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	`++ --`	Suffix/postfix increment and decrement	Left-to-right
	`( )`	Function call	
	`[ ]`	Array subscripting	
	`.`	Structure and union member access	
	`->`	Structure and union member access through pointer	
	`(`*type*`){`*list*`}`	Compound literal(C99)	
2	`++ --`	Prefix increment and decrement[note 1]	Right-to-left
	`+ -`	Unary plus and minus	
	`! ~`	Logical NOT and bitwise NOT	
	`(`*type*`)`	Cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of[note 2]	
	`_Alignof`	Alignment requirement(C11)	
3	`* / %`	Multiplication, division, and remainder	Left-to-right
4	`+ -`	Addition and subtraction	
5	`<< >>`	Bitwise left shift and right shift	
6	`< <=`	For relational operators < and ≤ respectively	
	`> >=`	For relational operators > and ≥ respectively	
7	`== !=`	For relational = and ≠ respectively	
8	`&`	Bitwise AND	
9	`^`	Bitwise XOR (exclusive or)	
10	`|`	Bitwise OR (inclusive or)	
11	`&&`	Logical AND	
12	`||`	Logical OR	
13	`?:`	Ternary conditional[note 3]	Right-to-left
14[note 4]	`=`	Simple assignment	
	`+= -=`	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	
15	`,`	Comma	Left-to-right

1. ↑ The operand of prefix ++ and `--` can't be a type cast. This rule grammatically forbids some expressions that would be semantically invalid anyway. Some compilers ignore this rule and detect the invalidity semantically.
2. ↑ The operand of `sizeof` can't be a type cast: the expression `sizeof (int) * p` is unambiguously interpreted as `(sizeof(int)) * p`, but not `sizeof((int)*p)`.
3. ↑ The expression in the middle of the conditional operator (between **?** and **:**) is parsed as if parenthesized: its precedence relative to `?:` is ignored.
4. ↑ Assignment operators' left operands must be unary (level-2 non-cast) expressions. This rule grammatically forbids some expressions that would be semantically invalid anyway. Many compilers ignore this rule and detect the invalidity semantically. For example, `e = a < d ? a++ : a = d` is an expression that cannot be parsed because of this rule. However, many compilers ignore this rule and parse it as `e = ( ((a < d) ? (a++) : a) = d )`, and then give an error because it is semantically invalid.

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 expression `*p++` is parsed as `*(p++)`, and not as `(*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.

Notes

Precedence and associativity are independent from order of evaluation.

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

In C++, the conditional operator has the same precedence as assignment operators, and prefix ++ and `--` and assignment operators don't have the restrictions about their operands.

Associativity specification is redundant for unary operators and is only shown for completeness: unary prefix operators always associate right-to-left (`sizeof ++*p` is `sizeof(++(*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++)`.

References

- C17 standard (ISO/IEC 9899:2018):
  - A.2.1 Expressions
- C11 standard (ISO/IEC 9899:2011):
  - A.2.1 Expressions
- C99 standard (ISO/IEC 9899:1999):
  - A.2.1 Expressions
- C89/C90 standard (ISO/IEC 9899:1990):
  - A.1.2.1 Expressions

See also

Order of evaluation of operator arguments at run time.

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(...)`
						`a, b`
						`(type) a`
						`? :`
						`sizeof`
						`_Alignof` (since C11)

C++ documentation for **C++ operator precedence**