**const\_cast**

Syntax

const\_cast<new\_type>( expression )

Returns a value of type new\_type.

A const\_cast operator adds or removes a const or volatile modifier to or from a type.

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; **(C++ 11)**
* a prvalue otherwise.

A single const\_cast expression may add or remove any number of const or volatile modifiers.

If a pointer to T1 can be converted to a pointer to T2 using const\_cast<T2>, where T1 and T2 are object types, you can also make the following types of conversions:

* An lvalue of type T1 to an lvalue of type T2 using const\_cast<T2&>
* An lvalue or xvalue of type T1 to an xvalue of type T2 using const\_cast<T2&&> **(C++ 11)**
* A prvalue of class type T1 to an xvalue of type T2 using const\_cast<T2&&> **(C++ 11)**

**Notes**

1. Pointers to functions and pointers to member functions are not subject to const\_cast
2. const\_cast makes it possible to form a reference or pointer to non-const type that is actually referring to a const object or a reference or pointer to non-volatile type that is actually referring to a volatile object.

Modifying a const object through a non-const access path and referring to a volatile object through a non-volatile glvalue results in undefined behavior.

# Example

#include <iostream>

using namespace std;

void f(int\* p) {

cout << \*p << endl;

}

int main(void) {

const int a = 10;

const int\* b = &a;

// Function f() expects int\*, not const int\* f(b);

int\* c = const\_cast<int\*>(b);

f(c);

//\*b = 20; // error: assignment of read-only location '\* b'

//\*c = 30; // Undefined behavior

int a1 = 40;

const int\* b1 = &a1;

int\* c1 = const\_cast<int\*>(b1);

// Integer a1, the object referred to by c1, has not been declared const

\*c1 = 50;

return 0;

}

Output:

10

# END