C++ Operator Precedence

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

Precedence	Operator	Description	Associativity
1	`::`	Scope resolution	Left-to-right

2	`++ --`
	type`()` *type*`{}`
	`()`
	`[]`
	`.`
	`->`
		Suffix/postfix increment and decrement
		Function-style type cast
		Function call
		Array subscripting
		Element selection by reference
		Element selection through pointer	

3	`++ --`
	`+ -`
	`! ~`
	`(`*type*`)`
	`*`
	`&`
	`sizeof`
	`new`, `new[]`
	`delete`, `delete[]`
		Prefix increment and decrement
		Unary plus and minus
		Logical NOT and bitwise NOT
		C-style type cast
		Indirection (dereference)
		Address-of
		Size-of[note 1]
		Dynamic memory allocation
		Dynamic memory deallocation
			Right-to-left

4	`.* ->*`	Pointer to member	Left-to-right
5	`* / %`	Multiplication, division, and remainder	
6	`+ -`	Addition and subtraction	
7	`<< >>`	Bitwise left shift and right shift	

8	`< <=`
	`> >=`
		For relational operators < and ≤ respectively
		For relational operators > and ≥ respectively	

9	`== !=`	For relational = and ≠ respectively	
10	`&`	Bitwise AND	
11	`^`	Bitwise XOR (exclusive or)	
12	`|`	Bitwise OR (inclusive or)	
13	`&&`	Logical AND	
14	`||`	Logical OR	

15	`?:`
	`=`
	`+= -=`
	`*= /= %=`
	`<<= >>=`
	`&= ^= |=`
		Ternary conditional[note 2]
		Direct assignment (provided by default for C++ classes)
		Assignment by sum and difference
		Assignment by product, quotient, and remainder
		Assignment by bitwise left shift and right shift
		Assignment by bitwise AND, XOR, and OR
			Right-to-left

16	`throw`	Throw operator (for exceptions)	
17	`,`	Comma	Left-to-right

1. ↑ The operand of `sizeof` can't 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 will be bound tighter (as if by parentheses) to its arguments than any operator that is listed on a row further below it. 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 are in the same cell (there may be several rows of operators listed in a cell) are evaluated with the same precedence, in the given direction. 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.

Operator precedence is unaffected by operator overloading.

Notes

Precedence and associativity are independent from order of evaluation.

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.).

Relative precedence of the conditional and assignment operators differs between C and C++: in C, assignment is not allowed on the right hand side of a conditional operator, so `e = a < d ? a++ : a = d` cannot be parsed. Many C compilers use a modified grammar where `?:` has higher precedence than =, which parses that as `e = ( ((a < d) ? (a++) : a) = d )` (which then fails to compile because `?:` is never lvalue in C and = requires lvalue on the left). In C++, `?:` and = have equal precedence and group right-to-left, so that `e = a < d ? a++ : a = d` parses as `e = ((a < d) ? (a++) : (a = d))`.

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->b a.b a->*b a.*b	a(...) a, b (type) a ? :
Special operators						
`static_cast` converts one type to another compatible type `dynamic_cast` converts virtual base class to derived class `const_cast` converts type to compatible type with different cv qualifiers `reinterpret_cast` converts type to incompatible type `new` allocates memory `delete` deallocates memory `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 **`C operator precedence`**

Retrieved from "http://en.cppreference.com/mwiki/index.php?title=cpp/language/operator_precedence&oldid=72147"