C++ Operator Precedence

The following table lists the precedence and associativity of C++ operators. Operators are listed top to bottom, in descending precedence. `a`, `b` and `c` are operands.

Precedence	Operator	Description	Associativity
1	`a::b`	Scope resolution	Left-to-right →
2	`a++` `a--`	Suffix/postfix increment and decrement	
	`type(a)` `type{a}`	Functional cast	
	`a()`	Function call	
	`a[]`	Subscript	
	`a.b` `a->b`	Member access	
3	`++a` `--a`	Prefix increment and decrement	Right-to-left ←
	`+a` `-a`	Unary plus and minus	
	`!a` `~a`	Logical NOT and bitwise NOT	
	`(type)a`	C-style cast	
	`*a`	Indirection (dereference)	
	`&a`	Address-of	
	`sizeof`	Size-of[note 1]	
	`co_await`	await-expression (C++20)	
	`new` – `new[]`	Dynamic memory allocation	
	`delete` – `delete[]`	Dynamic memory deallocation	
4	`a.*b` `a->*b`	Pointer-to-member	Left-to-right →
5	`a * b` `a / b` `a % b`	Multiplication, division, and remainder	
6	`a + b` `a - b`	Addition and subtraction	
7	`a << b` `a >> b`	Bitwise left shift and right shift	
8	`a <=> b`	Three-way comparison operator (since C++20)	
9	`a < b` `a <= b` `a > b` `a >= b`	For relational operators < and <= and > and >= respectively	
10	`a == b` `a != b`	For equality operators = and != respectively	
11	`a & b`	Bitwise AND	
12	`a ^ b`	Bitwise XOR (exclusive or)	
13	`a | b`	Bitwise OR (inclusive or)	
14	`a && b`	Logical AND	
15	`a || b`	Logical OR	
16	`a ? b : c`	Ternary conditional[note 2]	Right-to-left ←
	`throw`	throw operator	
	`co_yield`	yield-expression (C++20)	
	`a = b`	Direct assignment (provided by default for C++ classes)	
	`a += b` `a -= b`	Compound assignment by sum and difference	
	`a *= b` `a /= b` `a %= b`	Compound assignment by product, quotient, and remainder	
	`a <<= b` `a >>= b`	Compound assignment by bitwise left shift and right shift	
	`a &= b` `a ^= b` `a |= b`	Compound assignment by bitwise AND, XOR, and OR	
17	`a, b`	Comma	Left-to-right →

1. ↑ The operand of `sizeof` cannot be a C-style type cast: the expression `sizeof (int) * p` is unambiguously interpreted as `(sizeof(int)) * p`, but not `sizeof((int)*p)`.
2. ↑ The expression in the middle of the conditional operator (between **?** and **:**) is parsed as if parenthesized: its precedence relative to **?:** is ignored.

When parsing an expression, an operator which is listed on some row of the table above with a precedence will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it with a lower precedence. For example, the expressions `std::cout << a & b` and `*p++` are parsed as `(std::cout << a) & b` and `*(p++)`, and not as `std::cout << (a & b)` or `(*p)++`.

Operators that have the same precedence are bound to their arguments in the direction of their associativity. For example, the expression `a = b = c` is parsed as `a = (b = c)`, and not as `(a = b) = c` because of right-to-left associativity of assignment, but `a + b - c` is parsed `(a + b) - c` and not `a + (b - c)` because of left-to-right associativity of addition and subtraction.

Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`delete ++*p` is `delete(++(*p))`) and unary postfix operators always associate left-to-right (`a[1][2]++` is `((a[1])[2])++`). Note that the associativity is meaningful for member access operators, even though they are grouped with unary postfix operators: `a.b++` is parsed `(a.b)++` and not `a.(b++)`.

Operator precedence is unaffected by operator overloading. For example, `std::cout << a ? b : c;` parses as `(std::cout << a) ? b : c;` because the precedence of arithmetic left shift is higher than the conditional operator.

Notes

Precedence and associativity are compile-time concepts and are independent from order of evaluation, which is a runtime concept.

The standard itself doesn't specify precedence levels. They are derived from the grammar.

`const_cast`, `static_cast`, `dynamic_cast`, `reinterpret_cast`, `typeid`, `sizeof...`, `noexcept` and `alignof` are not included since they are never ambiguous.

Some of the operators have alternate spellings (e.g., `and` for `&&`, `or` for `||`, `not` for `!`, etc.).

In C, the ternary conditional operator has higher precedence than assignment operators. Therefore, the expression `e = a < d ? a++ : a = d`, which is parsed in C++ as `e = ((a < d) ? (a++) : (a = d))`, will fail to compile in C due to grammatical or semantic constraints in C. See the corresponding C page for details.

See also

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[...]`
					`*a`
					`&a`
					`a->b`
					`a.b`
					`a->*b`
					`a.*b`
						function call
						`a(...)`
						comma
						`a, b`
						conditional
						`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 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 **C operator precedence**