Precedence & associativity

- How would you evaluate the expression `17 - 8 * 2` ?

  Is it `17 - (8 * 2)`
  or `(17 - 8) * 2` ?

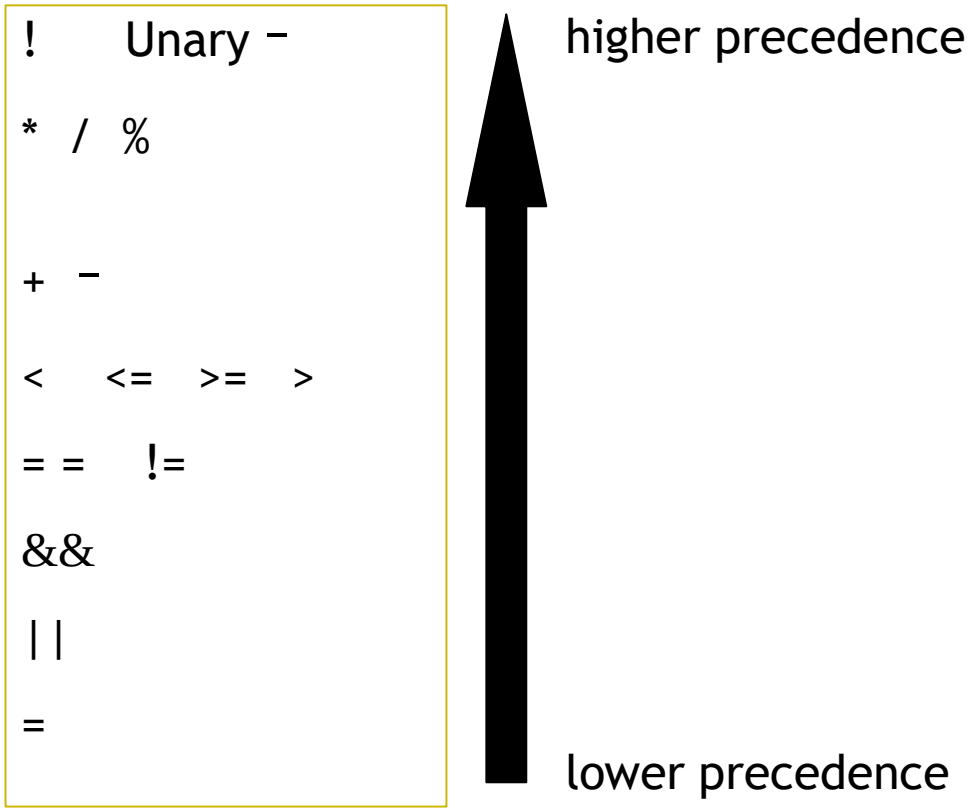
- These two forms give different results.
- We need rules!

Precedence & associativity

- When two operators compete for the same operand (e.g. in **17 - 8 * 2** the operators **-** and ***** compete for 8) the rules of precedence specify which operator wins.
  - The operator with the higher precedence wins

- If both competing operators have the same precedence, then the rules of associativity determine the winner.

Precedence & associativity



OPERATOR	TYPE	ASSOCIAVITY
( ) [ ] . ->		left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Unary Operator	right-to-left
* / %	Arithmetic Operator	left-to-right
+ -	Arithmetic Operator	left-to-right
<< >>	Shift Operator	left-to-right
< <= > >=	Relational Operator	left-to-right
== !=	Relational Operator	left-to-right
&	Bitwise AND Operator	left-to-right
^	Bitwise EX-OR Operator	left-to-right
|	Bitwise OR Operator	left-to-right
&&	Logical AND Operator	left-to-right
||	Logical OR Operator	left-to-right
? :	Ternary Conditional Operator	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Operator	right-to-left
,	Comma	left-to-right

Operator Precedence and Associativity in C

Operator precedence determines which operator is performed first in an expression with more than one operators with different precedence.

For example: Solve 10 + 20 * 30

10 + 20 * 30 is calculated as 10 + (20 * 30)
and not as (10 + 20) * 30

Operators Associativity is used when two operators of same precedence appear in an expression. Associativity can be either Left to Right or Right to Left.

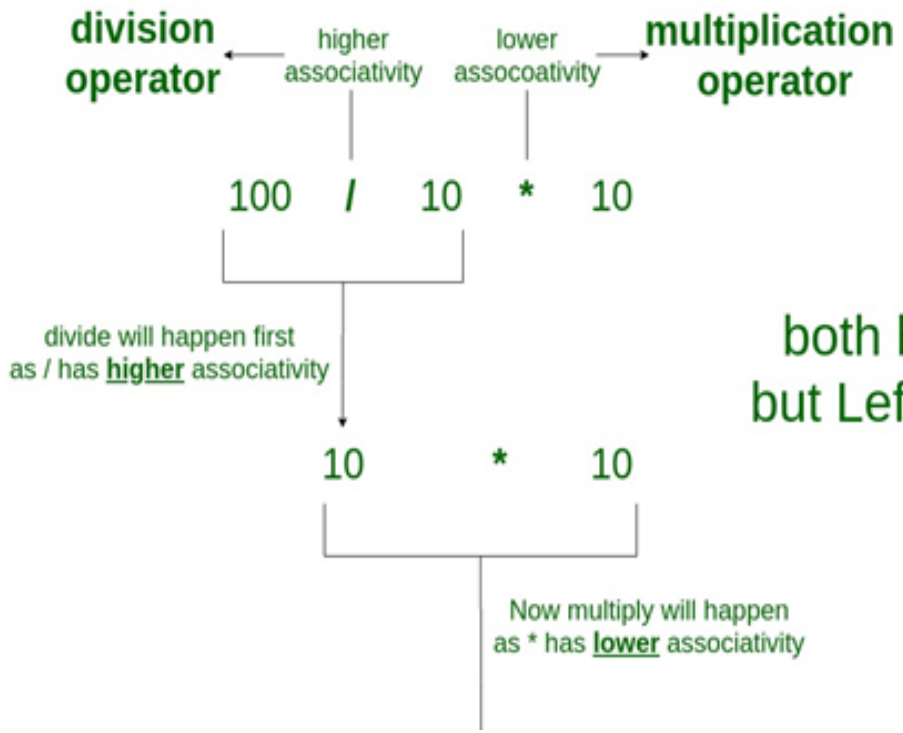
For example: '*' and '/' have same precedence and their associativity is Left to Right, so the expression "100 / 10 * 10" is treated as "(100 / 10) * 10".

Operators Precedence and Associativity are two characteristics of operators that determine the evaluation order of sub-expressions in absence of brackets

For example: Solve

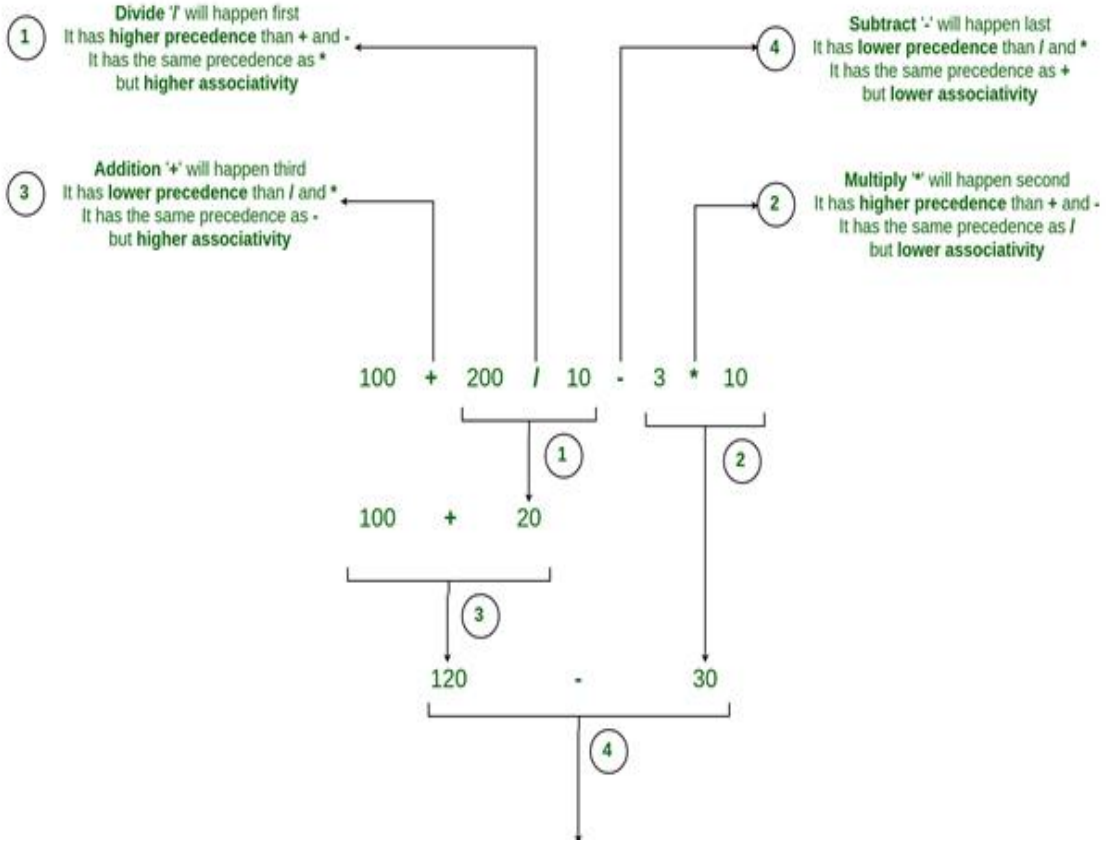
100 + 200 / 10 - 3 * 10

Operator Associativity



/ and ***
both have the same precedence
but Left to Right (**LTR**) associativity

Operator Precedence and Associativity



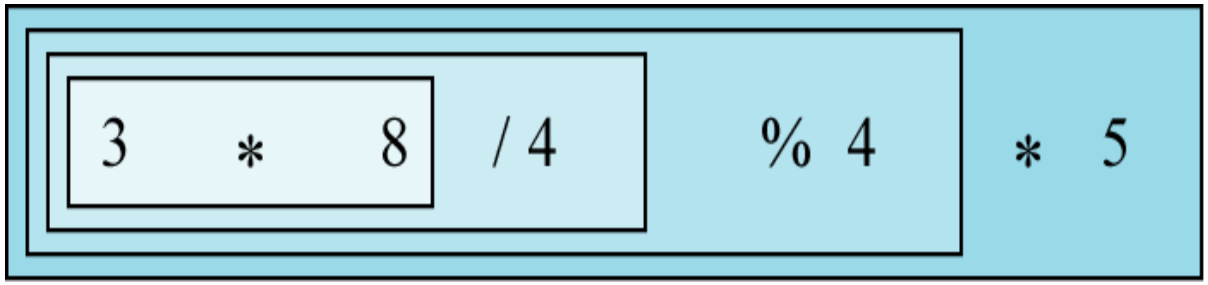
/ and ***
both have the same precedence
but Left to Right (**LTR**) associativity

+ and **-**
both have the same precedence
but Left to Right (**LTR**) associativity

/ and ***
have the higher precedence
than **+** and **-**

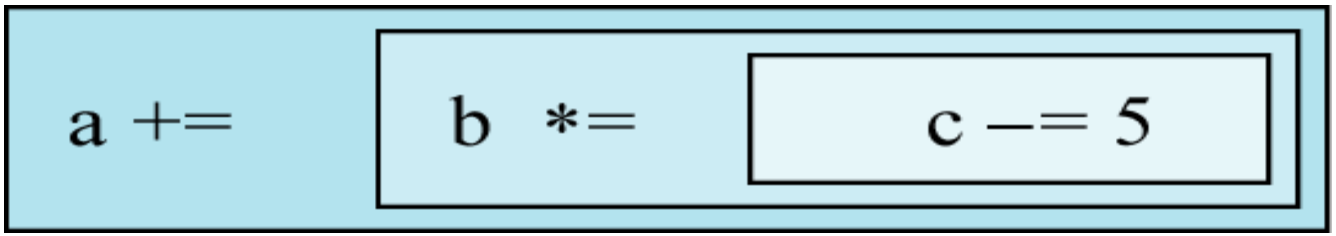
Example: **Left associativity**

3 * 8 / 4 % 4 * 5



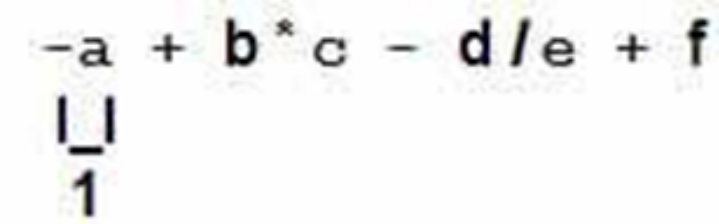
Example: Right associativity

a += b *= c-=5

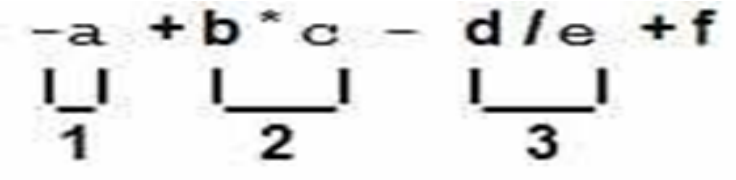


The steps to determine operator binding in an arithmetic expression are explained below with the help of the expression -a+ b * c – d /e + f.

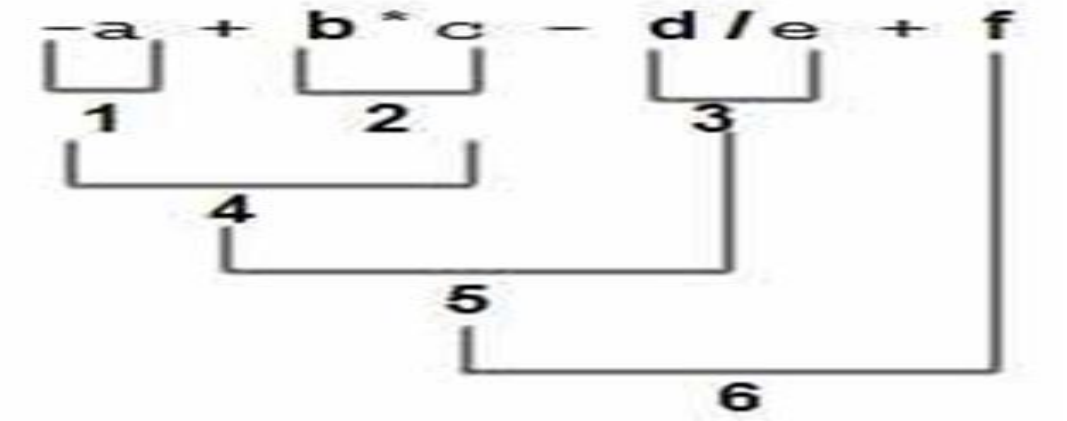
1. The unary operators (unary +, unary – , ++ and – -) have the highest precedence and right-to-left associativity. Thus, the given expression is first scanned from right to left and unary operators, if any, are bound to their operands. The order is indicated below the expression as follows:



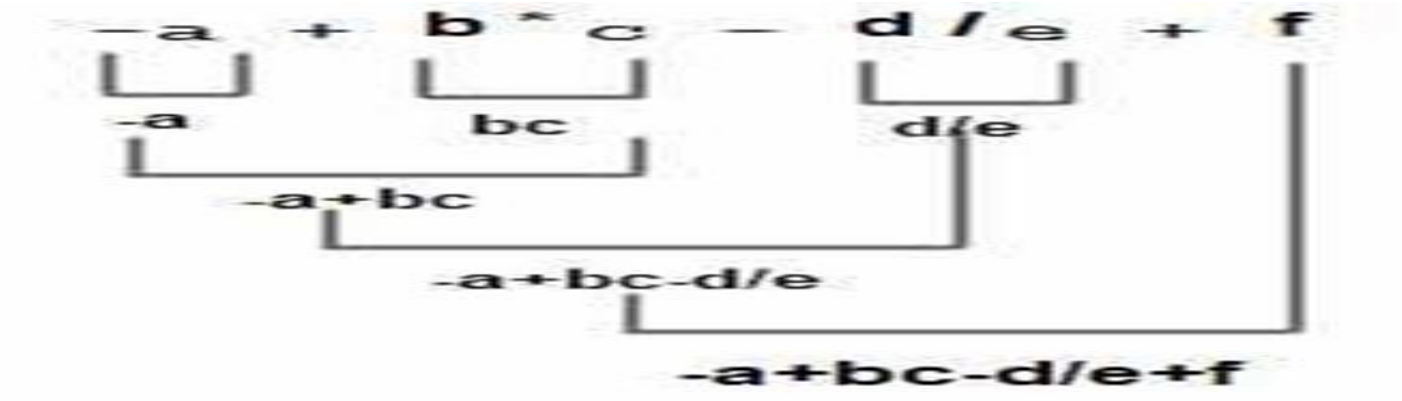
2. The multiplicative operators (*, I and %) have the next highest precedence and left to- right associativity. Thus, the expression is scanned from left-to-right and the multiplicative operators, if any, are bound to their operands as shown below. (Observe that after completion of the above step, sub-expressions -a, b * c and d / e will be operands for the remaining operator bindings.)



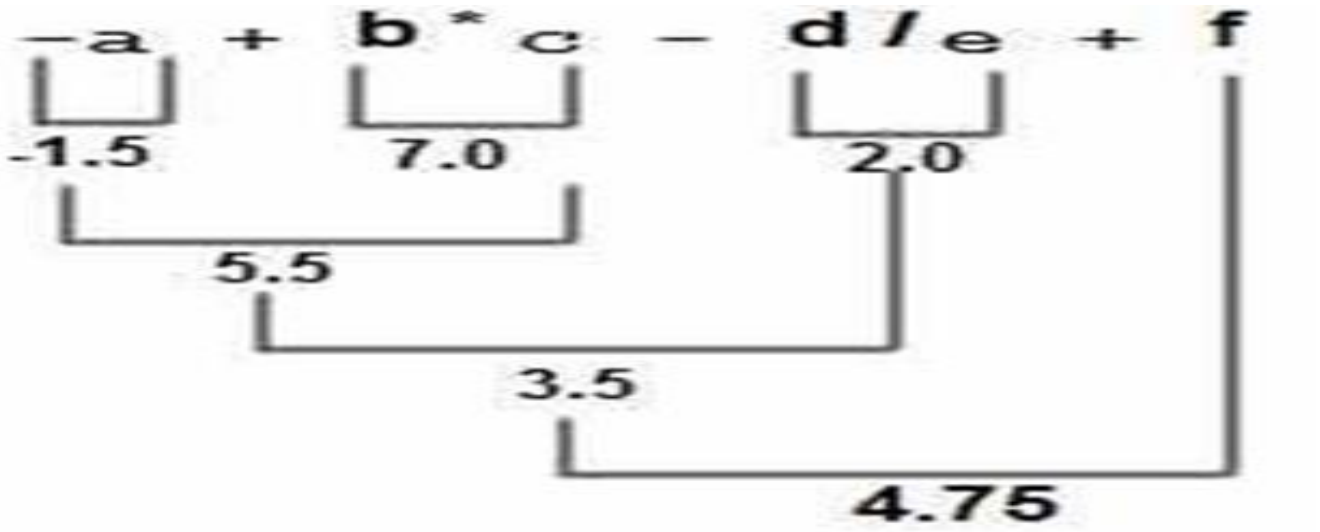
3. The additive operators (+ and -) have the next highest precedence and left-to-right associativity. Hence, the expression is scanned from left-to-right and the additive operators, if any, are bound to their operands as shown below. Observe that the operands for the first + operator are the sub-expressions -a and b * c. Similarly, the operands for the – operator are -a+ b * c and d /e.



Now we can write the mathematical equation for the given C expression by following the operator binding sequence as shown below:



Now the given expression can be evaluated, again by following the operator binding sequence, as shown below. Assume that the variables a, b, c, d, e and f are of type float and are initialized with values as a= 1. 5, b = 2. 0, c = 3. 5, d= 5.0, e = 2. 5 and f = 1. 25.



For example:

```
100 + (1 + 250 / 100) ** 3
    --> 100 + (1 + [250 / 100]) ** 3
    --> 100 + (1 + 2) ** 3
    --> 100 + ([1 + 2]) ** 3
    --> 100 + 3 ** 3
    --> 100 + [3 ** 3]
    --> 100 + 27
    --> 127
```

For example:

In the following, 25.0 ** 1 is not converted, and 1 / 3 is zero. ** has highest precedence than *,/,% operator and also has Left to Right associativity.

25.0 ** 1 / 2 * 3.5 ** (1 / 3)

--> [25.0 ** 1] / 2 * 3.5 ** (0)

--> 25.0 / 2 * 1.0

--> 25.0 / 2.0 * 1.0

--> 12.5*1.0

--> 12.5

- Examples:

Z = 10 + 9 * ((8 + 7) % 6) + 5 * 4 % 3 *2 + 1 ?

Not sure? Confused? then use parentheses in your code!

Sizeof() Operator

- C provides a unary operator named sizeof to find number of bytes needed to store an object.
- An expression of the form sizeof(object) returns an integer value that represents the number of bytes needed to store that object in memory.

- printf("%d",sizeof(int)); /* prints 2 */
  printf("%d",sizeof(char)); /* prints 1 */
  printf("%d",sizeof(float)); /* prints 4 */

```c
#include <stdio.h>
int main()
{
   int i = 6;
   int j;
   char c;
   float f;
   double d;
   printf("size of integer variable i = %d", sizeof(i));
   printf("\nsize of integer variable j = %d", sizeof(j));
   printf("\nsize of character variable = %d", sizeof(c));
   printf("\nsize of float variable = %d", sizeof(f));
   printf("\nsize of double variable = %d", sizeof(d));
   return 0;
}
```

Output

```
size of integer variable i = 2
size of integer variable j = 2
size of character variable = 1
size of float variable = 4
size of double variable = 8
```