Operators in C and C++

From Wikipedia, the free encyclopedia

This is a list of operators in the C and C++ programming languages. All the operators listed exist in C++; the fourth column "Included in C", dictates whether an operator is also present in C. Note that C does not support operator overloading.

When not overloaded, for the operators `&&`, `||`, and `,` (the comma operator), there is a sequence point after the evaluation of the first operand.

C++ also contains the type conversion operators `const_cast`, `static_cast`, `dynamic_cast`, and `reinterpret_cast`. The formatting of these operators means that their precedence level is unimportant.

Most of the operators available in C and C++ are also available in other languages such as C#, Java, Perl, and PHP with the same precedence, associativity, and semantics.

Contents

Table

For the purposes of this table, `a`, `b`, and `c` represent valid values (literals, values from variables, or return value), object names, or lvalues, as appropriate. `R`, `S` and `T` stand for any type(s), and `K` for a class type or enumerated type.

"Can overload" means that the operator can be overloaded in C++. "Included in C" means that the operator exists and has a semantic meaning in C (operators are not overloadable in C).

Arithmetic operators

Operator name		Syntax	Can overload	Included in C	Prototype examples	
					As member of K	Outside class definitions
Basic assignment		`a = b`	Yes	Yes	`R& K::operator =(S b);`	N/A
Addition		`a + b`	Yes	Yes	`R K::operator +(S b);`	`R operator +(K a, S b);`
Subtraction		`a - b`	Yes	Yes	`R K::operator -(S b);`	`R operator -(K a, S b);`
Unary plus (integer promotion)		`+a`	Yes	Yes	`R K::operator +();`	`R operator +(K a);`
Unary minus (additive inverse)		`-a`	Yes	Yes	`R K::operator -();`	`R operator -(K a);`
Multiplication		`a * b`	Yes	Yes	`R K::operator *(S b);`	`R operator *(K a, S b);`
Division		`a / b`	Yes	Yes	`R K::operator /(S b);`	`R operator /(K a, S b);`
Modulo (integer remainder)[a]		`a % b`	Yes	Yes	`R K::operator %(S b);`	`R operator %(K a, S b);`
Increment	Prefix	`++a`	Yes	Yes	`R& K::operator ++();`	`R& operator ++(K a);`
	Postfix	`a++`	Yes	Yes	`R K::operator ++(int);`	`R operator ++(K a, int);`
					Note: C++ uses the unnamed dummy-parameter `int` to differentiate between prefix and suffix increment operators.	
Decrement	Prefix	`--a`	Yes	Yes	`R& K::operator --();`	`R& operator --(K a);`
	Postfix	`a--`	Yes	Yes	`R K::operator --(int);`	`R operator --(K a, int);`
					Note: C++ uses the unnamed dummy-parameter `int` to differentiate between prefix and suffix decrement operators.	

Comparison operators/relational operators

Operator name	Syntax	Can overload	Included in C	Prototype examples	
				As member of K	Outside class definitions
Equal to	`a == b`	Yes	Yes	`R K::operator ==(S b);`	`R operator ==(K a, S b);`
Not equal to	`a != b` `a not_eq b`[b]	Yes	Yes	`R K::operator !=(S b);`	`R operator !=(K a, S b);`
Greater than	`a > b`	Yes	Yes	`R K::operator >(S b);`	`R operator >(K a, S b);`
Less than	`a < b`	Yes	Yes	`R K::operator <(S b);`	`R operator <(K a, S b);`
Greater than or equal to	`a >= b`	Yes	Yes	`R K::operator >=(S b);`	`R operator >=(K a, S b);`
Less than or equal to	`a <= b`	Yes	Yes	`R K::operator <=(S b);`	`R operator <=(K a, S b);`

Logical operators

Operator name	Syntax	Can overload	Included in C	Prototype examples	
				As member of K	Outside class definitions
Logical negation (NOT)	`!a` `not a`[b]	Yes	Yes	`R K::operator !();`	`R operator !(K a);`
Logical AND	`a && b` `a and b`[b]	Yes	Yes	`R K::operator &&(S b);`	`R operator &&(K a, S b);`
Logical OR	`a || b` `a or b`[b]	Yes	Yes	`R K::operator ||(S b);`	`R operator ||(K a, S b);`

Bitwise operators

Operator name	Syntax	Can overload	Included in C	Prototype examples	
				As member of K	Outside class definitions
Bitwise NOT	`~a` `compl a`[b]	Yes	Yes	`R K::operator ~();`	`R operator ~(K a);`
Bitwise AND	`a & b` `a bitand b`[b]	Yes	Yes	`R K::operator &(S b);`	`R operator &(K a, S b);`
Bitwise OR	`a | b` `a bitor b`[b]	Yes	Yes	`R K::operator |(S b);`	`R operator |(K a, S b);`
Bitwise XOR	`a ^ b` `a xor b`[b]	Yes	Yes	`R K::operator ^(S b);`	`R operator ^(K a, S b);`
Bitwise left shift[c]	`a << b`	Yes	Yes	`R K::operator <<(S b);`	`R operator <<(K a, S b);`
Bitwise right shift[c][d]	`a >> b`	Yes	Yes	`R K::operator >>(S b);`	`R operator >>(K a, S b);`

Compound assignment operators

Operator name	Syntax	Meaning	Can overload	Included in C	Prototype examples	
					As member of K	Outside class definitions
Addition assignment	`a += b`	`a = a + b`	Yes	Yes	`R& K::operator +=(S b);`	`R& operator +=(K a, S b);`
Subtraction assignment	`a -= b`	`a = a - b`	Yes	Yes	`R& K::operator -=(S b);`	`R& operator -=(K a, S b);`
Multiplication assignment	`a *= b`	`a = a * b`	Yes	Yes	`R& K::operator *=(S b);`	`R& operator *=(K a, S b);`
Division assignment	`a /= b`	`a = a / b`	Yes	Yes	`R& K::operator /=(S b);`	`R& operator /=(K a, S b);`
Modulo assignment	`a %= b`	`a = a % b`	Yes	Yes	`R& K::operator %=(S b);`	`R& operator %=(K a, S b);`
Bitwise AND assignment	`a &= b` `a and_eq b`[b]	`a = a & b`	Yes	Yes	`R& K::operator &=(S b);`	`R& operator &=(K a, S b);`
Bitwise OR assignment	`a |= b` `a or_eq b`[b]	`a = a | b`	Yes	Yes	`R& K::operator |=(S b);`	`R& operator |=(K a, S b);`
Bitwise XOR assignment	`a ^= b` `a xor_eq b`[b]	`a = a ^ b`	Yes	Yes	`R& K::operator ^=(S b);`	`R& operator ^=(K a, S b);`
Bitwise left shift assignment	`a <<= b`	`a = a << b`	Yes	Yes	`R& K::operator <<=(S b);`	`R& operator <<=(K a, S b);`
Bitwise right shift assignment[d]	`a >>= b`	`a = a >> b`	Yes	Yes	`R& K::operator >>=(S b);`	`R& operator >>=(K a, S b);`

Member and pointer operators

Operator name	Syntax	Can overload	Included in C	Prototype examples	
				As member of K	Outside class definitions
Array subscript	`a[b]`	Yes	Yes	`R& K::operator [](S b);`	N/A
Indirection ("object pointed to by *a*")	`*a`	Yes	Yes	`R& K::operator *();`	`R& operator *(K a);`
Address ("address of *a*")	`&a`	Yes	Yes	`R K::operator &();`	`R operator &(K a);`
Structure dereference ("member *b* of object pointed to by *a*")	`a->b`	Yes	Yes	`R* K::operator ->();`[e]	N/A
Structure reference ("member *b* of object *a*")	`a.b`	No	Yes	N/A	
Member pointed to by *b* of object pointed to by *a*[f]	`a->*b`	Yes	No	`R& K::operator ->*(S b);`	`R& operator ->*(K a, S b);`
Member pointed to by *b* of object *a*	`a.*b`	No	No	N/A	

Other operators

Operator name	Syntax	Can overload	Included in C	Prototype examples	
				As member of K	Outside class definitions
Function call *See Function object.*	`a(a1, a2)`	Yes	Yes	`R K::operator ()(S a, T b, ...);`	N/A
Comma	`a, b`	Yes	Yes	`R K::operator ,(S b);`	`R operator ,(K a, S b);`
Ternary conditional	`a ? b : c`	No	Yes	N/A	
Scope resolution	`a::b`	No	No	N/A	
User-defined literals[g] *since C++11*	`"a"_b`	Yes	No	N/A	`R operator "" _b(T a)`
Size-of	`sizeof (a)`[h] `sizeof (type)`	No	Yes	N/A	
Size of parameter pack *since C++11*	`sizeof...(Args)`	No	No	N/A	

Align-of *since C++11*	`alignof (type)` or `_Alignof (type)`[i]	No	Yes	N/A	
Type identification	`typeid (a)` `typeid (type)`	No	No	N/A	
Conversion (C-style cast)	`(type) a` `type(a)`	Yes	Yes	`K::operator R();`	N/A
				Note: for user-defined conversions, the return type implicitly and necessarily matches the operator name.	
static_cast conversion	`static_cast<type>(a)`	No	No	N/A	
dynamic_cast conversion	`dynamic_cast<type>(a)`	No	No	N/A	
const_cast conversion	`const_cast<type>(a)`	No	No	N/A	
reinterpret_cast conversion	`reinterpret_cast<type>(a)`	No	No	N/A	
Allocate storage	`new type`	Yes	No	`void* K::operator new(size_t x);`	`void* operator new(size_t x);`
Allocate storage (array)	`new type[n]`	Yes	No	`void* K::operator new[](size_t a);`	`void* operator new[](size_t a);`
Deallocate storage (*delete* returns *void* so it isn't strictly speaking an operator)	`delete a`	Yes	No	`void K::operator delete(void *a);`	`void operator delete(void *a);`
Deallocate storage (array)	`delete[] a`	Yes	No	`void K::operator delete[](void *a);`	`void operator delete[](void *a);`
Exception check *since C++11*	`noexcept(a)`	No	No	N/A	

Notes:

1. **^** The modulus operator works just with integer operands, for floating point numbers a library function must be used instead (like `fmod`).
2. ^ ***a b c d e f g h i j k*** Requires `iso646.h` in C. See C++ operator synonyms
3. ^ ***a b*** In the context of iostreams, writers often will refer to << and >> as the “put-to” or "stream insertion" and “get-from” or "stream extraction" operators, respectively.
4. ^ ***a b*** According to the C99 standard, the right shift of a negative number is implementation defined. Most implementations, e.g., the GCC,[1] use an arithmetic shift (i.e., sign extension), but a logical shift is possible.

5. ^ The return type of `operator->()` must be a type for which the `->` operation can be applied, such as a pointer type. If `x` is of type `C` where `C` overloads `operator->()`, `x->y` gets expanded to `x.operator->()->y`.
6. ^ Meyers, Scott (Oct 1999), *Implementing operator->* for Smart Pointers* (http://aristeia.com/Papers/DDJ_Oct_1999.pdf) (PDF), *Dr.Dobbs* (Aristeia).
7. ^ About C++11 User-defined literals (http://en.cppreference.com/w/cpp/language/user_literal)
8. ^ The parentheses are not necessary when taking the size of a value, only when taking the size of a type. However, they are usually used regardless.
9. ^ C++ defines `alignof` operator, whereas C defines `_Alignof`. Both operators have the same semantics.

Operator precedence

The following is a table that lists the precedence and associativity of all the operators in the C and C++ languages (when the operators also exist in Java, Perl, PHP and many other recent languages, the precedence is the same as that given). Operators are listed top to bottom, in descending precedence. Descending precedence refers to the priority of evaluation. Considering an expression, an operator which is listed on some row will be evaluated prior to any operator that is listed on a row further below it. 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. An operator's precedence is unaffected by overloading.

The syntax of expressions in C and C++ is specified by a phrase structure grammar.[2] The table given here has been inferred from the grammar. For the ISO C 1999 standard, section 6.5.6 note 71 states that the C grammar provided by the specification defines the precedence of the C operators, and also states that the operator precedence resulting from the grammar closely follows the specification's section ordering:

"The [C] syntax [i.e., grammar] specifies the precedence of operators in the evaluation of an expression, which is the same as the order of the major subclauses of this subclause, highest precedence first."[3]

A precedence table, while mostly adequate, cannot resolve a few details. In particular, note that the ternary operator allows any arbitrary expression as its middle operand, despite being listed as having higher precedence than the assignment and comma operators. Thus `a ? b , c : d` is interpreted as `a ? (b, c) : d`, and not as the meaningless `(a ? b), (c : d)`. Also, note that the immediate, unparenthesized result of a C cast expression cannot be the operand of `sizeof`. Therefore, `sizeof (int) * x` is interpreted as `(sizeof(int)) * x` and not `sizeof ((int) *x)`.

Precedence	Operator	Description	Associativity
1 **highest**	`::`	Scope resolution (C++ only)	None
2	`++`	Suffix increment	Left-to-right
	`--`	Suffix decrement	
	`()`	Function call	
	`[]`	Array subscripting	
	`.`	Element selection by reference	
	`->`	Element selection through pointer	
	`typeid()`		

		Run-time type information (C++ only) (see typeid)	
	`const_cast`	Type cast (C++ only) (see const_cast)	
	`dynamic_cast`	Type cast (C++ only) (see dynamic_cast)	
	`reinterpret_cast`	Type cast (C++ only) (see reinterpret_cast)	
	`static_cast`	Type cast (C++ only) (see static_cast)	
3	`++`	Prefix increment	Right-to-left
	`--`	Prefix decrement	
	`+`	Unary plus	
	`-`	Unary minus	
	`!`	Logical NOT	
	`~`	Bitwise NOT (One's Complement)	
	`(`*`type`*`)`	Type cast	
	`*`	Indirection (dereference)	
	`&`	Address-of	
	`sizeof`	Size-of	
	`new`, `new[]`	Dynamic memory allocation (C++ only)	
	`delete`, `delete[]`	Dynamic memory deallocation (C++ only)	
4	`.*`	Pointer to member (C++ only)	Left-to-right
	`->*`	Pointer to member (C++ only)	
5	`*`	Multiplication	Left-to-right
	`/`	Division	
	`%`	Modulo (remainder)	
6	`+`	Addition	Left-to-right
	`-`	Subtraction	
7	`<<`	Bitwise left shift	Left-to-right
	`>>`	Bitwise right shift	
8	`<`	Less than	Left-to-right
	`<=`	Less than or equal to	
	`>`	Greater than	
	`>=`	Greater than or equal to	
9	`==`	Equal to	Left-to-right
	`!=`	Not equal to	
10	`&`	Bitwise AND	Left-to-right
11	`^`	Bitwise XOR (exclusive or)	Left-to-right
12	`|`	Bitwise OR (inclusive or)	Left-to-right
13	`&&`	Logical AND	Left-to-right
14	`||`	Logical OR	Left-to-right
15	`?:`	Ternary conditional (see ?:)	Right-to-left
	`=`	Direct assignment	Right-to-left
	`+=`	Assignment by sum	
	`-=`	Assignment by difference	

16	`*=`
	`/=`
	`%=`
	`<<=`
	`>>=`
	`&=`
	`^=`
	`|=`
		Assignment by product
		Assignment by quotient
		Assignment by remainder
		Assignment by bitwise left shift
		Assignment by bitwise right shift
		Assignment by bitwise AND
		Assignment by bitwise XOR
		Assignment by bitwise OR	

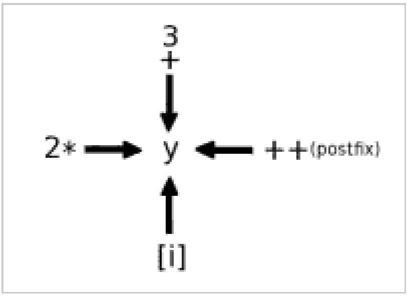
17	`throw`	Throw operator (exceptions throwing, C++ only)	Right-to-left
18 **lowest**	`,`	Comma	Left-to-right

[4]

Notes

The precedence table determines the order of binding in chained expressions, when it is not expressly specified by parentheses.

- For example, `++x*3` is ambiguous without some precedence rule(s). The precedence table tells us that: `x` is 'bound' more tightly to `++` than to `*`, so that whatever `++` does (now or later—see below), it does it ONLY to `x` (and not to `x*3`); it is equivalent to (`++x`, `x*3`).
- Similarly, with `3*x++`, where though the post-fix `++` is designed to act AFTER the entire expression is evaluated, the precedence table makes it clear that ONLY `x` gets incremented (and NOT `3*x`). In fact, the expression (`tmp=x++`, `3*tmp`) is evaluated with `tmp` being a temporary value. It is functionally equivalent to something like (`tmp=3*x`, `++x`, `tmp`).



Precedence and bindings

- Abstracting the issue of precedence or binding, consider the diagram above for the expression 3+2*y[i]++. The compiler's job is to resolve the diagram into an expression, one in which several unary operators (call them 3+(.), 2*(.), (.)++ and (.)[i]) are competing to bind to y. The order of precedence table resolves the final sub-expression they each act upon: (.)[i] acts only on y, (.)++ acts only on y[i], 2*(.) acts only on y[i]++ and 3+(.) acts 'only' on 2*((y[i])++). It is important to note that WHAT sub-expression gets acted on by each operator is clear from the precedence table but WHEN each operator acts is not resolved by the precedence table; in this example, the (.)++ operator acts only on y[i] by the precedence rules but binding levels alone do not indicate the timing of the Suffix ++ (the (.)++ operator acts only after y[i] is evaluated in the expression).

Many of the operators containing multi-character sequences are given "names" built from the operator name of each character. For example, `+=` and `-=` are often called *plus equal(s)* and *minus equal(s)*, instead of the more verbose "assignment by addition" and "assignment by subtraction". The binding of operators in C and C++ is specified (in the corresponding Standards) by a factored language grammar, rather than a precedence table. This creates some subtle conflicts. For example, in C, the syntax for a conditional expression is:

```
logical-OR-expression ? expression : conditional-expression
```

while in C++ it is:

```
logical-OR-expression ? expression : assignment-expression
```

Hence, the expression:

```
e = a < d ? a++ : a = d
```

is parsed differently in the two languages. In C, this expression is a syntax error, but many compilers parse it as:

```
e = ((a < d ? a++ : a) = d)
```

which is a semantic error, since the result of the conditional-expression (which might be *a++*) is not an lvalue. In C++, it is parsed as:

```
e = (a < d ? a++ : (a = d))
```

which is a valid expression.

Criticism of bitwise and equality operators precedence

The precedence of the bitwise logical operators has been criticized.[5] Conceptually, & and | are arithmetic operators like + and *.

The expression `a & b == 7` is syntactically parsed as `a & (b == 7)` whereas the expression `a + b == 7` is parsed as `(a + b) == 7`. This requires parentheses to be used more often than they otherwise would.

Moreover, in C++ (and later versions of C) equality operations yield bool type values which are conceptually a single bit (1 or 0) and as such do not properly belong in "bitwise" operations.

C++ operator synonyms

C++ defines[6] keywords to act as aliases for a number of operators: `and` (`&&`), `bitand` (`&`), `and_eq` (`&=`), `or` (`||`), `bitor` (`|`), `or_eq` (`|=`), `xor` (`^`), `xor_eq` (`^=`), `not` (`!`), `not_eq` (`!=`), and `compl` (`~`). These can be used exactly the same way as the punctuation symbols they replace, as they are not the same operator under a different name, but rather simple token replacements for the *name* (character string) of the respective operator. This means that the expressions `(a > 0 and flag)` and `(a > 0 && flag)` have identical meanings. It also mean that, for example, the `bitand` keyword may be used to replace not only the *bitwise-and* operator but also the *address-of* operator, and it can even be used to specify reference types (e.g., `int bitand ref = n`). The ISO C specification makes allowance for these keywords as preprocessor macros in the header file `iso646.h`. For compatibility with C, C++ provides the header `ciso646`, inclusion of which has no effect.

See also

- Order of operations
- Operator associativity
- Sequence point
- Bitwise operations in C
- Bit manipulation
- Bitwise operation
- Logical operator
- Boolean algebra (logic)
- Table of logic symbols
- C++

References

1. ^ "Integers implementation" (http://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/Integers-implementation.html#Integers-implementation), *GCC 4.3.3*, GNU.
2. ^ *ISO/IEC 9899:201x Programming Languages - C*. ISO/IEC JTC1/SC22/WG14 (http://www.open-std.org/jtc1/sc22/wg14/) — The C Standards Committee. 19 December 2011. p. 465.
3. ^ *the ISO C 1999 standard, section 6.5.6 note 71* (Technical report). ISO. 1999.
4. ^ http://msdn.microsoft.com/en-us/library/126fe14k(v=vs.80).aspx
5. ^ *C history § Neonatal C* (http://cm.bell-labs.com/cm/cs/who/dmr/chist.html), Bell labs.
6. ^ *ISO/IEC 14882:1998(E) Programming Language C++*. ISO/IEC JTC1/SC22/WG21 (http://www.open-std.org/jtc1/sc22/wg21/) — The C++ Standards Committee. 1 September 1998. pp. 40–41.

External links

- "Operators" (http://cppreference.com/wiki/language/operators), *C++ reference* (wiki).
- "Prefix vs. Suffix operators in C and C++" (http://msdn.microsoft.com/en-us/library/e1e3921c(VS.80).aspx), (Developer network), Microsoft Missing or empty `|title=` (help).

Retrieved from "http://en.wikipedia.org/w/index.php?title=Operators_in_C_and_C%2B%2B&oldid=629540030"

Categories: C programming language | C++ | Operators (programming)