

Class B;

class A { int i; public:

A (int a=0) : i(a)

{ cout << "A:: A(i) \n"; }

operator int()

{ cout << "A:: operator int() \n";

return i;

any logic
can be written
};

class B { public:

conversion operator

operator(A)

Not have
return type

any logic can
be written

{ cout << "B:: operator A() \n";

return A(); }

}; ~~must~~ return A type.

this operator is invoked ~~when we~~ when we
try to cast B type Object into A type
Object.

syntax

operator Type () ! it will return

Type

222 // $B \Rightarrow A$

$a = (b)$ → Here conversion operator in B will be used not the constructor
 $a = b \text{ static_cast } A(b)$
 $a = (A)b$
uses $B::\text{operator } A()$

// $A \Rightarrow \text{int}$

$i = (a)$ → Here conversion operator of a to int will be used.

$i = \text{static_cast } \text{int}(a)$

$i = (\text{int})a$

uses $A::\text{}$

operator int

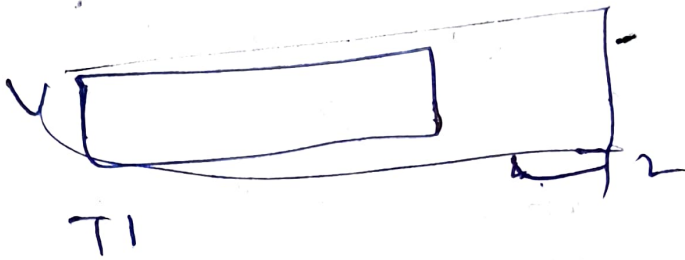
Note:

- ① conversion operator is necessary when we want to convert from user defined type to builtin type
- ②. For two user defined type we could use conversion operator or constructor but not both.

reinterpret_cast Operator

(24)

- It converts any pointer type to any other pointer type even of unrelated class.
- The operation results in a simple binary copy of the value from one pointer to the other.



- All pointer conversions are allowed. However, the content pointed to by the pointer type itself is not checked.
- It can also cast pointers to or from integer types.
- The format in which this integer value represents a pointer is platform specific.
- The only guarantee is that pointer cast to an integer type large enough to fully contain it (such as intptr_t) is guaranteed to be able to ~~cast~~ take back to a valid pointer.

Q22) The conversion that can be performed by `reinterpret_cast` but not by `static_cast` are lowlevel operation based on reinterpreting the binary representations of the type which in most cases result in code which is system specific and thus non portable.

```
class A{};
```

```
class B{};
```

```
int i = 2;
```

```
double d = 3.7
```

```
double *pd = &d;
```

```
i = pd // error
```

```
i = reinterpret_cast<int>(pd) //  
                                okay
```

```
i = (int) pd;
```

```
A *pA
```

```
B *pB
```

```
pA = pB // implicit error
```

```
pA = reinterpret_cast<A*>(pB)
```

```
pA = (A*) pB
```


dynamic_cast

(239)

↳ This is the only cast operator which is based on runtime behaviour of program.

↳ 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.

↳ dynamic_cast can also downcast (convert from pointer-to-base to pointer-to-derived) polymorphic classes (those with virtual members) if and 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

(23) bad-cost is thrown.

* dynamic-cast can also perform the other implicit casts allowed on pointers: casting new pointer both pointer types (even both unrelated classes) and casting any pointer of any type to a void* pointer

dynamic-cast operator: pointers

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

class B: public A {};

class C: { public: virtual ~C() {} }

A a; B b; C c; A* pA; B* pB;

C* pC; void* pV;

pB = &b; pA = dynamic_cast<A*>(pB)

A
↑
B
==

cout << pB << "cast to" << pA << "
void upcast" << endl;

pA = &a; → b type of pointer

pB = dynamic_cast<B*>(pA);

cout << pA << "cast to" << pB << "void
downcast" << endl; contents B type

Case-III

(232)

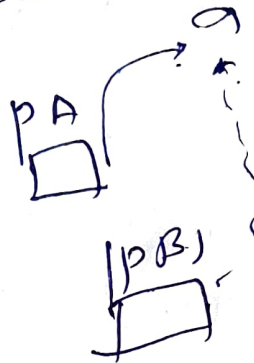
Let's see what's the actual value of PA
if it's some or pointer to the original
return

PA = &a;

.PB = dynamic_cast<B*> (PA);

cout << PA << " cast to " << PB << " Null
Invalid Down cast " << endl;

return NULL



A
↑
B

If we use static cast like
PB = static_cast<B*> (PA); then the
result will be value of PA. bcoz in
statically we don't know what PA is
pointing to.

Case-IV

PA = (A*) &C;

PC = dynamic_cast<C*> (PA);

cout << PA << " cast to " << PC <<

" Invalid unrelated cast " << endl;

Case-V

(23)

$PA = 0;$

$PC = \text{dynamic_cast} \langle C^* \rangle (PA);$

$\text{cout} \ll PA \ll "cast to " \ll PC \ll " un$
- related cast valid for run " end;

Case-VI

$PA = 2a;$

$PV = \text{dynamic_cast} \langle \text{void}^* \rangle (PA);$

$\text{cout} \ll PA \ll "cast to " \ll PV \ll$
" cast to void* valid " end;

(VII)

$PA = \text{dynamic_cast} \langle A^* \rangle (PV);$
1/corror

O/p:

00EFFCA8 cast to 00EFFCA8: valid up
" " " : valid down

00EFFCB4 cast to 00000000: Invalid
down cast

00EFFC9C cast to 00000000: Invalid
unrelated cast

00000000 cast to 00000000: Invalid
valid for NULL

(137) OOEEFFCBY cost to OOEEFFCBY!
cost to ~~void~~ valid valid

dynamic_cast Operator: References

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

class B: public A {};

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

A a B b C c

try {

B &rB1 = b

A &rA2 = dynamic_cast<A*>(rB1);

cout << "upcast valid" << endl;

A &rA3 = b;

B &rB4 = dynamic_cast<B*>(rA3);

cout << "Downcast valid" << endl;

try {

A &rA5 = a;

B &rB6 = dynamic_cast<B*>(rA5);

} catch { bad-cost e }

{ cout << "Downcast invalid" << endl; }

(235) try h

A & rA = (A & C

C & r(C = dynamic_cast<C*>
(rA);

} catch (bad_cast e)

{ cout << "Unrelated cast: Invalid
" << e.what() << endl; }

} catch (bad_cast e)

{ cout << "Bad cast" << e.what()
<< endl; }

}

O/p:

Up cast valid

Down cast valid

Down cast Invalid: Bad dynamic
cast

Unrelated-cast: Invalid:

Bad dynamic cast.

typeid operator

(22)

* ~~typeid~~ typeid

typeid

* 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 object refers to the same type.

* name: implementation defined name of the type.

* typeid operator works for polymorphic type only. (as it uses RTTI

- virtual fn table)

* If the polymorphic objects had, the typeid throws bad_typeid exception.

(237)

ex

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

class B: public A {};

A a

cout << typeid(a).name() <<

typeid(&a).name() << // static

A *p = &a

cout << typeid(p).name() << typeid(*p)

.name() // dynamic

B b

cout << typeid(b).name() << typeid

(&b).name() // static

p = &b

cout << typeid(p).name() << typeid(*p)

.name() << endl // dynamic

What is the type
of object it is pointing to

A r1 = a, A r2 = b

cout << typeid(r1).name() <<

typeid(r2).name();

o/p:

class A, class A*

class A*, class A

class B, class B* (23P)

class A* class B

class A class B

typeid for non polymorphic hierarchy

class X{

class Y : public X{

If the hierarchy is non polymorphic then
it does not have virtual fn. so the
object does not have virtual fn pointer to
the table. so now it is not possible at
runtime to say what is the class from
where the object came.

```
X x;  
cout << typeid(x).name() << typeid(x).  
name() << endl; static
```

Y y;

```
X xq = x;  
cout << typeid(q).name() << typeid(  
xq).name() << endl; dynamic
```

Y y;

```
cout << typeid(y).name() << typeid(y).  
name() << endl;
```

(239)

$q = 2y;$

`cout << typeid(q).name() << typeid`

`(*(q)).name() << endl; // dynamic`
fails

$x \&r1 = x; \quad x \&r2 = y;$

`cout << typeid(r1).name() <<`

`typeid(r2).name() << endl;`

O/p: `class X, class X*`

`class X* class X`

`class Y, class Y*`

`class X*, class X*` // fails
dynamic
type

`class X, class X`

Using typeid operator bad_typeid Exception

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

`class B: public A {};`

`A* pA = new A`

`{`

`cout << typeid(pA).name() <<`

`cout << typeid(*pA).name() <<`
`endl;`

`}`

`catch (const bad_typeid &e)`

`{`

240

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;

}

→ o/p:

class A*

class A

class A*

Caught Access violation -
not RTTI data

Class A*

(241)

A caught Attempted a typeid of
Null pointer!

we should not use typeid as a promise
as it uses RTTI.