# 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** | `a++` `a--` | Suffix/postfix increment and decrement | |
| | *type*`()` *type*`{}` | Functional cast | |
| | `a()` | Function call | |
| | `a[]` | Subscript | |
| | `.` `->` | Member access | |
| **3** | `++a` `--a` | Prefix increment and decrement | Right-to-left ← |
| | `+a` `-a` | Unary plus and minus | |
| | `!` `~` | Logical NOT and bitwise NOT | |
| | (*type*) | 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** | `.*` `->*` | 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** | `<<` `>>` | Bitwise left shift and right shift | |
| **8** | `<=>` | Three-way comparison operator (since C++20) | |
| **9** | `<` `<=` `>` `>=` | For relational operators < and ≤ and > and ≥ respectively | |
| **10** | `==` `!=` | For equality operators = and ≠ respectively | |
| **11** | `a&b` | Bitwise AND | |
| **12** | `^` | Bitwise XOR (exclusive or) | |
| **13** | `\|` | Bitwise OR (inclusive or) | |
| **14** | `&&` | Logical AND | |
| **15** | `\|\|` | Logical OR | |
| **16** | `a?b:c` | Ternary conditional[note 2] | Right-to-left ← |
| | `throw` | throw operator | |
| | `co_yield` | yield-expression (C++20) | |
| | `=` | Direct assignment (provided by default for C++ classes) | |
| | `+=` `-=` | Compound assignment by sum and difference | |
| | `*=` `/=` `%=` | Compound assignment by product, quotient, and remainder | |
| | `<<=` `>>=` | Compound assignment by bitwise left shift and right shift | |
| | `&=` `^=` `\|=` | Compound assignment by bitwise AND, XOR, and OR | |
| **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 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`<br>`a += b`<br>`a -= b`<br>`a *= b`<br>`a /= b`<br>`a %= b`<br>`a &= b`<br>`a \|= b`<br>`a ^= b`<br>`a <<= b`<br>`a >>= b` | `++a`<br>`--a`<br>`a++`<br>`a--` | `+a`<br>`-a`<br>`a + b`<br>`a - b`<br>`a * b`<br>`a / b`<br>`a % b`<br>`~a`<br>`a & b`<br>`a \| b`<br>`a ^ b`<br>`a << b`<br>`a >> b` | `!a`<br>`a && b`<br>`a \|\| b` | `a == b`<br>`a != b`<br>`a < b`<br>`a > b`<br>`a <= b`<br>`a >= b`<br>`a <=> b` | `a[b]`<br>`*a`<br>`&a`<br>`a->b`<br>`a.b`<br>`a->*b`<br>`a.*b` | `a(...)`<br>`a, b`<br>`a ? b : c` |
| **Special operators** | | | | | | |
| `static_cast` converts one type to another related type<br>`dynamic_cast` converts within inheritance hierarchies<br>`const_cast` adds or removes cv qualifiers<br>`reinterpret_cast` converts type to unrelated type<br>C-style cast converts one type to another by a mix of `static_cast`, `const_cast`, and `reinterpret_cast`<br>`new` creates objects with dynamic storage duration<br>`delete` destructs objects previously created by the new expression and releases obtained memory area<br>`sizeof` queries the size of a type<br>`sizeof...` queries the size of a parameter pack (since C++11)<br>`typeid` queries the type information of a type<br>`noexcept` checks if an expression can throw an exception (since C++11)<br>`alignof` queries alignment requirements of a type (since C++11) | | | | | | |

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