

## Unit-4

# Operators and Expressions

### Introduction

An operator specifies an operation to be performed on operand. The variables, constant can be joined by various operators to form expressions. An operand is a data item on which an operator acts. Some operator required two operands, while others can act upon only one operand. C includes a large number of operators that fall under several different categories which are as:

1. Arithmetic operator
2. Assignment operator
3. Increment and decrement operator
4. Relational operator
5. Logical operators
6. Conditional operator
7. Comma operator
8. Sizeof operator
9. Bitwise operator

### 1. Arithmetic operator:

Arithmetic operators are used for numeric calculations. They are of two types:

- i. Unary arithmetic operator
- ii. Binary arithmetic operators.

#### i. Unary Arithmetic operator:

Unary operators require only one operand. For example:

+x                      -y

Here, '-' changes the sign of the operand y.

#### ii. Binary Arithmetic operator:

Binary operators require two operands. There are five binary arithmetic operands:

Operators	Purpose
+	Addition
-	Subtraction
*	Multiplication

/	Division
%	Give the remainder in integer division

% (modulus operator) cannot be applied with floating point operands. There is no exponent operator in C. However there is a library function pow ( ) to carry out an exponentiation operation. *Note that* Unary plus and unary minus operators are different from the addition and subtraction operators.

### Integer Arithmetic:

When both operands are integers, then the arithmetic operation with these operands is called integer arithmetic and the resulting value is always an integer. Let us take two variables a and b. The value of a = 17 and b = 4. The results of the following operations are:

Expressions	Result
a + b	21
a – b	13
a * b	68
a / b	4 (decimal part truncates)
a % b	1 (remainder after integer division)

After division operation the decimal part will be truncated and result is only integer part of quotient. After modulus operation the result will be remainder part of integer division. The second operand must be non zero for division and modulus operations.

```
/* Program to understand the integer arithmetic operation*/
```

```
#include<stdio.h>
```

```
void main ( )
```

```
{
```

```
    int a = 17, b = 4;
```

```
    printf ("Sum = %d\n", a +b );
```

```
    printf ("Difference = %d\n", a - b);
```

```
    printf ("Product = %d\n", a * b);
```

```
    printf ("Quotient = %d\n", a / b);
```

```
    printf ("Remainder = %d\n", a % b);
```



}

**Output:**

Sum = 21  
Difference = 13  
Product = 68  
Quotient = 4  
Remainder = 1

**Floating Point Arithmetic:**

When both operands are of float type then the arithmetic operation with these operands is called floating point arithmetic operator. Let us take two variables a and b. The value of a = 12.4 and b = 3.1, the results of the following operations are as:

Expression	Result
a + b	15.5
a – b	9.3
a * b	38.44
a / b	4.0

The modulus operator % cannot be used with floating point numbers.

```
/* Program to understand the floating point arithmetic operation*/  
#include<stdio.h>  
void main ( )  
{  
    int a = 12.4, b = 3.1;  
    printf ("Sum = %.2f\n", a +b );  
    printf ("Difference = %.2f\n", a - b);  
    printf ("Product = %.2f\n", a * b);  
    printf ("a/b = %.2f\n", a/b);  
}
```

**Output:**

Sum = 15.50  
Difference = 9.3  
Product = 38.44



$$a/b = 4.0$$

### Mixed Mode Arithmetic:

When one operand is of integer type and the other is of floating type then the arithmetic operation with these operands is known as mixed mode arithmetic and the resulting value is float type. If  $a = 12$  and  $b = 2.5$

Expression	Result
$a + b$	14.5
$a - b$	9.5
$a * b$	30.0
$a / b$	4.8

Sometimes mixed mode arithmetic can help in getting exact result. For example the result of expression  $5 / 2$  will be 2 since integer arithmetic is applied. If we want exact we can make one of the operand float type. For example,  $5.0/2$  or  $5/2.0$  both will give result 2.5 .

## 2. Assignment operator

A value can be stored in a variable with the use of assignment operators. The assignment operators “=” is used in assignment expression and assignment statements.

The operand on the left hand side should be a variable, while the operand on the right hand side can be any variable, value or expression. The value of the right hand operand is assigned to the left hand operand. Here are some examples of assignment expressions;

$x = 8$  /\* 8 is assigned to x\*/

$y = 5$  /\* 5 is assigned to y\*/

$s = x + y - 2$  /\* value of expression  $x + y - 2$  is assigned to s\*/

$y = x$  /\* value of x is assigned to y\*/

$x = y$  /\*value of y is assigned to x\*/

The value that is being assigned is considered as value of the assignment expression. For example,  $x = 8$  is an assignment expression whose value is 8.

We can have multiple assignment expressions also for example;

$x = y = z = 20$

Here all the three variable  $x$ ,  $y$ ,  $z$  will be assigned value 20, and the value of whole expressions will be 20.

If we put a semicolon after the assignment expression, then it becomes an assignment statement. For example these are assignment expressions;

$x = 8;$



```
y = 5;  
s = x + y - 2;  
x = y = z = 20;
```

When the variable on the left hand side of the assignment operator also occurs on right hand side then we can avoid writing the variable twice by using compound assignment operator. For example:

```
x = x + 5
```

can be also written as `x += 5`

Here `+=` is a compound assignment operator.

Similarly we have other compound assignment operators:

```
x -= 5 is equivalent to x = x - 5  
y *= 5 is equivalent to y = y * 5  
sum /= 5 is equivalent to sum = sum/5  
k %= 5 is equivalent to k = k % 5.
```

### 3. Increment and Decrement operator

C has two useful operator increment ( `++` ) and decrement ( `--` ). These are unary operators because they operate only the single operand. The increment operator ( `++` ) increments the value of the variable by one and decrement operator ( `--` ) decrements the value of variable by 1.

```
++x is equivalent to x = x + 1  
--x is equivalent to x = x - 1
```

These operators should be used only with variables; they can't be used with constant for expressions. For example the expression `++5` or `++(x+y+z)` are invalid.

**These operators are of two types:**

- i. Prefix increment / decrement - operator is written before operand. eg. `++x` or `--x`
- ii. Postfix increment/decrement - operator is written after operand eg. `x++` or `x--`

#### Prefix Increment/decrement:

Here first the value of variable is incremented/decremented then the new value is used in the operation. Let's us take a variable `x` whose value is 3. The statement `y = ++x;` means first increment of the value of `x` by 1, then assign the value of `x` to `y`. This single statement is equivalent to these two statements;

```
x = x + 1;  
y = x;
```

Hence, now the value of `x` is 4 and value of `y` is 4.



The statement `y = --x;` means first decrement the value of `x` by 1 then assign the value of `x` to `y`. This single statement is equivalent to these two statements;

```
x = x - 1;  
y = x
```

Hence, now the value of `x` is 3 and value of `y` is 3 .

```
/* Program to understand the use of prefix increment/decrement*/
```

```
#include <stdio.h>
```

```
void main ( )
```

```
{  
    int x = 8;  
    printf("x = %d\t", x);  
    printf("x = %d\t", ++x); /*prefix increment*/  
    printf("x = %d\t", x);  
    printf("x = %d\t", --x); /*prefix decrement*/  
    printf("x = %d", x);  
}
```

### Output:

x = 8            x = 9            x = 9            x = 8            x = 8

In the second `printf` statement, the first value of `x` is incremented and then printed; similarly in the fourth `printf` statement, first the value of `x` is decremented and then printed.

### Postfix Increment/Decrement:

Here first the value of variable is used in the operation and then increment/decrement is performed. Let us take a variable whose value is 3.

The statement `y = x ++;` means first the value of `x` is assigned to `y` and then `x` is incremented by 1. This statement is equivalent to these two statements;

```
y = x ;  
x = x + 1;
```

Hence, now value of `x` is 4 and value of `y` is 3.

The statement `y = x--;` means first the value of `x` is assigned to `y` and then `x` is decremented by 1. This statement is equivalent to these two statements;

```
y = x;
```



```
x = x -1;
```

Hence, now the value of x is 3 and value of y is 4.

```
/* Program to understand the use of postfix increment/decrement*/
```

```
#include <stdio.h>
```

```
void main ( )
```

```
{
```

```
    int x = 8;
```

```
    printf("x = %d\t", x);
```

```
    printf("x = %d\t", x++); /*postfix increment*/
```

```
    printf("x = %d\t", x);
```

```
    printf("x = %d\t", x--); /*postfix decrement*/
```

```
    printf("x = %d", x);
```

```
}
```

### Output:

x = 8            x = 8            x = 9                    x = 9            x = 8

In the second printf statement, first the value of x is printed and then incremented; similarly in the fourth printf statement, first the value of x is printed and then decremented.

## 4. Relational operator

Relational operators are used to compare values of two expressions depending on their relations. An expression that contains relational operator is called relational expression. If the relation is true, then the value of relational expression is 1 and if the relation is false, the value of expression is 0. The relational operators are:

Operator	Meaning
<	Less than
<=	Less than or equal to
==	Equal to
!=	Not equal to
>	Greater than
>=	Greater than or equal to

Let us take two variables, a = 9 and b = 5, and form simple relational expression with them.



Expression	Relation	Value of expression
a < b	False	0
a <= b	False	0
a ==b	False	0
a != b	True	1
a > b	True	1
a >= b	True	1
a == 0	False	0
b != 0	True	1
a > 8	True	1
2 >4	False	0

/\* Program to understand the use of relational operators\*/

```
#include <stdio.h>
```

```
void main ( )
```

```
{
```

```
    int a,b;
```

```
    printf("Enter values for a and b:");
```

```
    scanf("%d%d", &a, &b);
```

```
    if(a<b)
```

```
        printf("%d is less than %d\n", a,b);
```

```
    if (a<=b)
```

```
        printf("%d is less than equal to %d\n", a,b);
```

```
    if (a==b)
```

```
        printf ("%d is equal to %d\n", a,b);
```

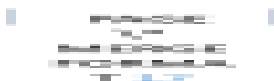
```
    if (a !=b)
```

```
        printf("%d is not equal to %d\n", a,b);
```

```
    if (a>b)
```

```
        printf("%d is greater than %d\n", a,b);
```

```
    if (a>=b)
```





```
printf(“%d greater than or equal to %d\n”, a,b);
}
```

### Output:

```
Enter values for a and b : 12 7
12 is not equal to 7
12 is greater than 7
12 is greater than or equal to 7
```

It is important to note that assignment operator (=) and equality operator (==) are entirely different. Assignment operator (=) used for assigning value while equality operator (==) is used to compare two expressions.

## 5. Logical operator

An expression that combines two or more expressions is termed as a logical expression. For combining these expressions we use logical operators. These operators return 0 for false and 1 for true. The operands may be constants, variables or expressions. C has three logical operators.

Operator	Meaning
&&	AND
	OR
!	NOT

Here logical NOT is a unary operator while the other two are binary operators. Before studying these operators let us understand the concept of true and false. In C any non zero value is regarded as true and zero is regarded as false.

### AND (&&) Operator

This operator gives the net result true if both the conditions are true, otherwise the result is false.

Condition 1	Condition 2	Result
False	False	False
False	True	False
True	False	False
True	True	True

Let us take three variables a = 10, b = 5 and c = 0



Suppose we have a logical expression:

`(a == 10) && (b < a)`

Here both the conditions `a == 10` and `b < a` are true, hence this whole expression is true. Since the logical operators return 1 for true hence the value of this expression is 1.

### OR (|) Operator

This operator gives the net result false, if both the conditions have the value false, otherwise the result is true.

Condition 1	Condition 2	Result
False	False	False
False	True	True
True	False	True
True	True	True

Let us take three variables `a = 10`, `b = 5` and `c = 0`

Consider the logical expression:

`(a >= b) || (b > 15)`

This gives result true because one condition is true.

### Not (!) Operator

This is unary operator and it neglects the value of the condition. If the value of the condition is false then it gives the result true. If the value of the condition is true then it gives the result false.

Condition	Result
False	True
True	False

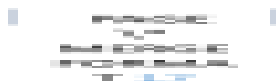
Let us take three variable `a = 10`, `b = 5` and `c = 0`

Suppose we have this logical expression:

`!(a == 10)`

The value of the condition `(a == 10)` is true. NOT operator negates the value of the condition. Hence the result is false.

## 6. Conditional Operator



Conditional operator is a ternary operator (? and -) which requires three expressions as operands. This is written as-

Test expression ? expression 1 : expression2

Firstly the test expression is evaluated,

- i. If testexpression is true (nonzero), then expression1 is evaluated and it becomes the value of the overall conditional expression.
- ii. If test expression is false(zero), then expression2 is evaluated and it becomes the value of overall conditional expression.

For example consider this conditional expression-

$a > b ? a : b$

Here first the expression  $a > b$  is evaluated, if the value is true then the value of variable  $a$  becomes the value of conditional expression otherwise the value of  $b$  becomes the value of conditional expression.

Suppose  $a = 5$  and  $b = 8$ , and we use the above conditional expression in a statement as-

$\text{max} = a > b ? a : b;$

First the expression  $a > b$  is evaluated, since it is false so the value  $b$  becomes the value of conditional expression and it is assigned to variable  $\text{max}$ .

In our next example we have written a conditional statement by putting a semicolon after the conditional expression.

$a < b ? \text{printf}(\text{"a is smaller"}) : \text{printf}(\text{"b is smaller"});$

Since the expression  $a < b$  is true, so the first  $\text{printf}$  function is executed.

`/* Program to print the larger of two numbers using conditional operator */`

`#include<stdio.h>`

`void main ( )`

`{`

`int a, b, max;`

`printf("Enter values for a and b:");`

`scanf("%d %d", &a, &b);`

`max = a > b ? a : b;                   /* ternary operator*/`

`printf("Larger of %d and %d is %d\n", a, b, max);`

`}`



**Output:**

Enter the values for a and b : 12 7

Larger of 12 and 7 is 12

**7. Comma Operator**

The comma operator ( , ) is used to permit different expressions to appear in situations where only one expression would be used. The expressions are separated by the comma operator. The separated expressions are evaluated from left to right and the type and value of the compound expression.

For example consider this expression-

`a = 8, b = 7, c = 9, a + b + c`

Here we have combined 4 expressions. Initially 8 is assigned to the variable a then 7 is assigned to the variable b, 9 is assigned to variable c and after this `a + b + c` is evaluated which becomes the value of whole expression. So the value of the above expression is 24. Now consider this statement.

`sum = (a = 8, b = 7, c = 9, a + b + c);`

Here the value of the whole expression on right side will be assigned to variable sum i.e. sum will be assigned value 24. Since precedence of comma operator is lower than that of assignment operator hence the parentheses are necessary here. The comma operator helps to make the code more compact, for example without the use of comma operator the above task would have been done in 4 statements.

`a = 8;`

`b = 7;`

`c = 9;`

`sum = a + b + c;`

`/* Program to understand the use of comma operator */`

`#include <stdio.h>`

`void main ( )`

`{`

`int a, b, c, sum;`

`sum = ( a = 8, b = 7, c = 9, a + b + c);`

`printf("Sum = %d\n", sum);`

`}`



**Output:**

Sum = 24

```
/* Program to interchange the value of two variables using comma operator */  
  
#include<stdio.h>  
  
void main ( )  
{  
    int a = 8, b = 7, temp;  
    printf("a = %d, b = %d\n", a, b);  
    temp = a, a = b, b = temp;  
    printf("a = %d, b = %d\n", a, b);  
}
```

**Output:**

a = 8 , b = 7

a = 7 , b = 8

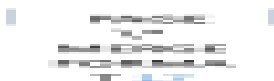
**8. Sizeof Operator**

*Size of* is an unary operator. This operator gives the size of its operand in terms of bytes. The operand can be a variable, constant or any data type (int, float, char etc). For example *sizeof(int)* gives the bytes occupied by the int datatype i.e. 2.

```
/*Program to understand the size of operator*/  
  
#include<stdio.h>  
  
void main ( )  
{  
    int var;  
    printf("size of int = \"%d\", sizeof(int));  
    printf("size of float = \"%d\", sizeof(float));  
    printf("size of var = \"%d\", sizeof(var));  
    printf("size of an integer constant=%d", sizeof(45));  
}
```

**Output:**

size of int = 2



size of float = 4

size of var = 2

size of an interger constant = 2

Generally sizeof operator is used to make portable programs i.e. programs that can be run on different machines. For example if we write our program assuming int to be of 2 bytes, then it won't run correctly on a machine on which int is of 4 bytes. So to make general code that can run on all machines we can use sizeof operator.

## 9. Bitwise Operator

C has ability to support that manipulation of data at the bit level. Bitwise operators are used for operations on individual bits. Bitwise operators operate on integers only. The bitwise operators are as-

Bitwise operator	Meaning
&	bitwise AND
	bitwise OR
~	one's complement
<<	left shift
>>	right shift
^	bitwise XOR

## Precedence and Associativity of Operators:

For evaluation of expressions having more than one operator, there are certain precedence and associativity rules defined in C. Let us see what these rules and why are they required.

Consider the following expression-

$$2 + 3 * 5$$

Here we have two operators - addition and multiplication operators. If addition is performed before multiplication then result will be 25 and if multiplication is performed before addition then the result will be 17.

In C language, operators are grouped together and each group is given a precedence level. The precedence of all the operators is given in the following table. The upper row in the table having higher precedence and it decreases as we move down the table. Hence the operators with precedence level 1 have highest precedence and with precedence level 15 have lowest precedence. So whenever an expression contains more than one operator, the



operator with a higher precedence is evaluated first. For example in the above expression, multiplication will be performed before addition since multiplication operator has higher precedence than the addition operator.

Operator	Description	Precedence Level	Associativity
( ) [ ] . →	Function call Array subscript Arrow operator Dot operator	1	Left to right
+ - ++ -- ! ~ * & (datatype) ) sizeof	Unary Plus Unary minus Increment Decrement Logical NOT One's complement Indirection Address Type case Size in bytes	2	Right to left
* / %	Multiplication Division Modulus	3	Left to right
+ -	Addition Subtraction	4	Left to right

<< >>	Left shift Right shift	5	Left to right
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	6	Left to right
== !=	Equal to Not equal to	7	Left to right
&	Bitwise AND	8	Left to right
^	Bitwise XOR	9	Left to right
	Bitwise OR	10	Left to right
&&	Logical AND	11	Left to right
	Logical OR	12	Left to right
? :	Conditional operator	13	Right to left
= *= /= %= += -= &= ^=  = <<= >>=	Assignment Operators	14	Right to left
,	Comma operator	15	Left to right

i.  $x = a + b < c$

Here + operator has higher precedence than < and =, and < has more precedence than =, so first  $a + b$  will be evaluated, then < operator will be evaluated, and at last the whole value will be assigned to x. If initial values are  $a = 2$ ,  $b = 6$  and  $c = 9$  then final value of x will be 1.

ii.  $x * = a + b$

Here + operator has higher precedence than \*=, so  $a + b$  will be evaluated before compound assignment. This is interpreted as  $x = x * (a + b)$  and not as  $x = x * a + b$

If initial values are  $x = 5$ ,  $a = 2$  and  $b = 6$ , the final value of x will be 13.

iii.  $x = a <= b \parallel b == c$

Here order evaluation of operators will be <=, ==, ||, =. If initial values are  $a = 2$ ,  $b = 3$  and  $c = 4$ , then the final value of x will be 1.

In the above examples we have considered expressions that contain operators having different precedence levels. Now consider a situation when two operators with the same precedence occur in an expression.

For example-





$$5 + 16 / 2 * 4$$

Here / and \* have higher precedence than + operator, so they will be evaluated before addition. But / and \* have same precedence, so which one of them will be evaluated first still remains a problem. If / is evaluated before \*, then the result is 37 otherwise the result is 7. Similarly consider this expression-

$$20 - 7 - 5 - 2 - 1$$

Here we have four subtraction operators, which of course have the same precedence level. If we decide to evaluate from left to right then answer will be 5 and if we evaluate from right to left then the answer will be 17.

To solve these types of problems, an associativity property is assigned to each operator. Associativity of the operators within same group is same. All the operators either associate from left to right or from right to left. The associativity of all operators is also given in the precedence table. Now again consider the above two expressions-

$$5 + 16 / 2 * 4$$

Since / and \* operators associate from left to right so / will be evaluated before \* and the correct result is 37.

$$20 - 7 - 5 - 2 - 1$$

The subtraction operator associated from left to right so the value of this expression is 5.

The assignment operator associated from right to left. Suppose we have a multiple assignment expression like this-

$$x = y = z = 5$$

Here initially the integer value 5 is assigned to variable z and then value of expression z = 5 is assigned to variable y. The value of expression z = 5 is 5, so 5 is assigned to variable y and now the value of expression y = z = 5 becomes 5. Now value of this expression is assigned to x and the value of whole expression x = y = z = 5 becomes 5.

## **Typecasting and conversion**

The value of an expression can be converted to a different data type if desired. The process of changing the type of an expression, precede the expression with name of type enclose in parenthesis in called typecasting.

It is of two types:

- a) Implicit
- b) Explicit

The conversion or typecast done automatically by the compiler itself is called implicit type conversion and one we do by typing the expression for conversion is known as explicit type

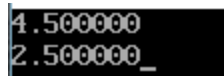


conversion. For example, if *i* is a type `int`, the expression `(float)i` casts *i* to type `float`. In other words, the program makes an internal copy of the value of *i* in floating-point format.

#### Example

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=5,b=2;
    float c=9,x;
    clrscr();
    x=c/b;           //implicit type conversion
    printf("%f\n",x);
    x=(float)a/b;    //explicit type conversion
    printf("%f",x);
    getch();
}
```

#### Output



```
4.500000
2.500000_
```