**static\_cast**

Syntax

static\_cast<new\_type>( expression )

Return: Returns a value of type new\_type.

Additionally, static\_cast can also perform the following:

* Explicitly call a single-argument constructor or a conversion operator.
* Convert to rvalue references.
* Convert enum class values into integers or floating-point values.
* Convert any type to void, evaluating and discarding the value.

Only the following conversions can be done with static\_cast:

(Note: Such conversions should not cast away constness or volatility.)

* If there is an implicit conversion sequence from expression to new\_type.
* If new\_type is the type void (possibly cv-qualified), static\_cast discards the value of expression after evaluating it.
* Convert from void\* to any pointer type. In this case, it guarantees that if the void\* value was obtained by converting from that same pointer type, the resulting pointer value is the same.
* If conversion of expression to new\_type involves lvalue-to-rvalue, array-to-pointer, or function-to-pointer conversion, it can be performed explicitly by static\_cast.

# Conversion to enum types

Convert integers, floating-point values and enum types to enum types:

A value of integer or enumeration type can be converted to any complete enumeration type.

* If the underlying type is not fixed, the result is unspecified if the value of expression is out of range.
* If the underlying type is fixed, the result is the same as the converting the original value first to the underlying type of the enumeration and then to the enumeration type.

A value of a floating-point type can also be converted to any complete enumeration type. The result is the same as converting the original value first to the underlying type of the enumeration, and then to the enumeration type.

# Class cast

If new\_type is a pointer or reference to some class D and the type of expression is a pointer or reference to its non-virtual base B, static\_cast performs a downcast. This downcast is ill-formed if B is ambiguous, inaccessible, or virtual base (or a base of a virtual base) of D.

Such static\_cast makes no runtime checks to ensure that the object's runtime type is actually D, and may only be used safely if this precondition is guaranteed by other means, such as when implementing static polymorphism. Safe downcast may be done with dynamic\_cast.

class Base {};

class Derived: public Base {};

Base \* bPtr = new Base;

Derived \* dPtr = static\_cast<Derived\*>(bPtr);

A pointer to member of some class D can be upcast to a pointer to member of its unambiguous, accessible base class B. This static\_cast makes no checks to ensure the member actually exists in the runtime type of the pointed-to object.

As with all cast expressions, the result is:

* an lvalue if new\_type is an lvalue reference type or an rvalue reference to function type;
* an xvalue if new\_type is an rvalue reference to object type;
* a prvalue otherwise.

Two objects a and b are pointer-interconvertible if:

* they are the same object, or
* one is a union object and the other is a non-static data member of that object, or
* one is a standard-layout class object and the other is the first non-static data member of that object, or, if the object has no non-static data members, the first base class subobject of that object, or
* there exists an object c such that a and c are pointer-interconvertible, and c and b are pointer-interconvertible.

union U { int a; double b; } u;

void\* x = &u; // x's value is "pointer to u"

double\* y = static\_cast<double\*>(x); // y's value is "pointer to u.b"

char\* z = static\_cast<char\*>(x); // z's value is "pointer to u"

**Notes**

static\_cast may also be used to disambiguate function overloads by performing a function-to-pointer conversion to specific type, as in std::transform(s.begin(), s.end(), s.begin(), static\_cast<int(\*)(int)>(std::toupper));

# Examples

// class type casting

#include <iostream>

using namespace std;

class Base {

int m;

public:

Base() : m(0) {}

Base(int val) : m(val) {}

~Base() {m = 0;}

int fun\_base(void) {cout << "m: " << m << endl;}

int get\_m(void) {return m;}

};

class Derived : public Base {

int n;

public:

Derived() : Base(0), n(0) {}

Derived(int val1, int val2) : Base(val1), n(val2) {}

~Derived() {n = 0;}

int fun\_derived() {cout << "m: " << get\_m() << " n: " << n << endl;}

};

int main(void) {

Base \* bptr1 = new Base(5);

Derived \* dptr1 = new Derived(7, 13);

Base \* bptr2 = nullptr;

Derived \* dptr2 = nullptr;

bptr2 = static\_cast<Base \*>(dptr1);

dptr2 = static\_cast<Derived \*>(bptr1);

bptr1->fun\_base();

dptr1->fun\_derived();

bptr2->fun\_base();

dptr2->fun\_derived();

return 0;

}

Output:

m: 5

m: 7 n: 13

m: 7

m: 5 n: 0

//simple casting

#include <iostream>

#include <vector>

using namespace std;

enum class enum\_class\_var {ZERO, ONE, TWO, THREE};

enum enum\_var {ZERO, ONE, FOUR = 4, FIVE};

int main(void) {

int int\_var = 0;

short int sint\_var = 10;

double dbl\_var = 0.0;

int\_var = static\_cast<int>(6325.253);

cout << "static\_cast<int>(6325.253) " << int\_var << endl;

int\_var = static\_cast<int>(sint\_var);

cout << "static\_cast<int>(sint\_var) " << int\_var << endl;

cout << endl;

sint\_var = static\_cast<short int>(985623);

cout << "static\_cast<short int>(985623) " << sint\_var << endl;

sint\_var = static\_cast<short int>(2569843.256);

cout << "static\_cast<short int>(2569843.256) " << sint\_var << endl;

cout << endl;

dbl\_var = static\_cast<double>(8563246);

cout << "static\_cast<double>(8563246) " << dbl\_var << endl;

sint\_var = 5;

dbl\_var = static\_cast<double>(sint\_var);

cout << "static\_cast<double>(sint\_var) " << dbl\_var << endl;

cout << endl;

cout << "------------------------" << endl;

// enum to int and vice versa

enum\_var evar = ONE;

int\_var = static\_cast<int>(evar);

cout << "static\_cast<int>(evar) " << int\_var << endl;

enum\_class\_var ecvar = enum\_class\_var::TWO;

int\_var = static\_cast<int>(ecvar);

cout << "static\_cast<int>(ecvar) " << int\_var << endl;

evar = static\_cast<enum\_var>(3);

cout << "static\_cast<enum\_var>(3) " << evar << endl;

evar = static\_cast<enum\_var>(7);

cout << "static\_cast<enum\_var>(7) " << evar << endl;

ecvar = static\_cast<enum\_class\_var>(3);

//cout << "static\_cast<enum\_class\_var>(3) " << ecvar << endl; // CE error: no match for 'operator<<' (operand types are 'std::basic\_ostream<char>' and 'enum\_class\_var')

ecvar = static\_cast<enum\_class\_var>(7);

// cout << "static\_cast<enum\_class\_var>(7) " << ecvar << endl; // CE error: no match for 'operator<<' (operand types are 'std::basic\_ostream<char>' and 'enum\_class\_var')

evar = FOUR;

ecvar = static\_cast<enum\_class\_var>(evar);

//cout << "static\_cast<enum\_class\_var>(evar) " << ecvar << endl; // error: no match for 'operator<<' (operand types are 'std::basic\_ostream<char>' and 'enum\_class\_var')

ecvar = enum\_class\_var::THREE;

evar = static\_cast<enum\_var>(ecvar);

cout << "static\_cast<enum\_var>(ecvar) " << evar << endl;

return 0;

}

Output:

static\_cast<int>(6325.253) 6325

static\_cast<int>(sint\_var) 10

static\_cast<short int>(985623) 2583

static\_cast<short int>(2569843.256) 32767

static\_cast<double>(8563246) 8.56325e+06

static\_cast<double>(sint\_var) 5

------------------------

static\_cast<int>(evar) 1

static\_cast<int>(ecvar) 2

static\_cast<enum\_var>(3) 3

static\_cast<enum\_var>(7) 7

static\_cast<enum\_var>(ecvar) 3

# END