Precedence: Consider the following expression:

3 + 4 * 5

Is the value of the expression **(3 + 4)*5 =35**?

Or, is the value of the expression **3 + (4 * 5)=23**?

Multiplication has *precedence* over addition and, hence, the value of the expression **3 + 4 * 5=23**.

The addition operator + and the multiplication operator * are *binary* operators: both operators require *two* operands. The addition operator looks left and right for the nearest operands and finds **3** on the left and **4** on the right. Similarly, the multiplication operator looks left and right for the nearest operands and finds **4** on the left and **5** on the right. Operand **4** is in contention. Operand **4** is simultaneously the right operand of the addition operator and the left operand of the multiplication operator. Both operators cannot be executed simultaneously. One operation must be performed first. The rules of precedence determine which operation is performed first and which operator is bound to operands in contention.

Name	Example	Precedence	Associativity
names, literals	**id 1.602e-19 '\n' "tom"**	17	n/a
scope resolution	*class-name* **::** *member*	16	left
scope resolution	*namespace-name* **::** *member*	16	left
global	**::** *name*	16	right
global	**::** *qualified-name*	16	right
member selection	*object* **.** *member*	15	left
member selection	*pointer* **->** *member*	15	left
subscripting	*pointer* **[** *expr* **]**	15	left
function call	*expr* **(** *expr-list* **)**	15	left
value construction	*type* **(** *expr-list* **)**	15	left
post increment	*lvalue* ++	15	left
post decrement	*lvalue* **--**	15	left
type identification	*typeid* **(** *type* **)**	15	left
run-time type identification	*typeid* **(** *expr* **)**	15	left
compile-time checked conversion	*dynamic-cast* **<** *type* **>** **(** *expr* **)**	15	left
unchecked conversion	*reinterpret-cast* **<** *type* **>** **(** *expr* **)**	15	left
const conversion	*const-cast* **<** *type* **>** **(** *expr* **)**	15	left
size of object	**sizeof** *expr*	14	left
size of type	**sizeof (** *type* **)**	14	left
pre increment	++ *lvalue*	14	right
pre decrement	**--** *lvalue*	14	right
complement	~ *expr*	14	right
not	**!** *expr*	14	right
unary minus	**-** *expr*	14	right
unary plus	+ *expr*	14	right
address of	**&** *lvalue*	14	right
dereference	* *expre*	14	right
create (allocate)	**new** *type*	14	right
create (allocate and initialize)	**new** *type* **(** *expr-list* **)**	14	right
create (place)	**new (** *expr-list***)** *type*	14	right
create (place and initialize)	**new (** *expr-list* **)** *type* **(** *expr-list* **)**	14	right
destroy (de-allocate)	**delete** *pointer*	14	right
destroy array	**delete[]** *pointer*	14	right
cast (type conversion)	**(***type***)** *expr*	14	right
member selection	*object* **.** * *pointer-to-member*	13	left

Name	Example	Precedence	Associativity
member selection	*pointer* **->** ***** *pointer-to-member*	13	left
multiply	*expr* ***** *expr*	12	left
divide	*expr* **/** *expr*	12	left
modulo (remainder)	*expr* **%** *expr*	12	left
add (plus)	*expr* **+** *expr*	11	left
subtract (minus)	*expr* **-** *expr*	11	left
shift left	*expr* **<<** *expr*	10	left
shift right	*expr* **>>** *expr*	10	left
less than	*expr* **<** *expr*	9	left
less than or equal	*expr* **<=** *expr*	9	left
greater than	*expr* **>** *expr*	9	left
greater than or equal	*expr* **>=** *expr*	9	left
equal	*expr* **==** *expr*	8	left
not equal	*expr* **!=** *expr*	8	left
bitwise AND	*expr* **&** *expr*	7	left
bitwise exclusive OR	*expr* **^** *expr*	6	left
bitwise inclusive OR	*expr* **|** *expr*	5	left
conditional expression	*expr* **?** *expr* **:** *expr*	4	right
simple assignment	*lvalue* **=** *expr*	3	right
multiply and assign	*lvalue* ***=** *expr*	3	right
divide and assign	*lvalue* **/=** *expr*	3	right
modulo and assign	*lvalue* **%=** *expr*	3	right
add and assign	*lvalue* **+=** *expr*	3	right
subtract and assign	*lvalue* **-=** *expr*	3	right
shift left and assign	*lvalue* **<<=** *expr*	3	right
shift right and assign	*lvalue* **>>=** *expr*	3	right
AND and assign	*lvalue* **&=** *expr*	3	right
inclusive OR and assign	*lvalue* **|=** *expr*	3	right
exclusive OR and assign	*lvalue* **^=** *expr*	3	right
throw exception	**throw** *expr*	2	right
comma (sequencing)	*expr* **,** *expr*	1	left

Associativity: When operators have the same precedence, associativity governs the order of evaluation. For example, **a + b + c + d** is evaluated **((a + b) +c) + d**. The addition operator associates to the left. Correctly parenthesized expressions that coerce the order of operations of a left-associative operator accumulate on the left.

Consider **a=b=c=d**. The expression **a=b=c=d** is evaluated **a=(b=(c=d))**. First **d** is assigned to **c**, then **c** is assigned to **b** and, finally, **b** is assign to **a**. The assignment operators associate to the right.