Logical operators

Returns the result of a boolean operation.

Operator name	Syntax	Overloadable	Prototype examples (for `class T`)	
			Inside class definition	Outside class definition
negation	`not a` `!a`	Yes	`bool T::operator!() const;`	`bool operator!(const T &a);`
AND	`a and b` `a && b`	Yes	`bool T::operator&&(const T2 &b) const;`	`bool operator&&(const T &a, const T2 &b);`
inclusive OR	`a or b` `a || b`	Yes	`bool T::operator||(const T2 &b) const;`	`bool operator||(const T &a, const T2 &b);`

Notes

- The keyword-like forms (`and`, `or`, `not`) and the symbol-like forms (`&&`, `||`, `!`) can be used interchangeably (See alternative representations)
- All built-in operators return `bool`, and most user-defined overloads also return `bool` so that the user-defined operators can be used in the same manner as the built-ins. However, in a user-defined operator overload, any type can be used as return type (including `void`).
- Builtin operators **&&** and **||** perform short-circuit evaluation (do not evaluate the second operand if the result is known after evaluating the first), but overloaded operators behave like regular function calls and always evaluate both operands

Explanation

The logic operator expressions have the form

! *rhs*	(1)
lhs **&&** *rhs*	(2)
lhs **||** *rhs*	(3)

1) Logical NOT
2) Logical AND
3) Logical inclusive OR

If the operand is not `bool`, it is converted to `bool` using contextual conversion to bool: it is only well-formed if the declaration `bool t(arg)` is well-formed, for some invented temporary `t`.

The result is a `bool` prvalue.

For the built-in logical NOT operator, the result is `true` if the operand is `false`. Otherwise, the result is `false`.

For the built-in logical AND operator, the result is `true` if both operands are `true`. Otherwise, the result is `false`. This operator is short-circuiting : if the first operand is `false`, the second operand is not evaluated

For the built-in logical OR operator, the result is `true` if either the first or the second operand (or both) is `true`. This operator is short-circuiting: if the first operand is `true`, the second operand is not evaluated.

Note that bitwise logic operators do not perform short-circuiting.

Results

a	`true`	`false`
!a	`false`	`true`

and		**a**	
		`true`	`false`
b	`true`	`true`	`false`
	`false`	`false`	`false`

or		a	
		true	false
b	true	true	true
	false	true	false

In overload resolution against user-defined operators, the following built-in function signatures participate in overload resolution:

```
bool operator!(bool)
bool operator&&(bool, bool)
bool operator||(bool, bool)
```

Example

Run this code

```
#include <iostream>
#include <string>
int main()
{
    int n = 2;
    int* p = &n;
    // pointers are convertible to bool
    if(    p && *p == 2   // "*p" is safe to use after "p &&"
       || !p &&  n != 2 ) // || has lower precedence than &&
        std::cout << "true\n";

    // streams are also convertible to bool
    std::cout << "Enter 'quit' to quit.\n";
    for(std::string line;    std::cout << "> "
                          && std::getline(std::cin, line)
                          && line != "quit"; )
        ;
}
```

Output:

```
true
Enter 'quit' to quit.
> test
> quit
```

Standard library

Because the short-circuiting properties of operator&& and operator|| do not apply to overloads, and because types with boolean semantics are uncommon, only two standard library classes overload these operators:

operator!	applies a unary arithmetic operator to each element of the valarray (public member function of std::valarray<T>)
operator&& **operator||**	applies binary operators to each element of two valarrays, or a valarray and a value (function template)
operator!	checks if an error has occurred (synonym of fail()) (public member function of std::basic_ios<CharT,Traits>)

See also

Operator precedence

Operator overloading

Common operators						
assignment	increment decrement	arithmetic	**logical**	comparison	member access	other

`a = b`
`a += b`
`a -= b`
`a *= b`
`a /= b`
`a %= b`
`a &= b`
`a |= b`
`a ^= b`
`a <<= b`
`a >>= b`
	`++a`
	`--a`
	`a++`
	`a--`
		`+a`
		`-a`
		`a + b`
		`a - b`
		`a * b`
		`a / b`
		`a % b`
		`~a`
		`a & b`
		`a | b`
		`a ^ b`
		`a << b`
		`a >> b`
			`!a`
			`a && b`
			`a || b`
				`a == b`
				`a != b`
				`a < b`
				`a > b`
				`a <= b`
				`a >= b`
				`a <=> b`
					`a[b]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						`a(...)`
						`a, b`
						`a ? b : c`

Special operators
`static_cast` converts one type to another related type `dynamic_cast` converts within inheritance hierarchies `const_cast` adds or removes cv qualifiers `reinterpret_cast` converts type to unrelated type C-style cast converts one type to another by a mix of `static_cast`, `const_cast`, and `reinterpret_cast` `new` creates objects with dynamic storage duration `delete` destructs objects previously created by the new expression and releases obtained memory area `sizeof` queries the size of a type `sizeof...` queries the size of a parameter pack (since C++11) `typeid` queries the type information of a type `noexcept` checks if an expression can throw an exception (since C++11) `alignof` queries alignment requirements of a type (since C++11)

C documentation for **Logical operators**