C++ keywords

This is a list of reserved keywords in C++. Since they are used by the language, these keywords are not available for re-definition or overloading.

A – C	D – P	R – Z

`alignas` (since C++11)
`alignof` (since C++11)
`and`
`and_eq`
`asm`
`atomic_cancel` (TM TS)
`atomic_commit` (TM TS)
`atomic_noexcept` (TM TS)
`auto` (1)
`bitand`
`bitor`
`bool`
`break`
`case`
`catch`
`char`
`char8_t` (since C++20)
`char16_t` (since C++11)
`char32_t` (since C++11)
`class` (1)
`compl`
`concept` (since C++20)
`const`
`consteval` (since C++20)
`constexpr` (since C++11)
`constinit` (since C++20)
`const_cast`
`continue`
`co_await` (since C++20)
`co_return` (since C++20)
`co_yield` (since C++20)
	`decltype` (since C++11)
	`default` (1)
	`delete` (1)
	`do`
	`double`
	`dynamic_cast`
	`else`
	`enum`
	`explicit`
	`export` (1) (3)
	`extern` (1)
	`false`
	`float`
	`for`
	`friend`
	`goto`
	`if`
	`inline` (1)
	`int`
	`long`
	`mutable` (1)
	`namespace`
	`new`
	`noexcept` (since C++11)
	`not`
	`not_eq`
	`nullptr` (since C++11)
	`operator`
	`or`
	`or_eq`
	`private`
	`protected`
	`public`
		`reflexpr` (reflection TS)
		`register` (2)
		`reinterpret_cast`
		`requires` (since C++20)
		`return`
		`short`
		`signed`
		`sizeof` (1)
		`static`
		`static_assert` (since C++11)
		`static_cast`
		`struct` (1)
		`switch`
		`synchronized` (TM TS)
		`template`
		`this`
		`thread_local` (since C++11)
		`throw`
		`true`
		`try`
		`typedef`
		`typeid`
		`typename`
		`union`
		`unsigned`
		`using` (1)
		`virtual`
		`void`
		`volatile`
		`wchar_t`
		`while`
		`xor`
		`xor_eq`

- (1) — meaning changed or new meaning added in C++11.
- (2) — meaning changed in C++17.
- (3) — meaning changed in C++20.

Note that `and`, `bitor`, `or`, `xor`, `compl`, `bitand`, `and_eq`, `or_eq`, `xor_eq`, `not`, and `not_eq` (along with the digraphs `<%`, `%>`, `<:`, `:>`, `%:`, and `%:%:`) provide an alternative way to represent standard tokens.

In addition to keywords, there are *identifiers with special meaning*, which may be used as names of objects or functions, but have special meaning in certain contexts.

`final` (C++11) `override` (C++11) `transaction_safe` (TM TS) `transaction_safe_dynamic` (TM TS) `import` (C++20) `module` (C++20)

Also, all identifiers that contain a double underscore `__` in any position and each identifier that begins with an underscore followed by an uppercase letter is always reserved and all identifiers that begin with an underscore are reserved for use as names in the global namespace. See identifiers for more details.

The namespace `std` is used to place names of the standard C++ library. See Extending namespace std for the rules about adding names to it.

The name `posix` is reserved for a future top-level namespace. The behavior is undefined if a program declares or defines anything in that namespace. (since C++11)

The following tokens are recognized by the preprocessor when in context of a preprocessor directive:

`if`
`elif`
`else`
`endif`
	`ifdef`
	`ifndef`
	`define`
	`undef`
		`include`
		`line`
		`error`
		`pragma`
			`defined`
			`__has_include` (since C++17)
			`__has_cpp_attribute` (since C++20)
				`export` (C++20)
				`import` (C++20)
				`module` (C++20)

The following tokens are recognized by the preprocessor *outside* the context of a preprocessor directive:

`_Pragma` (since C++11)

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 ≤ respectively	
	`>` `>=`	For relational operators > and ≥ respectively	
10	`==` `!=`	For relational operators = and ≠ respectively	
11	`&`	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`
`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`
						`? :`

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 **`C operator precedence`**