WIKIPEDIA

Operators in C and C++

This is a list of operators in the C and C++ programming languages. All the operators listed exist in C++; the column "Included in C", states 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 C-family languages such as C#, D, Java, Perl, and PHP with the same precedence, associativity, and semantics.

Contents

Table

For the purposes of these tables, `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.

Arithmetic operators

All arithmetic operators exists in C and C++ and can be overloaded in C++.

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

Comparison operators/relational operators

All comparison operators can be overloaded in C++.

Operator name	Syntax	Included in C	Prototype examples	
			As member of K	Outside class definitions
Equal to	`a == b`	Yes	`bool K::operator ==(S const& b) const;`	`bool operator ==(K const& a, S const& b);`
Not equal to	`a != b` `a not_eq b`[b]	Yes	`bool K::operator !=(S const& b) const;`	`bool operator !=(K const& a, S const& b);`
Greater than	`a > b`	Yes	`bool K::operator >(S const& b) const;`	`bool operator >(K const& a, S const& b);`
Less than	`a < b`	Yes	`bool K::operator <(S const& b) const;`	`bool operator <(K const& a, S const& b);`
Greater than or equal to	`a >= b`	Yes	`bool K::operator >=(S const& b) const;`	`bool operator >=(K const& a, S const& b);`
Less than or equal to	`a <= b`	Yes	`bool K::operator <=(S const& b) const;`	`bool operator <=(K const& a, S const& b);`
Three-way comparison[c]	`a <=> b`	No	`auto K::operator <=>(const S &b);`	`auto operator <=>(const K &a, const S &b);`
			The operator has a total of 3 possible return types: `std::weak_ordering`, `std::strong_ordering` and `std::partial_ordering` to which they all are convertible to.	

Logical operators

All logical operators exist in C and C++ and can be overloaded in C++, albeit the overloading of the logical AND and logical OR is discouraged, because as overloaded operators they behave as ordinary function calls, which means that *both* of their operands are evaluated, so they lose their well-used and expected short-circuit evaluation property.[1]

Operator name	Syntax	C++ prototype examples	
		As member of K	Outside class definitions
Logical negation (NOT)	`!a` `not a`[b]	`bool K::operator !();`	`bool operator !(K a);`
Logical AND	`a && b` `a and b`[b]	`bool K::operator &&(S b);`	`bool operator &&(K a, S b);`
Logical OR	`a || b` `a or b`[b]	`bool K::operator ||(S b);`	`bool operator ||(K a, S b);`

Bitwise operators

All bitwise operators exist in C and C++ and can be overloaded in C++.

<table>
<tr><th rowspan="2">Operator name</th><th rowspan="2">Syntax</th><th colspan="2">Prototype examples</th></tr>
<tr><th>As member of K</th><th>Outside class definitions</th></tr>
<tr><td>Bitwise NOT</td><td>~a
compl a[b]</td><td>R K::operator ~();</td><td>R operator ~(K a);</td></tr>
<tr><td>Bitwise AND</td><td>a & b
a bitand b[b]</td><td>R K::operator &(S b);</td><td>R operator &(K a, S b);</td></tr>
<tr><td>Bitwise OR</td><td>a | b
a bitor b[b]</td><td>R K::operator |(S b);</td><td>R operator |(K a, S b);</td></tr>
<tr><td>Bitwise XOR</td><td>a ^ b
a xor b[b]</td><td>R K::operator ^(S b);</td><td>R operator ^(K a, S b);</td></tr>
<tr><td>Bitwise left shift[d]</td><td>a << b</td><td>R K::operator <<(S b);</td><td>R operator <<(K a, S b);</td></tr>
<tr><td>Bitwise right shift[d][e]</td><td>a >> b</td><td>R K::operator >>(S b);</td><td>R operator >>(K a, S b);</td></tr>
</table>

Assignment operators

All assignment expressions exist in C and C++ and can be overloaded in C++.

For the given operators the semantic of the built-in combined assignment expression `a ⊙= b` is equivalent to `a = a ⊙ b`, except that `a` is evaluated only once.

<table>
<tr><th rowspan="2">Operator name</th><th rowspan="2">Syntax</th><th colspan="2">C++ prototype examples</th></tr>
<tr><th>As member of K</th><th>Outside class definitions</th></tr>
<tr><td>Direct assignment</td><td>a = b</td><td>R& K::operator =(S b);</td><td>N/A</td></tr>
<tr><td>Addition assignment</td><td>a += b</td><td>R& K::operator +=(S b);</td><td>R& operator +=(K& a, S b);</td></tr>
<tr><td>Subtraction assignment</td><td>a -= b</td><td>R& K::operator -=(S b);</td><td>R& operator -=(K& a, S b);</td></tr>
<tr><td>Multiplication assignment</td><td>a *= b</td><td>R& K::operator *=(S b);</td><td>R& operator *=(K& a, S b);</td></tr>
<tr><td>Division assignment</td><td>a /= b</td><td>R& K::operator /=(S b);</td><td>R& operator /=(K& a, S b);</td></tr>
<tr><td>Modulo assignment</td><td>a %= b</td><td>R& K::operator %=(S b);</td><td>R& operator %=(K& a, S b);</td></tr>
<tr><td>Bitwise AND assignment</td><td>a &= b
a and_eq b[b]</td><td>R& K::operator &=(S b);</td><td>R& operator &=(K& a, S b);</td></tr>
<tr><td>Bitwise OR assignment</td><td>a |= b
a or_eq b[b]</td><td>R& K::operator |=(S b);</td><td>R& operator |=(K& a, S b);</td></tr>
<tr><td>Bitwise XOR assignment</td><td>a ^= b
a xor_eq b[b]</td><td>R& K::operator ^=(S b);</td><td>R& operator ^=(K& a, S b);</td></tr>
<tr><td>Bitwise left shift assignment</td><td>a <<= b</td><td>R& K::operator <<=(S b);</td><td>R& operator <<=(K& a, S b);</td></tr>
<tr><td>Bitwise right shift assignment[e]</td><td>a >>= b</td><td>R& K::operator >>=(S b);</td><td>R& operator >>=(K& a, S b);</td></tr>
</table>

Member and pointer operators

Operator name	Syntax	Can overload in C++	Included in C	C++ prototype examples	
				As member of K	Outside class definitions
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-of ("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 ->();`[f]	N/A
Structure reference ("member *b* of object *a*")	`a.b`	No	Yes	N/A	
Member selected by pointer-to-member *b* of object pointed to by *a*[g]	`a->*b`	Yes	No	`R& K::operator ->*(S b);`	`R& operator ->*(K a, S b);`
Member of object *a* selected by pointer-to-member *b*	`a.*b`	No	No	N/A	

Other operators

Operator name	Syntax	Can overload in C++	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[h] ***since C++11***	"a"_b	Yes	No	N/A	R operator "" _b(T a)
Sizeof	sizeof(a)[i] sizeof(*type*)	No	Yes	N/A	
Size of parameter pack ***since C++11***	sizeof...(Args)	No	No	N/A	
Alignof ***since C++11***	alignof(*type*) or _Alignof(*type*)[j]	No	Yes	N/A	
Type identification	typeid(a) typeid(*type*)	No	No	N/A	
Conversion (C-style cast)	(*type*)a	Yes	Yes	K::operator R();[3]	N/A
Conversion	*type*(a)	No	No	Note: behaves like const_cast/static_cast/reinterpret_cast[4]	
static_cast conversion	static_cast<*type*>(a)	Yes	No	K::operator R(); explicit K::operator R(); *since C++11* Note: for user-defined conversions, the return type implicitly and necessarily matches the operator name.	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	void* operator new[](size_t a);

				`new[](size_t a);`	
Deallocate storage	`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. Requires `iso646.h` in C. See C++ operator synonyms
3. About C++20 three-way comparison (https://en.cppreference.com/w/cpp/language/operator_comparison#Three-way_comparison)
4. 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.
5. According to the C99 standard, the right shift of a negative number is implementation defined. Most implementations, e.g., the GCC,[2] use an arithmetic shift (i.e., sign extension), but a logical shift is possible.
6. 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`.
7. Meyers, Scott (October 1999), "Implementing operator->* for Smart Pointers" (http://aristeia.com/Papers/DDJ_Oct_1999.pdf) (PDF), *Dr. Dobb's Journal*, Aristeia.
8. About C++11 User-defined literals (http://en.cppreference.com/w/cpp/language/user_literal)
9. 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.
10. 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 the grouping of operators and operands. Considering an expression, an operator which is listed on some row will be grouped 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 grouped 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.[5] 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."[6]

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)`. So, the expression in the middle of the conditional operator (between **?** and **:**) is parsed as if parenthesized. 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	++	Postfix increment	Left-to-right
	--	Postfix 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	Sizeof	
	_Alignof	Alignment requirement (since C11)	
	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	<=>	Three-way comparison (Introduced in C++20 -	Left-to-right

		C++ only)	
9	< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left-to-right
10	== !=	Equal to Not equal to	Left-to-right
11	&	Bitwise AND	Left-to-right
12	^	Bitwise XOR (exclusive or)	Left-to-right
13	|	Bitwise OR (inclusive or)	Left-to-right
14	&&	Logical AND	Left-to-right
15	||	Logical OR	Left-to-right
16	?: = += -= *= /= %= <<= >>= &= ^= |= throw	Ternary conditional (see ?:) Direct assignment Assignment by sum Assignment by difference 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 Throw operator (exceptions throwing, C++ only)	Right-to-left
17 **lowest**	,	Comma	Left-to-right

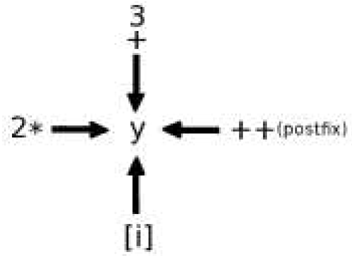
[7][8][9]

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 postfix ++ (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, because the syntax for an assignment expression in C is:

```
unary-expression '=' assignment-expression
```

In C++, it is parsed as:

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

which is a valid expression.[10][11]

If you want to use comma-as-operator within a single function argument, variable assignment, or other comma-separated list, you need to use parentheses,[12][13] e.g.:

```
int a = 1, b = 2, weirdVariable = (++a, b), d = 4;
```

Criticism of bitwise and equality operators precedence

The precedence of the bitwise logical operators has been criticized.[14] 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.

Historically, there was no syntactic distinction between the bitwise and logical operators. In BCPL, B and early C, the operators `&& ||` didn't exist. Instead `& |` had different meaning depending on whether they are used in a 'truth-value context' (i.e. when a Boolean value was expected, for example in `if (a==b & c) {...}` it behaved as a logical operator, but in `c = a & b` it behaved as a bitwise one). It was retained so as to keep backward compatibility with existing installations.[15]

Moreover, in C++ (and later versions of C) equality operations, with the exception of the three-way comparison operator, 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[16] certain keywords to act as aliases for a number of operators:

Keyword	Operator
`and`	`&&`
`and_eq`	`&=`
`bitand`	`&`
`bitor`	`|`
`compl`	`~`
`not`	`!`
`not_eq`	`!=`
`or`	`||`
`or_eq`	`|=`
`xor`	`^`
`xor_eq`	`^=`

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 not flag)` and `(a > 0 && !flag)` have identical meanings. It also means 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`, the inclusion of which has no effect.

See also

- Bitwise operations in C
- Bit manipulation
- Logical operator
- Boolean algebra (logic)
- Table of logic symbols
- Digraphs and trigraphs in C and in C++

References

1. "Standard C++" (https://isocpp.org/wiki/faq/operator-overloading).
2. "Integers implementation", *GCC 4.3.3* (https://gcc.gnu.org/onlinedocs/gcc-4.3.3/gcc/Integers-implementation.html#Integers-implementation), GNU.
3. "user-defined conversion" (https://en.cppreference.com/w/cpp/language/cast_operator). Retrieved 5 April 2020.
4. Explicit type conversion (https://en.cppreference.com/w/cpp/language/explicit_cast) in C++
5. *ISO/IEC 9899:201x Programming Languages - C*. open-std.org – The C Standards Committee. 19 December 2011. p. 465.
6. *the ISO C 1999 standard, section 6.5.6 note 71* (Technical report). ISO. 1999.
7. "C Operator Precedence - cppreference.com" (https://en.cppreference.com/w/c/language/operator_precedence). *en.cppreference.com*. Retrieved 16 July 2019.
8. "C++ Built-in Operators, Precedence and Associativity" (https://docs.microsoft.com/en-US/cpp/cpp/cpp-built-in-operators-precedence-and-associativity). *docs.microsoft.com*. Retrieved 11 May 2020.
9. "C++ Operator Precedence - cppreference.com" (https://en.cppreference.com/w/cpp/language/operator_precedence). *en.cppreference.com*. Retrieved 16 July 2019.
10. "C Operator Precedence - cppreference.com" (https://en.cppreference.com/w/c/language/operator_precedence). *en.cppreference.com*. Retrieved 10 April 2020.
11. "Does the C/C++ ternary operator actually have the same precedence as assignment operators?" (https://stackoverflow.com/questions/13515434/does-the-c-c-ternary-operator-actually-have-the-same-precedence-as-assignment/13515505). *Stack Overflow*. Retrieved 22 September 2019.
12. "Other operators - cppreference.com" (https://en.cppreference.com/w/c/language/operator_other). *en.cppreference.com*. Retrieved 10 April 2020.
13. "c++ - How does the Comma Operator work" (https://stackoverflow.com/questions/54142/how-does-the-comma-operator-work/). *Stack Overflow*. Retrieved 1 April 2020.
14. *C history § Neonatal C* (https://www.bell-labs.com/usr/dmr/www/chist.html), Bell labs.

15. "Re^10: next unless condition" (https://www.perlmonks.org/?node_id=1159769). *www.perlmonks.org*. Retrieved 23 March 2018.
16. *ISO/IEC 14882:1998(E) Programming Language C++*. open-std.org – The C++ Standards Committee. 1 September 1998. pp. 40–41.

External links

- "Operators", *C++ reference* (https://en.cppreference.com/w/cpp/language/expressions#Operators) (wiki).
- C Operator Precedence (https://en.cppreference.com/w/c/language/operator_precedence)
- *Postfix Increment and Decrement Operators: ++ and --* (https://docs.microsoft.com/en-US/cpp/cpp/postfix-increment-and-decrement-operators-increment-and-decrement) (Developer network), Microsoft.

Retrieved from "https://en.wikipedia.org/w/index.php?title=Operators_in_C_and_C%2B%2B&oldid=1057903561"

This page was last edited on 30 November 2021, at 09:43 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.