const_cast conversion

Converts between types with different cv-qualification.

Syntax

const_cast < *new_type* > (*expression*)

Returns a value of type new_type.

Explanation

Only the following conversions can be done with `const_cast`. In particular, only `const_cast` may be used to cast away (remove) constness or volatility.

1) Two possibly multilevel pointers to the same type may be converted between each other, regardless of cv-qualifiers at each level.
2) lvalue of any type T may be converted to a lvalue or rvalue reference to the same type T, more or less cv-qualified. Likewise, an rvalue may be converted to a more or less cv-qualified rvalue reference.
3) Same rules apply to possibly multilevel pointers to data members.
4) null pointer value may be converted to the null pointer value of *new_type*

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.

Notes

Pointers to functions and pointers to member functions are not subject to `const_cast`

Even though `const_cast` may remove constness or volatility from any pointer or reference, using the resulting pointer or reference to write to an object that was declared `const` or to access an object that was declared `volatile` invokes undefined behavior.

Keywords

`const_cast`

Example

Run this code

```
#include <iostream>

struct type {
    type() :i(3) {}
    void m1(int v) const {
        // this->i = v;                 // compile error: this is a pointer to const
        const_cast<type*>(this)->i = v; // OK as long as the type object isn't const
    }
    int i;
};

int main()
{
```

```
    int i = 3;                    // i is not declared const
    const int& cref_i = i;
    const_cast<int&>(cref_i) = 4; // OK: modifies i
    std::cout << "i = " << i << '\n';

    type t; // note, if this is const type t;, then t.m1(4); is UB
    t.m1(4);
    std::cout << "type::i = " << t.i << '\n';

    const int j = 3; // j is declared const
    int* pj = const_cast<int*>(&j);
    *pj = 4;         // undefined behavior!

    void (type::*mfp)(int) const = &type::m1; // pointer to member function
//  const_cast<void(type::*)(int)>(mfp); // compiler error: const_cast does not
                                         // work on function pointers
}
```

Output:

```
i = 4
type::i = 4
```

See also

- static_cast
- dynamic_cast
- reinterpret_cast
- explicit cast
- implicit conversions