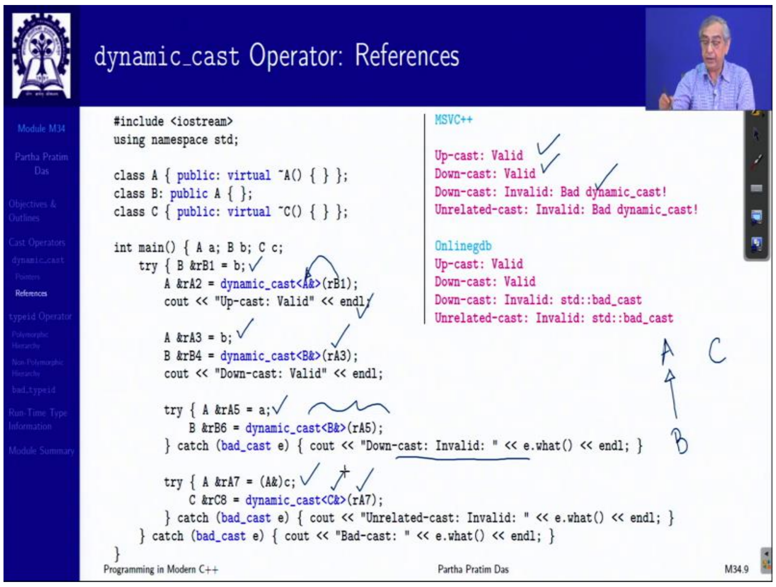
Now specifically, if the pointer is null and I am trying to, what I am doing here and what I am doing here is the same. Only difference is here the pointer is null, if the pointer is null then it is a valid cast. Though since the pointer originally is null, the dynamic_cast will return a null but this is a null on success. It is a valid cast for unrelated types as well. So, you can use that in building up the logic.

Now, let us take pA to point to an A object and use dynamic_cast to cast the pointer to void*, that is forget about the type which means forget about what that object might have. So, this is casting to void which is valid, so this will give you the same pointer back telling you that, well you have been able to cast it right and now you hold the object as a void type.

However, you cannot do the reverse for obvious reasons, if you have a void type pointer, I do not want to say, let me think that this is got any object, you cannot do that because it is given a void type, the dynamic_cast does not know how to find out the type information of that object. It has to do it at the runtime, so, it has no way of knowing that. Since it does not know that it cannot give you a conversion.

So, here it is not a question of error this is a question of an invalid expression. Because dynamic_cast cannot even operate on this. So, this is a static time compilation error, this is what you will get and you have this kind of, I mean something like invaded expression type for dynamic_cast. So, dynamic_cast can cast to void* but cannot cast from void*.

(Refer Slide Time: 15:36)



dynamic_cast Operator: References

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

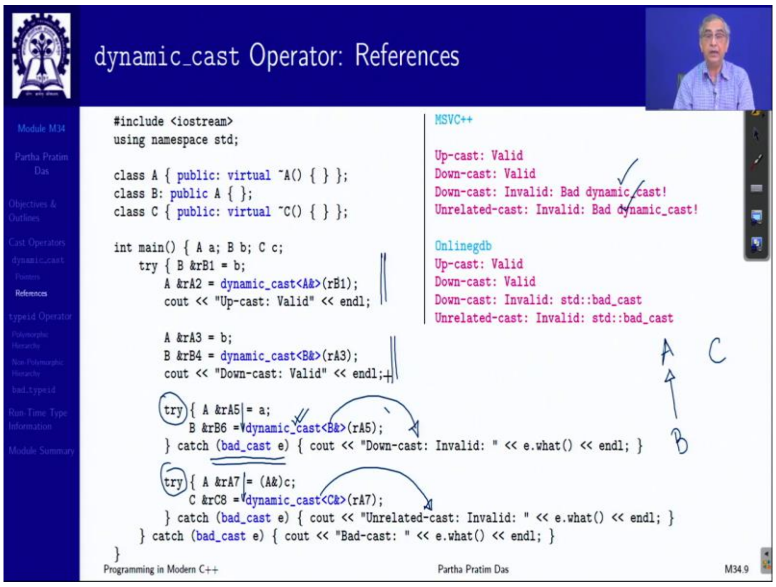
Module Summary

```
#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; }
}
```

```
MSVC++

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

Onlinegdb
Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: std::bad_cast
Unrelated-cast: Invalid: std::bad_cast
```

$$\begin{tikzcd} A & C \\ B \arrow[u] & \end{tikzcd}$$

Programming in Modern C++ Partha Pratim Das M34.9



dynamic_cast Operator: References

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

Module Summary

```
#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; }
}
```

```
MSVC++

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

Onlinegdb
Up-cast: Valid
Down-cast: Valid
Down-cast: Invalid: std::bad_cast
Unrelated-cast: Invalid: std::bad_cast
```

Programming in Modern C++ Partha Pratim Das M34.9

So, let us look at the same example but here instead of using pointer, we will be using reference. I said dynamic_cast works for both of them, so this is a reference I have also named the reference easily so that you can make out. So, rB1 is a reference to a B object and I used dynamic_cast to change, to create a reference to an A object. Obviously, B to A, upcast this is valid, valid.

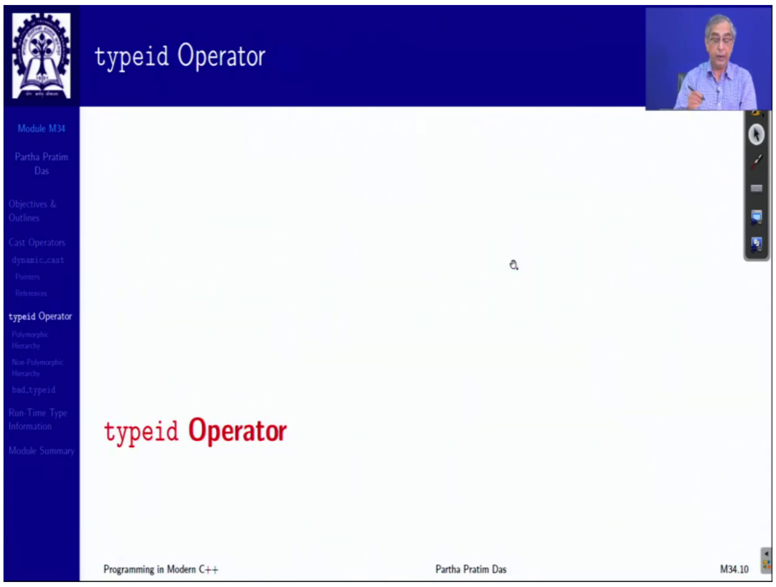
I take a B object and create a A type reference to it which is implicit upcast and then from that A type reference, I want to go back to the B type reference that is down cast. In this case, the object was valid so this also is valid. I have a A type reference to an A type object and I try to down cast it to a B type which is wrong because the actual object is of A type, the reference is of A.

The reference is actually to an A type object, so this is an invalid downcast. So, I get a message like bad dynamic_cast. Take the unrelated class, I take the unrelated class and kind of force its cast, its reference to A type because I have to force because they are unrelated and then I try to dynamically cast back to the C type which obviously is unrelated class, so, it is also something which cannot happen, which cannot be accepted.

So, therefore it also is an error. Now, the point to note in these two later cases where I have error, like in these two that since it has to return a reference here and it cannot return a null. So, how does it exemplify the failure? That is in terms of this what is called the try catch block, that you put the code within the try catch pair of curly braces and if this particular operator code has some error that it cannot continue and give me a reference then it will throw an exception known as bad cast which is caught by this mechanism.

So, this is the exception mechanism, we will talk in detail about it but that is how the flow is working here. So, it is it is coming from here to here because the error has happened. If the error did not happen then the code will simply go through. Actually, these also we should put under try catch, I just wanted to keep things simple because in these cases the exceptions will not be thrown because things are all ok.

(Refer Slide Time: 19:01)



typeid Operator

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

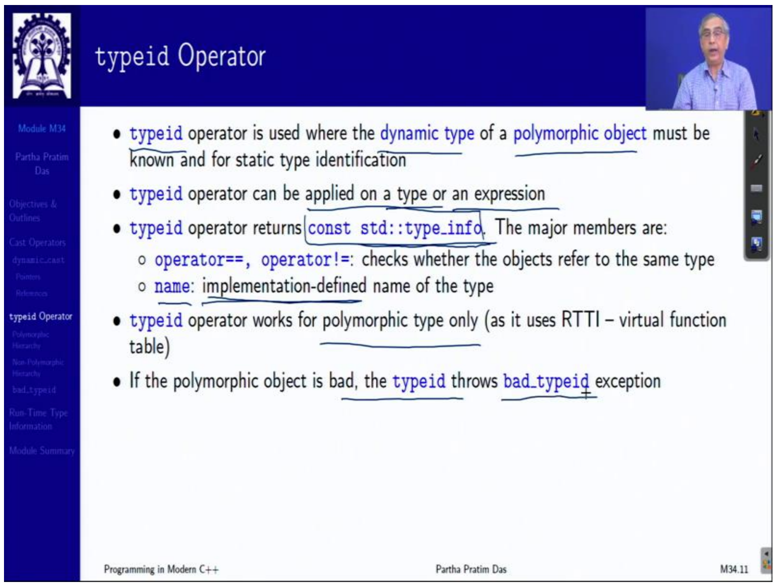
bad_typeid

Run-Time Type Information

Module Summary

typeid Operator

Programming in Modern C++ Partha Pratim Das M34.10


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

So, this is how to deal with ah down cast with references. So, we have seen how to do casting on a hierarchy particularly downcast using the down cast operator with either the pointer or the reference of an object. Now, in this relation, so what are you doing in the downcast? In the downcast we are certainly not only checking the type of the pointer or the type of the reference but we are checking the runtime type of the object, we have talked about static time type and run time type.

So, how do we check that runtime type? So, for that the system has to have a small runtime system which runs along with our code which is called the RTTI, Run Time Type Information. So, what the dynamic_cast does it actually when it gets the pointer or reference to an object, it consults this RTTI to check on a particular object corresponding to the given object which is called the type info.

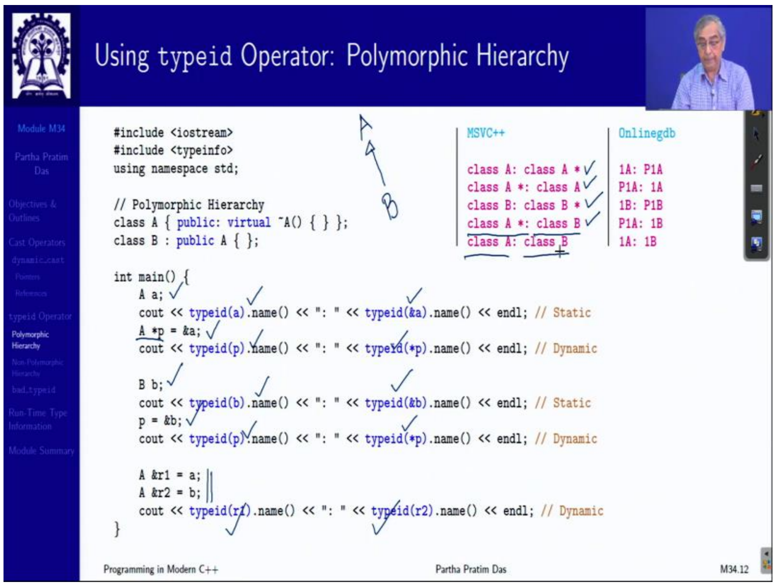
This type info has all the information of what type the runtime object is in. So, it is used only for the dynamic type of polymorphic object, that is the basic target. You know because if, it cannot be used for the static type because static type is compiler decided and static type information is not managed by the RTTI. So, there will be no information on that but is about the dynamic type where you have particularly polymorphic object is where you can get this typeid operator, put this operator to tell you in some way that what is the type.

So, it can be applied on a type or on an expression. So, this typeid operator actually gives you access to this type_info structure which as you can see is defined in the standard library under the std namespace. So, what you can do, you can basically compare to types, you can for equality and inequality naturally, less than greater than does not make sense.

And you can invoke a name member function which gives you a user understandable text representing the name. Now, the point to note is this is implementation defined that the standard has not specified how to name the types because actually the types are defined by the user and how do how the system will represent it is up to the compiler implementer.

So, if you are using typeids you have to look at the documentation to know how the types are named, so that you can correspond them to your source code. We will see examples. It works for polymorphic type only and if a polymorphic object is bad, it is not a proper type or something then it will again throw a bad typeid exception which is defined in the standard library.

(Refer Slide Time: 22:43)



Using typeid Operator: Polymorphic Hierarchy

Module M34 | Partha Pratim Das | Objectives & Outlines | Cast Operators | dynamic_cast | Pointers | References | typeid Operator | Polymorphic Hierarchy | Non-Polymorphic Hierarchy | bad_typeid | Run-Time Type Information | Module Summary

```
#include <iostream>
#include <typeinfo>
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; // Dynamic
}
```

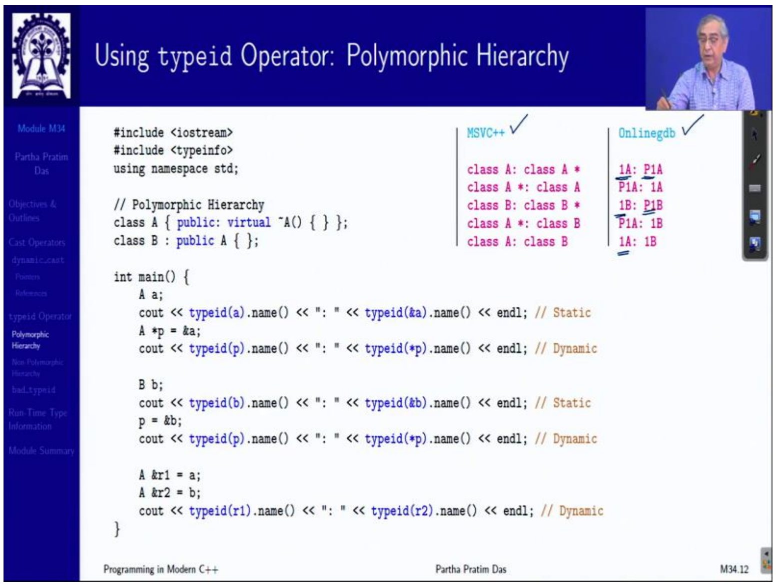
MSVC++	Onlinegdb
class A: class A *	1A: P1A
class A *: class A	P1A: 1A
class B: class B *	1B: P1B
class A *: class B	P1A: 1B
class A: class B	1A: 1B

A

A

B

Programming in Modern C++ | Partha Pratim Das | M34.12



Using typeid Operator: Polymorphic Hierarchy

```
#include <iostream>
#include <typeinfo>
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; // Dynamic
}
```

MSVC++	Onlinegdb
class A: class A *	1A: P1A
class A *: class A	P1A: 1A
class B: class B *	1B: P1B
class A *: class B	P1A: 1B
class A: class B	1A: 1B

Programming in Modern C++ | Partha Pratim Das | M34.12

So, first let us try to look at how to use the typeid operator on a polymorphic hierarchy. So, it is very simple, I have A and B specialized from that A has a virtual destructor so all objects of A and B are polymorphic. I create an A object, I ask what is the type of A and the pointer to A and these are the names that I get. It is class A and it is class A*.

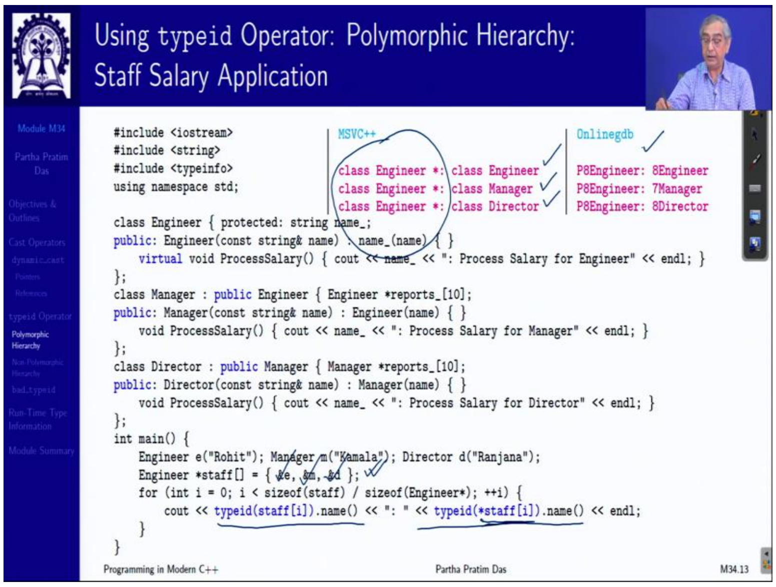
I create a pointer to the A object and ask what is the type of the pointer and what is the point type of the pointee. I get class A* class A. This is how you can actually. Now, most often you will not need to do this, your virtual functions, polymorphic dispatch and downcast operator will take care of all the runtime types but in case you want to go deeper and really want to find out what is the type you have and so on you can make use of this typeid operator. I would presume that normal application development programmers in C++ would never need to use this. But this is to give you more debugging and exploratory strength.

Now, I have a B object, I print B and the reference B and B*, I do an upcast, remember pointer a is of type A, I do you can see that the pointer type is A* that is how it is defined. I have done it upcast so the original object is of B type. You can get the entire picture here through this. This is with the references if you do this r1 and r2, r1 is a A object, r2 is actually reference of a B object but held as A type reference, you can see it rightly says that this is a A object and this is a B object.

So, this is how the typeid operator can be used and this is how you get. I have here, you can see that I have given two sets of outputs. This is what I found on my Microsoft Visual C++, compiler which is kind of more understandable and this is what I found on the online GDB. I said that its implementation dependent, its implementation defined.

So, for example you can see here that in every case the class name is just marked by 1 and pointer by p1 and then that given name in the source that comes up. So, that is, so here you have to check your documentation, the documentation of the system and know how to decipher these names.

(Refer Slide Time: 26:10)



Using typeid Operator: Polymorphic Hierarchy: Staff Salary Application

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

Module Summary

```
#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("Kamala"); 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;
    }
}
```

MSVC++

```
class Engineer *: class Engineer ✓
class Engineer *: class Manager ✓
class Engineer *: class Director ✓
```

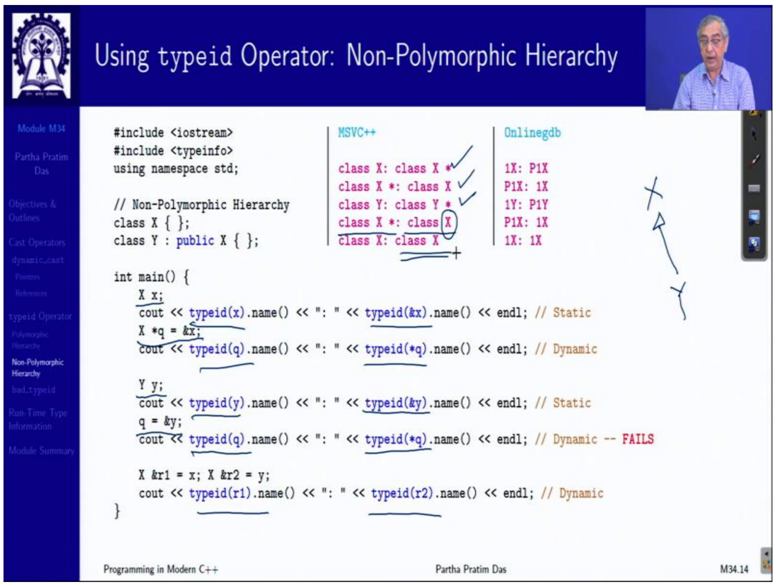
Onlinegdb ✓

```
P8Engineer: 8Engineer
P8Engineer: 7Manager
P8Engineer: 8Director
```

Programming in Modern C++ Partha Pratim Das M34.13

Next example our, staff salary code that we had. So, I just put in the staff salary, I put the typeid of the staff[i], staff[i] is an array of, is an element of the array of pointers each is of type Engineer. But they are pointing to different types of actual objects. So, if I dereference, I should get a different object, so that is what I get. For the pointer I always get Engineer but for the object I get specifically Engineer, Manager, Director as of port. This is how it comes in terms of the online GDB system, this is how they mark the type names. Good fun.

(Refer Slide Time: 27:02)



Using typeid Operator: Non-Polymorphic Hierarchy

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

bad_typeid

Run-Time Type Information

Module Summary

```
#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; // Dynamic
}
```

MSVC++

```
class X: class X * ✓
class X *: class X ✓
class Y: class Y * ✓
class X *: class X
class X: class X
```

Onlinegdb

```
1X: P1X
P1X: 1X
1Y: P1Y
P1X: 1X
1X: 1X
```

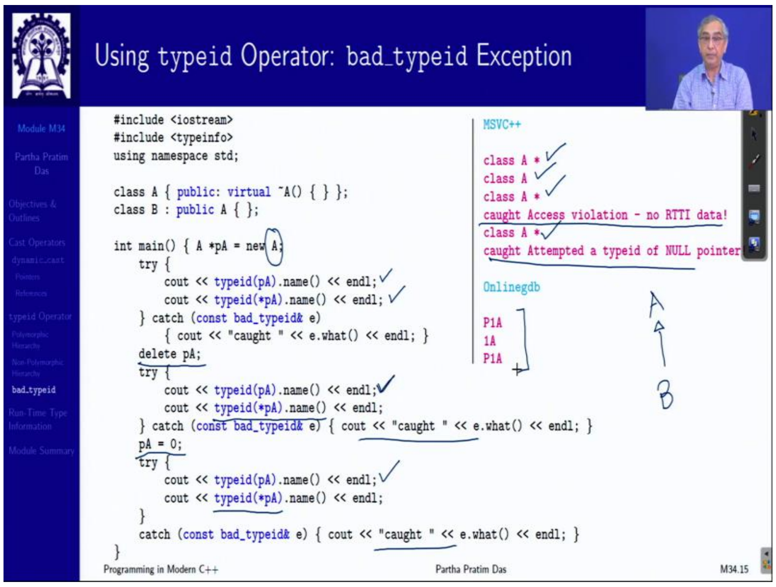
$$\begin{tikzcd} X \\ Y \arrow[u] \end{tikzcd}$$

Programming in Modern C++ Partha Pratim Das M34.14

Now, you can use it partly for non-polymorphic hierarchy also though it may not make much sense. For example, here I have X object, object and its pointer. I have X object, address of X object put to a X type pointer. I get this which is, I have Y object. So, this is a hierarchy but it is not polymorphic. The Y object, I do this, I get class Y. I do an upcast, Y object to the pointer of X object. So, I have done an upcast and I try to do this. I get X* and I get X object.

This is because it is non-polymorphic because as you know, if it is non-polymorphic then the compiler has to decide everything based on the static type. So, *q has a static type which is class X. Earlier when this was polymorphic then I would have got Y here because the runtime type is Y but this hierarchy is not polymorphic so I cannot get that. So, because of this the value of finding typeid on a non-polymorphic hierarchy is marginal. Similarly, you will see the same thing here in terms of the reference as well and different types of names for the online GDB system.

(Refer Slide Time: 28:59)



Using typeid Operator: bad_typeid Exception

```
#include <iostream>
#include <typeinfo>
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; }
}
```

MSVC++

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

Onlinegdb

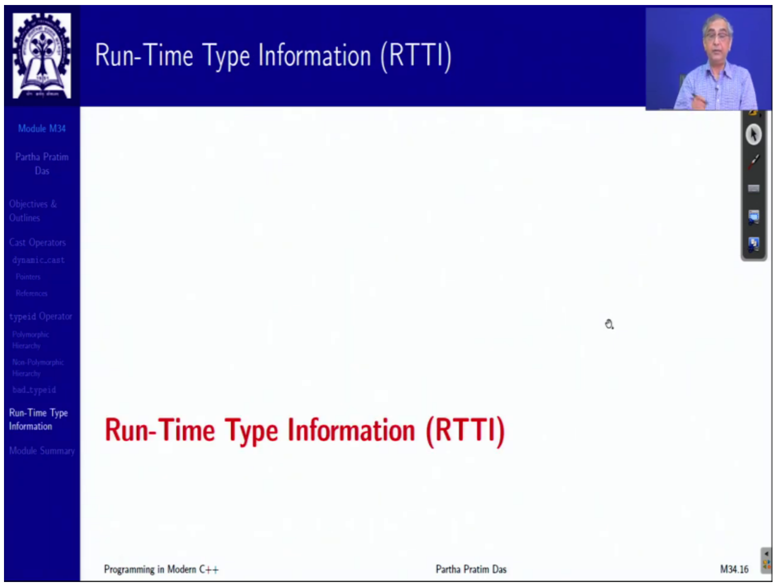
```
P1A
1A
P1A
```

Programming in Modern C++ Partha Pratim Das M34.15

So, if the typeid are bad then it shows an exception. So, this is fine, typeid of *pA again, the context is the same and this is a polymorphic hierarchy so if I do start typeid of *pA, I get this. If I, now have done this, I have deleted, I have deleted the object. So, I was holding the A object, I have deleted that. So, when I do typeid for pA, the pointer, I still get class A* which I showed but when I do *pA, this is an invalid axis because the object is already gone. So, I get this kind of a message on MSVC++, that access violation no RTTI data.

Now, I set pA to null, again the type of pA is correct but when I try to check the type for the typeid of *pA, I am doing a null pointer referencing. So, I get a corresponding error. So, this is what I caught by this exception. So, these are the cases, I mean some of the cases where bad typeid might get thrown because you are using the typeid on a bad object or a non-existing object and so on. Interestingly on online GDB you just get the correct values, correct results you do know nothing is printed for the errors that are produced it may be the settings of the GDB. I really did not check further on that but be careful, check on the documentation before you proceed.

(Refer Slide Time: 30:51)



Run-Time Type Information (RTTI)

Module M34

Partha Pratim Das

Objectives & Outlines

Cast Operators

dynamic_cast

Pointers

References

typeid Operator

Polymorphic Hierarchy

Non-Polymorphic Hierarchy

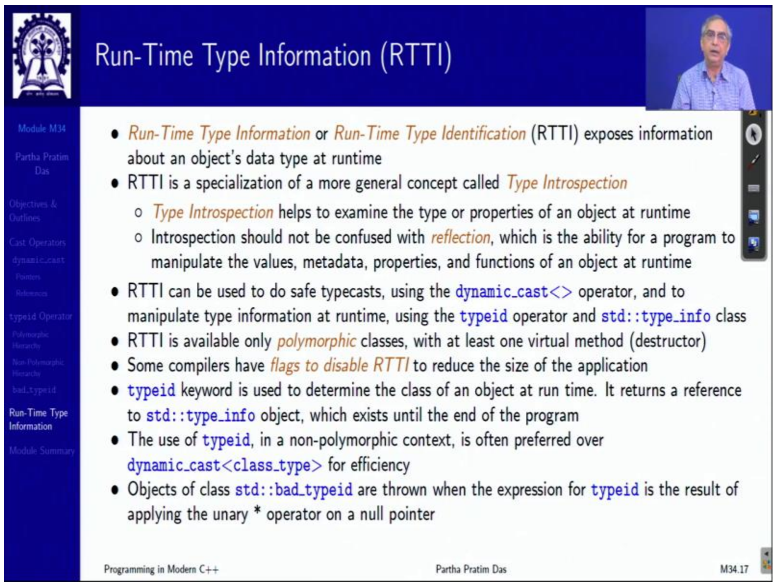
bad_typeid

Run-Time Type Information

Module Summary

Run-Time Type Information (RTTI)

Programming in Modern C++ Partha Pratim Das M34.16



Run-Time Type Information (RTTI)

- *Run-Time Type Information* or *Run-Time Type Identification* (RTTI) exposes information about an object's data type at runtime
- RTTI is a specialization of a more general concept called *Type Introspection*
  - *Type Introspection* helps to examine the type or properties of an object at runtime
  - Introspection should not be confused with *reflection*, which is the ability for a program to manipulate the values, metadata, properties, and functions of an object at runtime
- RTTI can be used to do safe typecasts, using the `dynamic_cast<>` operator, and to manipulate type information at runtime, using the `typeid` operator and `std::type_info` class
- RTTI is available only *polymorphic* classes, with at least one virtual method (destructor)
- Some compilers have *flags to disable RTTI* to reduce the size of the application
- `typeid` keyword is used to determine the class of an object at run time. It returns a reference to `std::type_info` object, which exists until the end of the program
- The use of `typeid`, in a non-polymorphic context, is often preferred over `dynamic_cast<class_type>` for efficiency
- Objects of class `std::bad_typeid` are thrown when the expression for `typeid` is the result of applying the unary * operator on a null pointer

Programming in Modern C++ Partha Pratim Das M34.17

Now, I will not go into the depth of RTTI, it is a very deep subject but I just want to tell you that all this is possible because you can say runtime system is helping your program to find this and in terms of a polymorphic hierarchy, in terms of a non-polymorphic hierarchy we have seen that it has hardly a new value.

In terms of a polymorphic hierarchy, if you ask yourself really where is that information of type coming from, is very simple. We have learnt about the implementation of virtual functions. So, what does a virtual function need? It needs that the class has a unique virtual function table and every instance of the class has a pointer to that class specific virtual function.

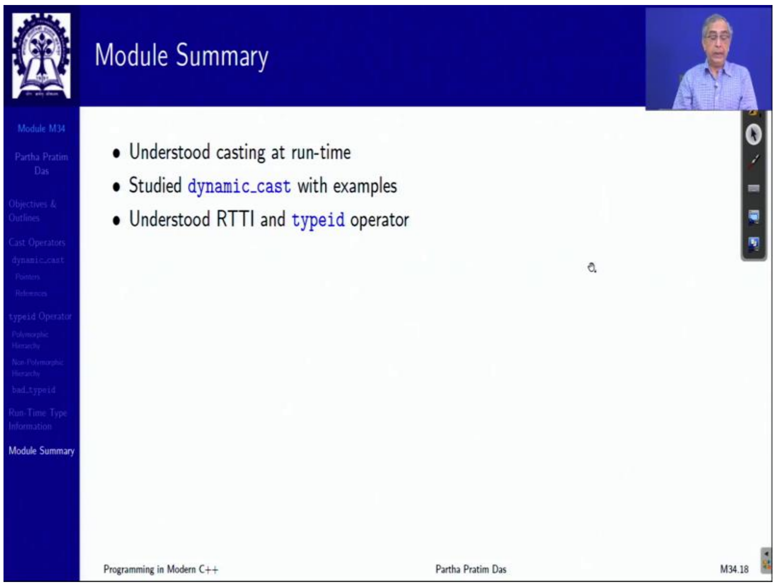
So, what the RTTI does in a gross abstraction, this is not the exact thing but in gross, principle what the RTTI does, it maintains a catalogue of the different classes that exist which are polymorphic and their virtual function table pointer value because that is a unique identity of any object of that class and so when whenever an object is given to RTTI either through downcast operator through the pointer reference or through the typeid operator.

If is a polymorphic object, it will have the naturally the pointer to the virtual function table, RTTI looks up the catalogue of classes and the corresponding virtual function table pointer values and wherever it finds a match it knows that this is the original class of which this object belongs and that is the type, runtime type of the object.

So, that is the basic functionality of the runtime type system in a very simplistic term but in a in terms of an effective understanding that actually the virtual function pointer table becomes the identity of the type which is then converted into its equivalent name and all that and you then you decide on the conversion rule, this type matches the target type that the down cast operator asks for and so on so forth but this in principle is how the whole computation in this, the backbone of the computation in this goes on.

So, that tells you why on a non-polymorphic hierarchy, the dynamic_cast or typeid has no meaning because since in those there is no virtual function pointer table. So, RTTI catalogue has no registry for that. RTTI registry is blank about those objects. So, all that you can infer is only from the compile time type which is the static type which should be always dealt with the static_cast operator not with the dynamic_cast operator.

(Refer Slide Time: 34:09)



Module Summary

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

So, that concludes our discussion on casting, that we have been continuing from the last two modules so this is the third part and we have studied the dynamic_cast, typeid operator and the runtime type information. Thank you for your attention and we will meet again in the next module.