



# Introduction to Operators

1. Write a program to **find the area** of rectangular garden provided that length of garden is 12 meters and breadth of garden is 10 meters.

2. W.A.P to check annual turnover of your company is **greater than equal** to the annual turn over of the your competitor.

3. W.A.P to make a decision that **if** you have both chair and table then you start studying the Operators topic in C, otherwise simply go to the sleep.

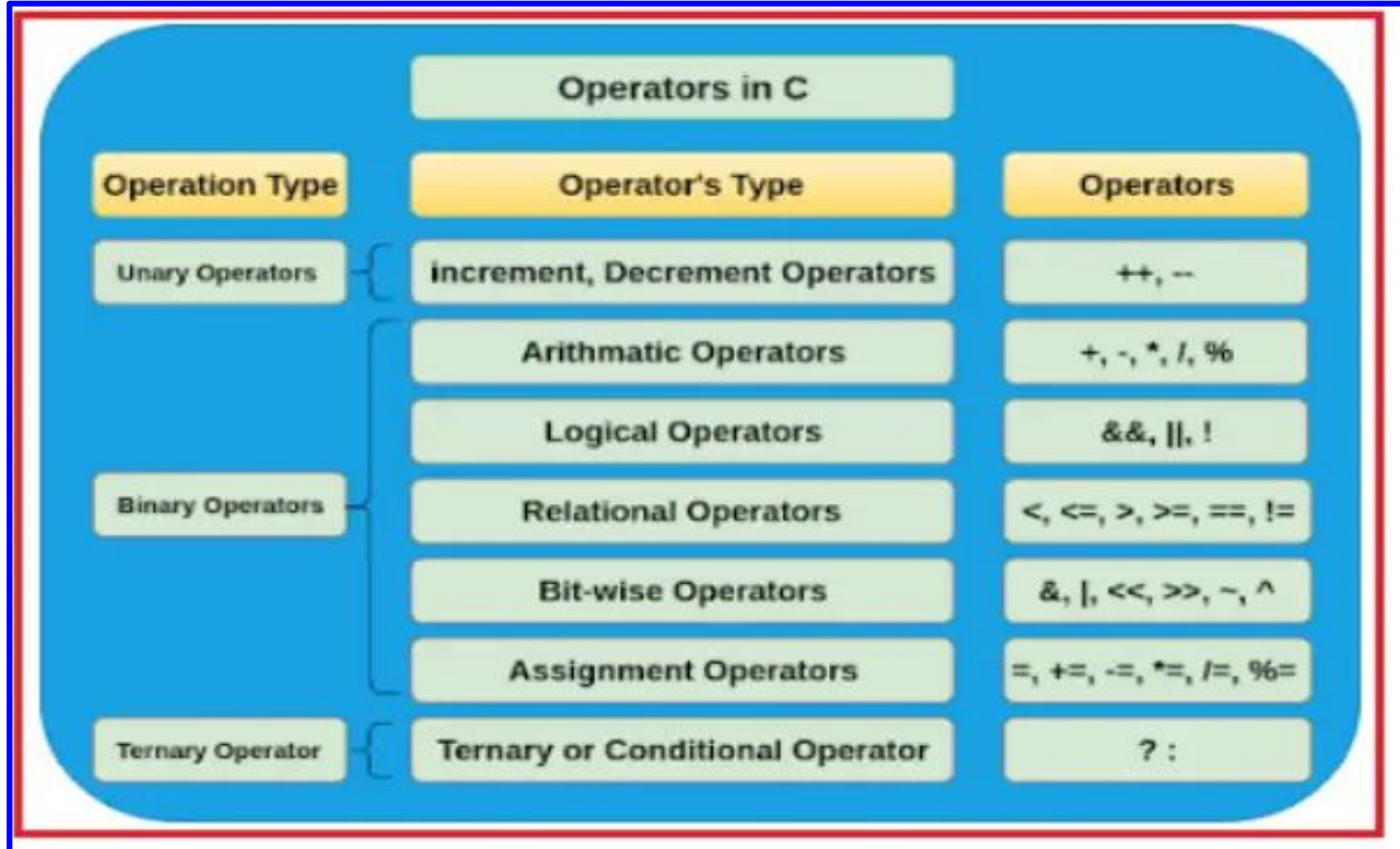
# What is the necessity of discussion?

1.  $12 * 10$  -> Arithmetic Operator

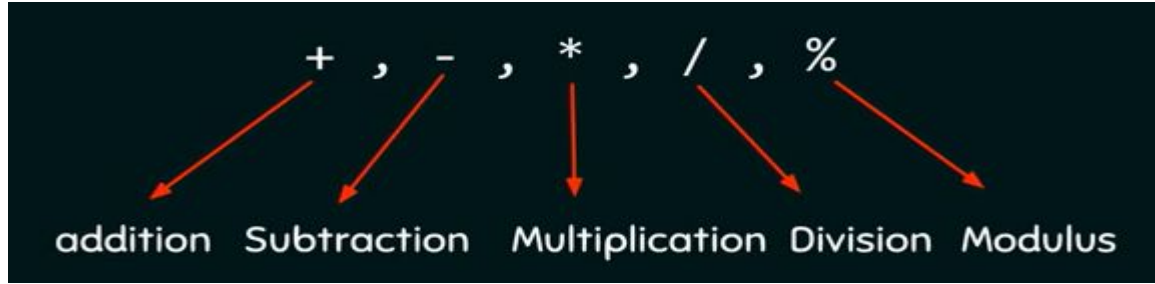
2.  $30000 \geq 50000$  -> Relational Operator

3. `if ( chair && table )` -> Logical Operator

# Types of Operators:



# Arithmetic Operators:



All are ***binary operators*** -> it means two operands are required to perform operation.

## Operator Precedence and Associativity:

| Precedence<br>↓ | Operators        | Associativity |
|-----------------|------------------|---------------|
| Highest         | $*$ , $/$ , $\%$ | Left to right |
| Lowest          | $+$ , $-$        | Left to right |

$a + b * c / d.$

**Associativity:** Associativity is used only when two or more operators are of same precedence.

Ex:

$$a + b * d - c \% a$$

↓

$$a + (b * d) - (c \% a)$$

## **Increment and Decrement Operators:**

Increment Operator is used to increment the value of variable by one. Similarly Decrement Operator is used to decrement the value of variable 1.

**Ex:**

```
int a = 12;
```

```
a++;
```

```
a=13.
```

Note:  $a++ \rightarrow a = a+1$ .

Pre-increment operator

`++a;`

Post-increment operator

`a++;`

Pre-decrement operator

`--a;`

Post-decrement operator

`a--;`

Note: You cannot use the **r value** before or after increment/decrement.

**Ex:** `(a+b)++`      -> Error

`++(a+b)`      -> Error

**Error:** l value required and an increment operand.



# What is significance of l value and r value ?

**l value** : an object that has identifiable location in the memory (i.e.address)

In any assignment statement “lvalue” must have the capability of hold the data.

lvalue must be variable because they have the capability of store the data.

lvalue cannot be function or expression .

**r value**: an object that has no identifiable location in memory.

```
(a + b)++;    error!
```

```
++(a + b);    error!
```

```
error: lvalue required as increment operand
```



Compiler is expecting a variable as an increment operand but we are providing an expression (a + b) which does not have the capability to store data.

## Pre Increment & Post increment Operator or pre-decrement & post decrement operator?

**Pre** -> means first increment/decrement then assign it to the another variable.

**Post** -> means first assign to another variable then increment/decrement.

`x = ++a;`

`x = a++;`

x

a

6

~~5~~

6

x

a

5

~~5~~

6

What is the output of the following program?

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

```
int main() {  
    int a = 4, b = 3;  
    printf("%d", a+++b);  
    return 0;  
}
```

a+++b

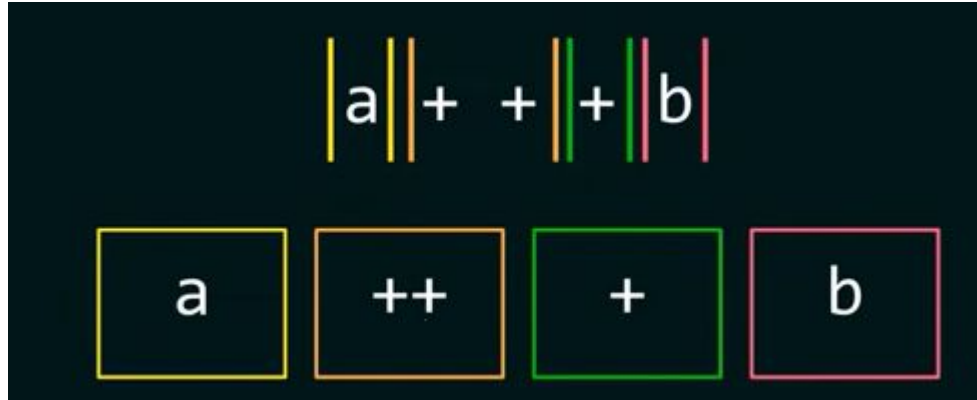
## Token Generation:

Lexical analysis is the first phase in the compilation process.

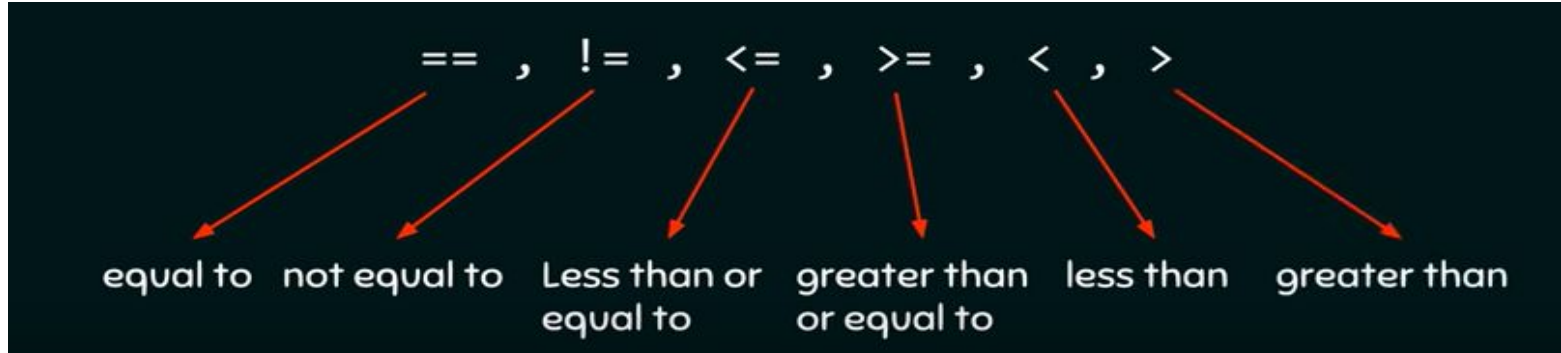
Lexical analyzer(scanner) scans the whole source program and its find meaningful sequence of characters(lexemes) then it convert it into a token.

Ex: int ->(keyword,int value)

int a = 5 ;



# Relational Operators:



Note: used to comparing two values ( return either true or false)

## Logical Operators:

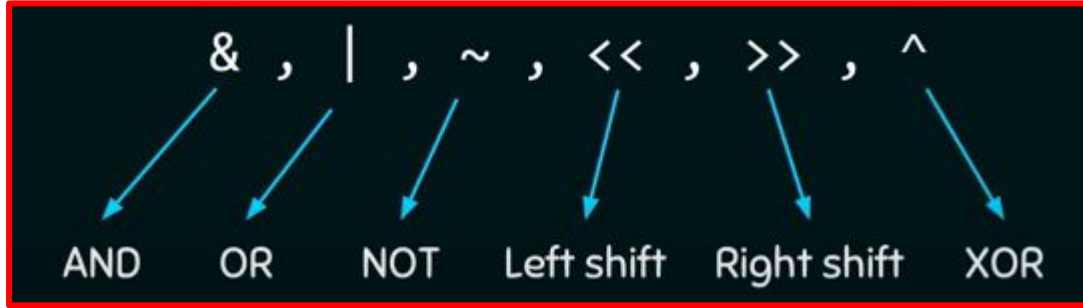
&& and || used to combine the two conditions.

&& -> returns true when all conditions under considerations are true and returns false when any one or more than one condition is false.

Ex:

```
int a=5;
if( a ==5 && a != 6 && a <=55)
{
    printf( " welcome to CDAC Hyderabad");
}
```

# Bitwise Operators:



## Bitwise AND:

7     $\rightarrow$    0 1 1 1

4     $\rightarrow$    & 0 1 0 0

---

4     $\leftarrow$    0 1 0 0

---

7 & 4 = 4

| A | B | A&B |
|---|---|-----|
| 0 | 0 | 0   |
| 0 | 1 | 0   |
| 1 | 0 | 0   |
| 1 | 1 | 1   |

## Difference between Bitwise and Logical Operators:

```
#include <stdio.h>

int main() {
    char x = 1, y = 2; //x = 1(0000 0001), y = 2(0000 0010)
    if(x&y) //1&2 = 0(0000 0000)
        printf("Result of x&y is 1");
    if(x&&y) //1&&2 = TRUE && TRUE = TRUE = 1
        printf("Result of x&&y is 1");

    return 0;
}
```



## Bitwise left shift operator :

First operand << Second Operand.

First operand -> whose bits get left shifted.

Second Operand -> Decides the number of places to shift the bits.

Note: when bits are shifted left then trailing positions are filled with zeros.

Ex:

```
int main() {  
    Char var = 3;  
    printf("%d", var <<1);  
    return 0;}
```



## How left shift works?

`var << 1`



Left shift by 1 position



`var = 3`

`3 = 0000 0011`



`0000 011_`



`0000 0110`

**Ans: 6**

Note: left shifting is equivalent to multiplication of by

$2^{\text{rightOperand}}$

Ex:

```
var = 3;
```

```
var << 1      Output: 6  [3 x 21]
```

# Conditional Operator: ? :

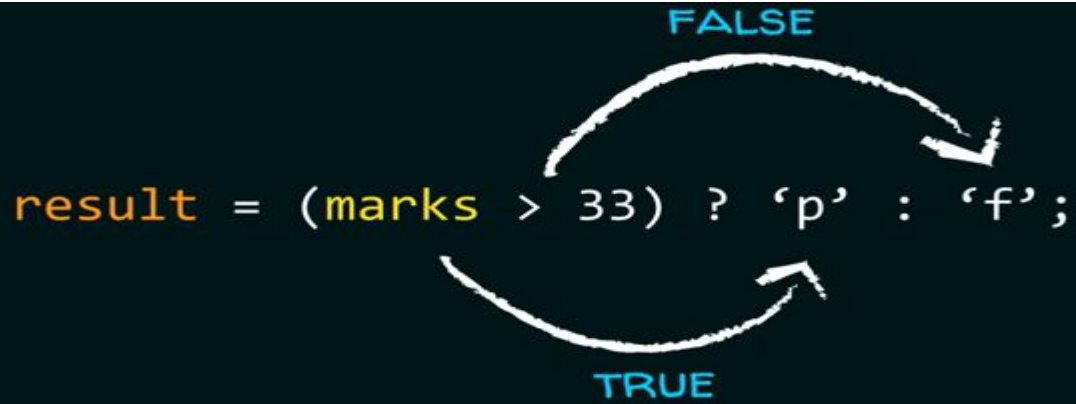
Ex:

```
char result;  
int marks;  
  
if(marks > 35)  
{  
    result = 'p';  
}  
  
else  
{  
    result = 'f';  
}
```

`(marks > 33)` is a boolean expression, therefore it will return either TRUE or FALSE

```
result = (marks > 35) ? 'p' : 'f'
```

`(marks > 33)` is a boolean expression, therefore it will return either TRUE or FALSE



The diagram shows the code `result = (marks > 33) ? 'p' : 'f';` on a dark background. Two white curved arrows originate from the condition `(marks > 33)`. One arrow points upwards and to the right to the character `'p'`, with the word `FALSE` written in light blue above it. The other arrow points downwards and to the right to the character `'f'`, with the word `TRUE` written in light blue below it.

```
result = (marks > 33) ? 'p' : 'f';
```

# Comma Operator:

It can be used as an operator.

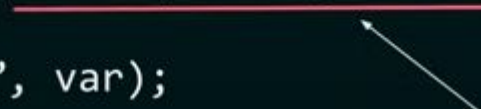
```
int a = (4,5,8);  
printf("%d",a);
```

**Note:** comma operator returns the rightmost operand of the expression.

**Note:** comma operator returns the rightmost operand of the expression and it simply evaluates the rest of operands and finally rejects them.

Example:

```
int var = (printf("%s\n", "HELLO!"), 5);  
printf("%d", var);
```



Comma operator is having least precedence among all operators in C.

```
int a;  
a = 3, 4, 8;  
  
printf("%d", a);
```

≡

```
int a;  
(a = 3), 4, 8;  
  
printf("%d", a);
```

O/P: 3

Note: Comma operator having least precedence among all operators in C.

```
int a;  
a=10,12,16;  
printf(“%d”,a);
```

```
int a;  
(a = 10),12,16;  
printf(“%d”,a);
```

**O/P : 10**



Ex:

```
int a = 3, 4, 8;  
printf("%d", a);
```

O/p: Error. (bcz in the variable declaration only we are initializing the values)  
*Here comma operator behaving like a separator with in function calls, definitions, variable declarations*

```
int a = 3, 4, 8;  
is equivalent to  
int a = 3; int 4; int 8;
```

## Type Casting:

- **Implicit conversation:** it is done by compiler and there is no loss of information.  
Ex: x is 8 bit no, y is 16 bit.,now perform (x+y)
- **Explicit Conversation:** it is done by programmer and there will be loss of information.

Ex: float f = 4.635

int y = (int)x;

### Imp Cases

- Case 1: int a=10, b=3;  
float c = a/b;
- Case 2: int a=10, b=3;  
float c = (float) a/b;
- Case 3: int a=10, b=3;  
float c=(float)(a/b);