```
class B;
class A {int i; public:
    A(int a=0) : i(a)
    { cout<<"A::A()\n"; }

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

class B { public:
    operator A()
    { cout<<"B::operator A()\n";
      return A();
    };
```

any logic can be written

conversion operator

Not compulsory return type

any logic can be written

must return A type.

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

syntax

operator Type() ; it will return

Type

(27) // B ⇒ A

a = (b) → Here conversion operator in B will be used not the constructor

a = b.static_cast<A>(b)

a = (A) b

uses B B::operator A()

// A ⇒ int

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

i = static_cast<int>(a)

i = (int) a

uses A::operator

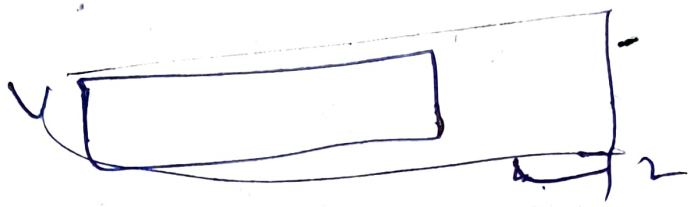
Note:

(i) conversion operator is necessary when we will want to convert from user defined type to built in type.

(ii) . for two user defined type we could use conversion operator or constructor but not both.

reinterpret_cast Operator

- 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



T1

- All pointer conversion are allowed neither the content pointed nor the pointer type itself 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 interpreter is guaranteed to be able to ~~cast~~ take cast back to a valid pointer

(129) The conversion that can be performed by reinterpret_cast but not by static_cast are lowlevel operations based on reinterpreting the binary representation of the type which on most compilers 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

- 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 desired 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_cast is thrown.

* dynamic_cast can also perform the other implicit casts allowed on pointers casting null pointers 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 up cast" << endl;

pA = &b; → b type of pointer

pB = dynamic_cast<B*>(pA);

cout << pA << "cast to" << pB {<"void

downcast" << endl; contents of byte

Case-III

```
pA = &a;
pB = dynamic_cast<B*>(pA);
cout<<pA<<"cast to"<<pB<<"Null
Invalid Down cast"<<endl;
return NULL
```

If we use static_cast like

pB = static_cast<B*>(pA) then the result will be value of pA. because statically we don't know what pA is pointing to.

Case-IV

```
pA = (A*)&C;
pC = dynamic_cast<C*>(A);
cout<<pA<<"cast to"<<pC<<
"Invalid unrelated cast"<<endl;
```

Case-V

```
pA = 0;
pC = dynamic_cast<C*>(pA);
cout << pA << "cast to" << pC << "un
-related cast valid for null" << endl;
```

Case-VI

```
pA = &a;
pV = dynamic_cast<void*>(pA);
cout << pA << "cast to" << pV <<
"cast to void" valid" << endl;
```

(VII)

```
pA = dynamic_cast<A*>(pV);
//error
```

O/P:

00EFFCA8 cast to 00EFFCA8 : valid upcast

" " " : valid downcast

00EFFCB4 cast to 00000000 : Invalid downcast

00EFFC9C cast to 00000000 : Invalid unrelated cast

00000000 cast to 00000000 : unrelated valid for NULL

(3) OOEFFCBY casts to OOEFFCBY :
cast to ~~void~~ void valid

<u>dynamic_cast operator: References</u>

```
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 << "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; }
```

(23) try {

A &rA = (A&)C

C &rC = 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: Upcast valid

Down cast valid

Down cast Invalid: Bad dyna
mic_cast!

Unrelated-cast: Invalid:
Bad dynamic cast

~~typeid operator~~

typeid

typeid

typeid

* typeid operator is used where the dynamic type of a polymorphic objects 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 function table)
* If the polymorphic objects is bad, the typeid throws bad_typeid exception.

(237) Ex. class A { public: virtual ~A(){}
};

class B : public A { };

A a

cout << typeid(a).name() <<
typeid(Ra).name() // Static

A *p = Ra

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

B b

cout << typeid(b).name() << typeid
(Rb).name() // Static

p = &b

cout << typeid(p).name() << typeid(*p)
.name() << endl // dynamic

(p) → Write time type of p

(*p) → Write time 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*

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 a runtime to say where is the class from where the object came.

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

Y(q).
X *q = &x;
cout << typeid(q).name() << typeid(*q).name() << endl; dynamic

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

```
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;
```

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
try {
    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: clan A*
clan A
clan A*
Caught Access violation -
not RTTI data

class A *

* caught Attempted a typeid of Null pointer!

* we should not use typeid as a practice as it uses RTTI.