The precedence of operators in C dictates the order in which the operators will be evolved in an expression. Associativity, on the other hand, defines the order in which the operators of the same precedence will be evaluated in an expression. Also, associativity can occur from either right to left or left to right.

**Use of the Operator Precedence and Associativity in C**

Precedence and Associativity are two of the characters that are used for operators in an expression for determining the order of sub-expressions when they do not have brackets.

**Operator Precedence**

Operator precedence helps us determine which of the operators in an expression must be evaluated first in case the expression consists of more than a single operator.

*Example,*

50 – 2 \* 15 is going to yield 20. It is because it gets evaluated as 50 – (2 \* 15), and not as (50 – 2) \* 15. The reason here is that subtraction (-) has lower precedence as compared to multiplication (\*).

**Operator Associativity**

We use associativity when two or more than two operators with the same precedence are present in the same expression.

*Example,*

The precedence of Division and Multiplication arithmetic operators is the same. So, let’s say we have an expression with us which is 6 \* 3 / 20. The evaluation of this expression would be (6 \* 3) / 20 because the associativity will be left to right for both the operators – multiplication and division. In a similar case, the calculation of 40 / 4 \* 5 would be (40 / 4) \* 5 because the associativity would be from right to left here as well.

**Summary of Operator Precedence and Associativity in C**

Here is how the operator precedence and associativity work in the C language:

|  |  |  |
| --- | --- | --- |
| **Operator** | **Description of Operator** | **Associativity** |
| . | Direct member selection | Left to right |
| -> | Indirect member selection | Left to right |
| [] | Array element reference | Left to right |
| () | Functional call | Left to right |
| ~ | Bitwise(1’s) complement | Right to left |
| ! | Logical negation | Right to left |
| – | Unary minus | Right to left |
| + | Unary plus | Right to left |
| — | Decrement | Right to left |
| ++ | Increment | Right to left |
| \* | Pointer reference | Right to left |
| & | Dereference (Address) | Right to left |
| (type) | Typecast (conversion) | Right to left |
| sizeof | Returns the size of an object | Right to left |
| % | Remainder | Left to right |
| / | Divide | Left to right |
| \* | Multiply | Left to right |
| – | Binary minus (subtraction) | Left to right |
| + | Binary plus (Addition) | Left to right |
| >> | Right shift | Left to right |
| << | Left shift | Left to right |
| > | Greater than | Left to right |
| < | Less than | Left to right |
| >= | Greater than or equal | Left to right |
| <= | Less than or equal | Left to right |
| == | Equal to | Left to right |
| != | Not equal to | Left to right |
| ^ | Bitwise exclusive OR | Left to right |
| & | Bitwise AND | Left to right |
| || | Logical OR | Left to right |
| | | Bitwise OR | Left to right |
| ?: | Conditional Operator | Right to left |
| && | Logical AND | Left to right |
| , | Separator of expressions | Left to right |
| = | Simple assignment | Right to left |
| /= | Assign quotient | Right to left |
| \*= | Assign product | Right to left |
| %= | Assign remainder | Right to left |
| -= | Assign difference | Right to left |
| += | Assign sum | Right to left |
| |= | Assign bitwise OR | Right to left |
| ^= | Assign bitwise XOR | Right to left |
| &= | Assign bitwise AND | Right to left |
| >>= | Assign right shift | Right to left |
| <<= | Assign left shift | Right to left |

**Some Important Points to Remember**

1. We only use associativity when we have two or more operators that have the same precedence in an expression.

The point to note here is that associativity is not applicable when we are defining the order of evaluation of operands with different levels of precedence.

2. All the operators that have a similar level of precedence have the same associativity. It is very important, or else the compiler won’t be able to decide what order of evaluation must an expression follow when it has two operators with the same precedence but different associativity. For example, – and + have the very same associativity.

3. The associativity and precedence of prefix ++ and postfix ++ are very different from each other. Here, the precedence of prefix ++ is less as compared to the postfix ++. Thus, their associativity also turns out to be different. Here, the associativity of prefix ++ is from right to left, while the associativity of postfix ++ is from left to right.

4. We must use a comma (,) very carefully, as it has the least level of precedence among all the other operators present in an expression.

5. We don’t have chaining of comparison in the C programming language. The Python language treats the expressions such as x > y > z as x > y and y > z. No similar type of chaining occurs in the C program.

**Practice Problems on Operator Precedence and Associativity in C**

**1.** What would be the output/result obtained from this program?

int main()

{

int x = 5, y = 10;

int z;

z = x \* 2 + y;

printf(“\n output = %d”, z);

return 0;

}

**A.** 0

**B.** 5

**C.** 20

**D.** 12

**Answer – C.** 20

We can calculate the output of this code using the precedence table. According to this table, the precedence of the assignment operator ( – ) and the addition operator ( + ) is lower than that of the multiplication operator ( \* ). Thus, multiplication will get evaluated first here.

Thus, z = x \* 2 + y

z = 5 \* 2 + 10

z = 10 + 10

z = 20

**2.** What would be the output/result obtained from this program?

int main()

{

int x = 50, y = 5;

int z;

z = x / 2 + y;

printf(“\n output = %d”, z);

return 0;

}

**A.** 80

**B.** 30

**C.** 0

**D.** 5

**Answer – B.** 30

We can calculate the output of this code using the precedence table. According to this table, the precedence of the assignment operator ( – ) and the addition operator ( + ) is lower than that of the division operator ( / ). Thus, multiplication will get evaluated first here.

Thus, z = x / 2 + y

z = 50 / 2 + 5

z = 25 + 5

z = 30

**3.** What would be the output/result obtained from this program?

int main()

{

int x = 30, y = 12;

int z;

z = x \* 2 / y;

printf(“\n output = %d”, z);

return 0;

}

**A.** 5

**B.** 10

**C.** 15

**D.** 60

**Answer – A.** 5

The evaluation of this expression would be (30 \* 2) / 12 because the associativity will be left to right for both the operators – multiplication and division.

Thus, z = x \* 2 / y

z = (30 \* 2) / 12

z = 60 / 12

z = 5

**4.** What would be the output/result obtained from this program?

int main()

{

int x = 80, y = 3;

int z;

z = x / 2 \* y;

printf(“\n output = %d”, z);

return 0;

}

**A.** 10

**B.** 30

**C.** 55

**D.** 120

**Answer – D.** 120

The evaluation of this expression would be (80 / 2) \* 3 because the associativity will be left to right for both the operators – multiplication and division.

Thus, z = x / 2 \* y

z = (80 / 2) \* 3

z = 40 \* 3

z = 120