Operator & Expression

An operator is a symbol that tells the compiler to perform specific mathematical or logical functions. C language is rich in built-in operators and provides the following types of operators −

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators

Arithmetic Operators

An arithmetic operator performs mathematical operations such as addition, subtraction and multiplication on numerical values (constants and variables).

Operator	Meaning of Operator
+	addition or unary plus
-	subtraction or unary minus
*	multiplication
/	division
%	remainder after division (modulo division)

Example : Arithmetic Operators

```c
// C Program to demonstrate the working of arithmetic operators
#include <stdio.h>
int main()
{
    int a = 9,b = 4, c;

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

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

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

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

    c=a%b;
    printf("Remainder when a divided by b = %d \n",c);

    return 0;
}
```

Output

```
a+b = 13
a-b = 5
a*b = 36
a/b = 2
Remainder when a divided by b=1
```

The operators +, - and * computes addition, subtraction and multiplication respectively as you might have expected.

In normal calculation, `9/4 = 2.25`. However, the output is 2 in the program.

It is because both variables a and b are integers. Hence, the output is also an integer. The compiler neglects the term after decimal point and shows answer 2 instead of 2.25.

The modulo operator % computes the remainder. When `a = 9` is divided by `b = 4`, the remainder is 1. The % operator can only be used with integers.

Suppose a = 5.0, b = 2.0, c = 5 and d = 2. Then in C programming,
a/b = 2.5 // Because both operands are floating-point variables
a/d = 2.5 // Because one operand is floating-point variable
c/b = 2.5 // Because one operand is floating-point variable
c/d = 2   // Because both operands are integers

Program: Converting given days into months and days

```c
#include<stdio.h>
int main()
{
    int months, days;

    printf("Enter Days: ");
    scanf("%d",&days);

    months = days/30;
    days = days%30;

    printf("Months: %d\nDays: %d\n",months,days);
    return 0;
}
```

Increment and decrement operators

C programming has two operators increment ++ and decrement -- to change the value of an operand (constant or variable) by 1.

Increment ++ increases the value by 1 whereas decrement -- decreases the value by 1. These two operators are unary operators, meaning they only operate on a single operand.

Example: Increment and Decrement Operators

```c
#include <stdio.h>
int main()
{
    int a = 10, b = 100;
    float c = 10.5, d = 100.5;

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

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

    printf("++c = %f \n", ++c);

    printf("--d = %f \n", --d);
    return 0;
}
```

Output

```
++a = 11
--b = 99
++c = 11.500000
++d = 99.500000
```

Here, the operators ++ and -- are used as prefix. These two operators can also be used as postfix like `a++` and `a--`.

++ and -- operator as prefix and postfix

Suppose you use ++ operator as prefix like: `++var`. The value of `var` is incremented by 1 then, it returns the value.

Suppose you use ++ operator as postfix like: `var++`. The original value of `var` is returned first then, `var` is incremented by 1.

This is demonstrated examples in 4 different programming languages.

Example:

```c
#include <stdio.h>

int main()

{
  int var=5;

  // 5 is displayed then, var is increased to 6.
  printf("%d\n",var++);

  // Initially, var = 6. It is increased to 7 then, it is displayed.
  printf("%d",++var);

  return 0;
}
```

Relational Operators

A relational operator checks the relationship between two operands. If the relation is true, it returns 1; if the relation is false, it returns value 0.

Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 returns 0
>	Greater than	5 > 3 returns 1
<	Less than	5 < 3 returns 0
!=	Not equal to	5 != 3 returns 1
>=	Greater than or equal to	5 >= 3 returns 1
<=	Less than or equal to	5 <= 3 return 0

Example: Relational Operators

```
#include <stdio.h>
int main()
{
    int a = 5, b = 5, c = 10;

    printf("%d == %d = %d \n", a, b, a == b); // true
    printf("%d == %d = %d \n", a, c, a == c); // false

    printf("%d > %d = %d \n", a, b, a > b); //false
    printf("%d > %d = %d \n", a, c, a > c); //false


    printf("%d < %d = %d \n", a, b, a < b); //false
    printf("%d < %d = %d \n", a, c, a < c); //true


    printf("%d != %d = %d \n", a, b, a != b); //false
    printf("%d != %d = %d \n", a, c, a != c); //true


    printf("%d >= %d = %d \n", a, b, a >= b); //true
    printf("%d >= %d = %d \n", a, c, a >= c); //false


    printf("%d <= %d = %d \n", a, b, a <= b); //true
    printf("%d <= %d = %d \n", a, c, a <= c); //true

    return 0;

}
```

Output

```
5 == 5 = 1
5 == 10 = 0
5 > 5 = 0
5 > 10 = 0
5 < 5 = 0
5 < 10 = 1
5 != 5 = 0
5 != 10 = 1
5 >= 5 = 1
5 >= 10 = 0
5 <= 5 = 1
5 <= 10 = 1
```

Exercise 1:

```
int i,j,k;
i = 1;
j = 2;
k = 3;
```

What will be the value of following expression?

```
i < j
(i+j) >= k
(j+k) > (i+5)
k != 3
j == 2
```

Exercise 2:

```
int i = 7;
float f = 5.5;
char c = 'w';
```

What will be the value of following expression?

```
f > 5
(i+f) <= 10
c == 119
c != 'p'
c >= 10*(i+f)
```

Logical Operators

An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false. Logical operators are commonly used in decision making in C programming.

Operator	Meaning of Operator	Example
&&	Logial AND. True only if all operands are true	If c = 5 and d = 2 then, expression ((c == 5) && (d > 5)) equals to 0.
||	Logical OR. True only if either one operand is true	If c = 5 and d = 2 then, expression ((c == 5) || (d > 5)) equals to 1.
!	Logical NOT. True only if the operand is 0	If c = 5 then, expression ! (c == 5) equals to 0.

Example : Logical Operators

```
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;

   result = (a == b) && (c > b);
   printf("(a == b) && (c > b) equals to %d \n", result);

   result = (a == b) && (c < b);
   printf("(a == b) && (c < b) equals to %d \n", result);

   result = (a == b) || (c < b);
   printf("(a == b) || (c < b) equals to %d \n", result);

   result = (a != b) || (c < b);
   printf("(a != b) || (c < b) equals to %d \n", result);

   result = !(a != b);
   printf("!(a == b) equals to %d \n", result);

   result = !(a == b);
   printf("!(a == b) equals to %d \n", result);

   return 0;
}
```

Output

```
(a == b) && (c > b) equals to 1
(a == b) && (c < b) equals to 0
(a == b) || (c < b) equals to 1
(a != b) || (c < b) equals to 0
!(a != b) equals to 1
!(a == b) equals to 0
```

Explanation of logical operator program

- (a == b) && (c > 5) evaluates to 1 because both operands (a == b) and (c > b) is 1 (true).
- (a == b) && (c < b) evaluates to 0 because operand (c < b) is 0 (false).
- (a == b) || (c < b) evaluates to 1 because (a = b) is 1 (true).
- (a != b) || (c < b) evaluates to 0 because both operand (a != b) and (c < b) are 0 (false).
- !(a != b) evaluates to 1 because operand (a != b) is 0 (false). Hence, !(a != b) is 1 (true).
- !(a == b) evaluates to 0 because (a == b) is 1 (true). Hence, !(a == b) is 0 (false).

Exercise 1:

```
int i = 7;
float f = 5.5;
char c = 'w';
```

What will be the value of following expression?

```
(i >= 6) && (c == 'w')
(i >= 6) || (c == 119)
(f < 11) && (i>100)
(c != 'p') || ((i+f) <= 10)
```

Exercise 2:

```
int i = 7;
float f = 5.5;
```

What will be the value of following expression?

```
f  >  5
!(f > 5)
i <= 3
! (i <= 3)
i > (f+1)
!(i > (f+1))
```

Assignment Operators

An assignment operator is used for assigning a value to a variable. The most common assignment operator is =

Operator	Example	Same as
=	a = b	a = b
+=	a += b	a = a+b
-=	a -= b	a = a-b
*=	a *= b	a = a*b
/=	a /= b	a = a/b
%=	a %= b	a = a%b

Example: Assignment Operators

```
#include <stdio.h>
int main()
{
    int a = 5, c;

    c = a;
    printf("c = %d \n", c);

    c += a; // c = c+a
    printf("c = %d \n", c);

    c -= a; // c = c-a
    printf("c = %d \n", c);

    c *= a; // c = c*a
    printf("c = %d \n", c);

    c /= a; // c = c/a
    printf("c = %d \n", c);

    c %= a; // c = c%a
    printf("c = %d \n", c);

    return 0;
}
```

Output

```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

Some More Example:

In the following assignment expressions, suppose that `i` is an integer-type variable.

Expression	*Value*
`i = 3.3`	3
`i = 3.9`	3
`i = -3.9`	-3

Now suppose that `i` and `j` are both integer-type variables, and that `j` has been assigned a value of 5. Several assignment expressions that make use of these two variables are shown below.

Expression	*Value*	
`i = j`	5	
`i = j / 2`	2	
`i = 2 * j / 2`	5	(left-to-right associativity)
`i = 2 * (j / 2)`	4	(truncated division, followed by multiplication)

Finally, assume that i is an integer-type variable, and that the ASCII character set applies.

Expression	*Value*
`i = 'x'`	120
`i = '0'`	48
`i = ('x' - '0') / 3`	24
`i = ('y' - '0') / 3`	24

Suppose that i and j are integer variables whose values are 5 and 7, and f and g are floating-point variables whose values are 5.5 and –3.25. Several assignment expressions that make use of these variables are shown below. Each expression utilizes the *original* values of i, j, f and g.

Expression	*Equivalent Expression*	*Final value*
`i += 5`	`i = i + 5`	10
`f -= g`	`f = f - g`	8.75
`j *= (i - 3)`	`j = j * (i - 3)`	14
`f /= 3`	`f = f / 3`	1.833333
`i %= (j - 2)`	`i = i % (j - 2)`	0

Bitwise Operators

During computation, mathematical operations like: addition, subtraction, addition and division are converted to bit-level which makes processing faster and saves power.

Bitwise operators are used in C programming to perform bit-level operations.

Operators	Meaning of operators
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR
~	Bitwise complement
<<	Shift left
>>	Shift right

Bitwise AND operator &

The output of bitwise AND is 1 if the corresponding bits of two operands is 1. If either bit of an operand is 0, the result of corresponding bit is evaluated to 0.

Let us suppose the bitwise AND operation of two integers 12 and 25.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bit Operation of 12 and 25
  00001100
& 00011001
  _________
  00001000  = 8 (In decimal)
```

Example : Bitwise AND

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a&b);
    return 0;
}
```

Output

```
Output = 8
```

Bitwise OR operator |

The output of bitwise OR is 1 if at least one corresponding bit of two operands is 1. In C Programming, bitwise OR operator is denoted by |.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise OR Operation of 12 and 25
  00001100
| 00011001
  _________
  00011101  = 29 (In decimal)
```

Example: Bitwise OR

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a|b);
    return 0;
}
```

Output

```
Output = 29
```

Bitwise XOR (exclusive OR) operator ^

The result of bitwise XOR operator is 1 if the corresponding bits of two operands are opposite. It is denoted by ^.

```
12 = 00001100 (In Binary)
25 = 00011001 (In Binary)

Bitwise XOR Operation of 12 and 25
  00001100
| 00011001
  ________
  00010101  = 21 (In decimal)
```

Example: Bitwise XOR

```
#include <stdio.h>
int main()
{
    int a = 12, b = 25;
    printf("Output = %d", a^b);
    return 0;
}
```

Output

```
Output = 21
```

Bitwise complement operator ~

Bitwise compliment operator is an unary operator (works on only one operand). It changes 1 to 0 and 0 to 1. It is denoted by ~.

```
35 = 00100011 (In Binary)

Bitwise complement Operation of 35
~ 00100011
  ________
  11011100  = 220 (In decimal)
```

Twist in bitwise complement operator in C Programming

The bitwise complement of 35 (~35) is -36 instead of 220, but why?

For any integer `n`, bitwise complement of `n` will be `-(n+1)`. To understand this, you should have the knowledge of 2's complement.

2's Complement

Two's complement is an operation on binary numbers. The 2's complement of a number is equal to the complement of that number plus 1. For example:

```
 Decimal        Binary          2's complement
   0          00000000          -(11111111+1) = -00000000 = -0(decimal)
   1          00000001          -(11111110+1) = -11111111 = -256(decimal)
   12          00001100          -(11110011+1) = -11110100 = -244(decimal)
   220         11011100          -(00100011+1) = -00100100 = -36(decimal)
```

Note: Overflow is ignored while computing 2's complement.

The bitwise complement of 35 is 220 (in decimal). The 2's complement of 220 is -36. Hence, the output is -36 instead of 220.

Bitwise complement of any number N is -(N+1). Here's how:

```
bitwise complement of N = ~N (represented in 2's complement form)
2'complement of ~N= -(~(~N)+1) = -(N+1)
```

Example : Bitwise complement

```c
#include <stdio.h>
int main()
{
    printf("complement = %d\n",~35);
    printf("complement = %d\n",~-12);
    return 0;
}
```

Output

```
complement = -36
Output = 11
```

Shift Operators in C programming

There are two shift operators in C programming:

- Right shift operator
- Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

```
212 = 11010100 (In binary)
212>>2 = 00110101 (In binary) [Right shift by two bits]
212>>7 = 00000001 (In binary)
212>>8 = 00000000
212>>0 = 11010100 (No Shift)
```

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

```
212 = 11010100 (In binary)
212<<1 = 110101000 (In binary) [Left shift by one bit]
212<<0 =11010100 (Shift by 0)
212<<4 = 110101000000 (In binary) =3392(In decimal)
```

Example : Shift Operators

```
#include <stdio.h>
int main()
{
    int num=212, i;
    for (i=0; i<=2; ++i)
        printf("Right shift by %d: %d\n", i, num>>i);

     printf("\n");

     for (i=0; i<=2; ++i)
        printf("Left shift by %d: %d\n", i, num<<i);

     return 0;
}
```

```
Right Shift by 0: 212
Right Shift by 1: 106
Right Shift by 2: 53

Left Shift by 0: 212
Left Shift by 1: 424
Left Shift by 2: 848
```

Misc Operators ↦ sizeof & ternary

Besides the operators discussed above, there are a few other important operators including **sizeof** and **? :** supported by the C Language.

Show Examples

Operator	Description	Example
sizeof()	Returns the size of a variable.	sizeof(a), where a is integer, will return 4.
&	Returns the address of a variable.	&a; returns the actual address of the variable.
*	Pointer to a variable.	*a;
? :	Conditional Expression.	If Condition is true ? then value X : otherwise value Y

Example

Try following example to understand all the miscellaneous operators available in C −

```
#include <stdio.h>

main() {

   int a = 4;
   short b;
   double c;
   int* ptr;

   /* example of sizeof operator */
   printf("Line 1 - Size of variable a = %d\n", sizeof(a) );
   printf("Line 2 - Size of variable b = %d\n", sizeof(b) );
   printf("Line 3 - Size of variable c= %d\n", sizeof(c) );

   /* example of & and * operators */
   ptr = &a;             /* 'ptr' now contains the address of 'a'*/
   printf("value of a is  %d\n", a);
   printf("*ptr is %d.\n", *ptr);
```

```
   /* example of ternary operator */
   a = 10;
   b = (a == 1) ? 20: 30;
   printf( "Value of b is %d\n", b );

   b = (a == 10) ? 20: 30;
   printf( "Value of b is %d\n", b );
}
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - Size of variable a = 4
Line 2 - Size of variable b = 2
Line 3 - Size of variable c= 8
value of a is  4
*ptr is 4.
Value of b is 30
Value of b is 20
```

Operators Precedence in C

Operator precedence determines the grouping of terms in an expression and decides how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has a higher precedence than the addition operator.

For example, x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has a higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

Exercise:

What will be the value of x, y, z?

```
float a,b,c,x,y,z;
a = 9;
b = 12;
c = 3;

x = a - b / 3 + c * 2 -1;
y = a - b / (3 + c) * (2 -1);
z = a - (b / (3 + c) * 2) -1;
```

Type Conversion in C

A type cast is basically a conversion from one type to another. There are two types of type conversion:

1. **Implicit Type Conversion** Also known as 'automatic type conversion'.
   - Done by the compiler on its own, without any external trigger from the user.
   - Generally, takes place when in an expression more than one data type is present. In such condition type conversion (type promotion) takes place to avoid loss of data.
   - All the data types of the variables are upgraded to the data type of the variable with largest data type.

     bool --> char --> short int --> int --> unsigned int --> long --> unsigned --> long long --> float --> double --> long double

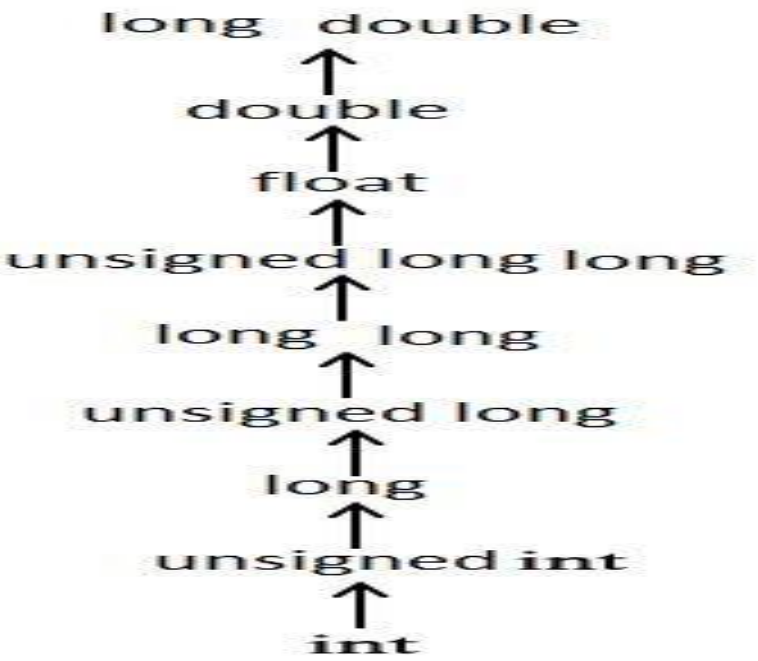
   - It is possible for implicit conversions to lose information, signs can be lost (when signed is implicitly converted to unsigned), and overflow can occur (when long long is implicitly converted to float).

Example of Type Implicit Conversion:

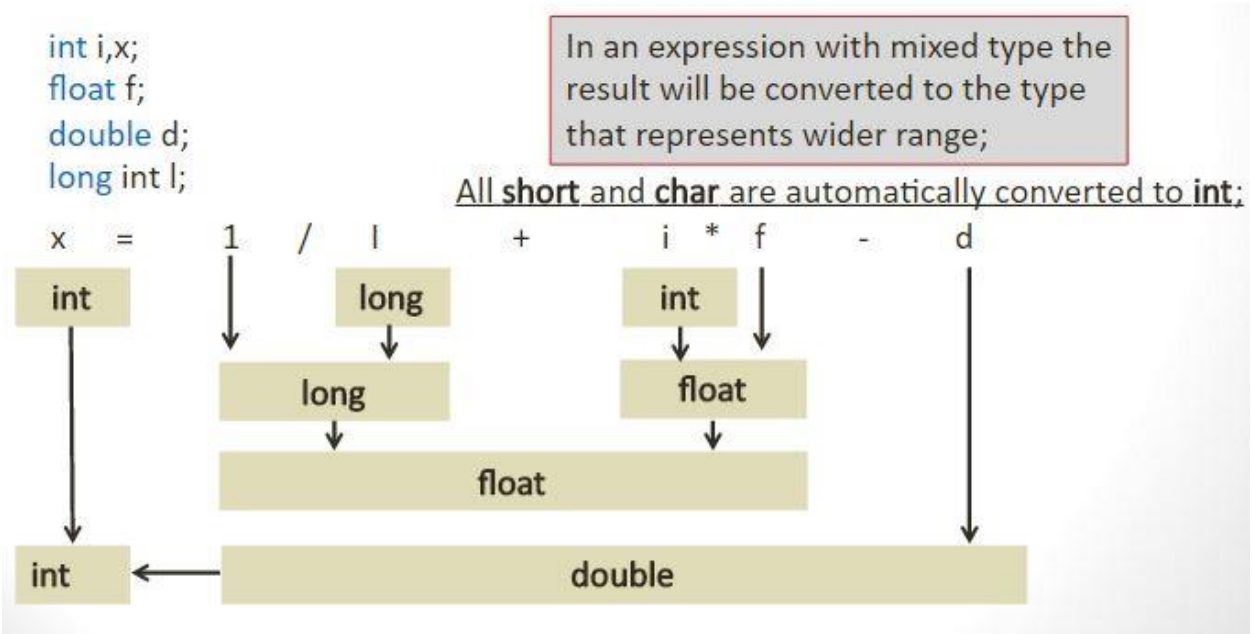
```
// An example of implicit conversion
#include<stdio.h>
int main()
{
    int x = 10;    // integer x
    char y = 'a';  // character c

    // y implicitly converted to int. ASCII
    // value of 'a' is 97

    x = x + y;
    // x is implicitly converted to float
    float z = x + 1.0;

    printf("x = %d, z = %f", x, z);
    return 0;
}
```

Output:

```
x = 107, z = 108.000000
```



Another Example:



2. **Explicit Type Conversion**– This process is also called type casting and it is user defined. Here the user can type cast the result to make it of a particular data type.
   The syntax in C:

```
(type) expression
```

Type indicated the data type to which the final result is converted.

```
// C program to demonstrate explicit type casting
#include<stdio.h>
int main()
{
    double x = 1.2;

    // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0;
}
```

Output:

```
sum = 2
```

Advantages of Type Conversion

- This is done to take advantage of certain features of type hierarchies or type representations.
- It helps us to compute expressions containing variables of different data types.