Introduction to C programming
MODULE-2

Chapter-1 Operators in C, Type conversion and Type Casting

2.1 Operators and Expressions

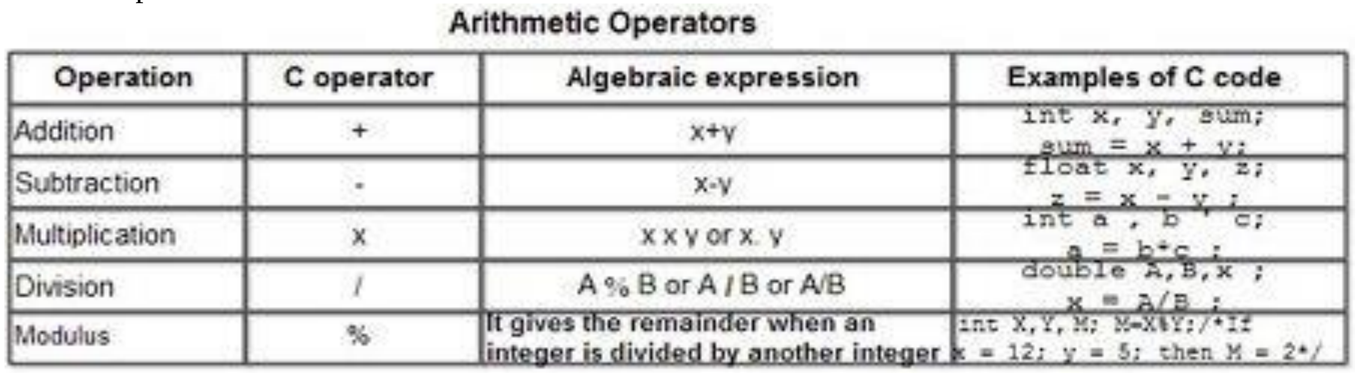
Expressions: An expression is a sequence of Operands and Operators that reduces to single value. An expression can be simple or complex.

- An operator is a syntactical token that requires an action to be taken
- An operand is an object on which an operation is performed. It receives an operation action.
- Simple expression contains only one operator. Ex 1+6
- A complex expression contains more than one operator. Ex 3*5+7

In C language we can formulate different types of expressions based on operators used. Various important operators available in C language are:

1. Arithmetic Operators
2. Relational Operators
3. Logical Operators
4. Assignment Operators
5. Increment / Decrement operators
6. Bitwise Operators
7. Conditional operator
8. Special operator

1. Arithmetic Operator: C provides all basic arithmetic operators. Here is a brief summary of different operators.



Arithmetic Operators

Operation	C operator	Algebraic expression	Examples of C code
Addition	+	x+y	int x, y, sum; sum = x + y;
Subtraction	-	x-y	float x, y, z; z = x - y ;
Multiplication	x	x x y or x. y	int a , b ' c; a = b*c ;
Division	/	A % B or A / B or A/B	double A,B,x ; x = A/B ;
Modulus	%	It gives the remainder when an integer is divided by another integer	int X,Y,M; M=X%Y;/*If x = 12; y = 5; then M = 2*/

```
// 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
```

Arithmetic expressions are of three types:

1. Integer Arithmetic
2. Real Arithmetic
3. Mixed mode Arithmetic

Integer Arithmetic: It is an expression evaluation where all the operands are of integer data type. Let us take an example:
int a=10,b=5,c; c=a/b; printf("%d", c);
c will contain quotient value 2

Real Arithmetic: It is an expression evaluation where all the operands are of float data type. Here is an example:

float x=2.5, y=10.0,z; z=y/x;

printf("%f", z);

z will contain value 4.000000

Mixed Mode Arithmetic: In an expression if some operands are of integer type and others are of float type it is called mixed mode arithmetic expression. Here the data type of small size gets automatically converted to data type of bigger size.

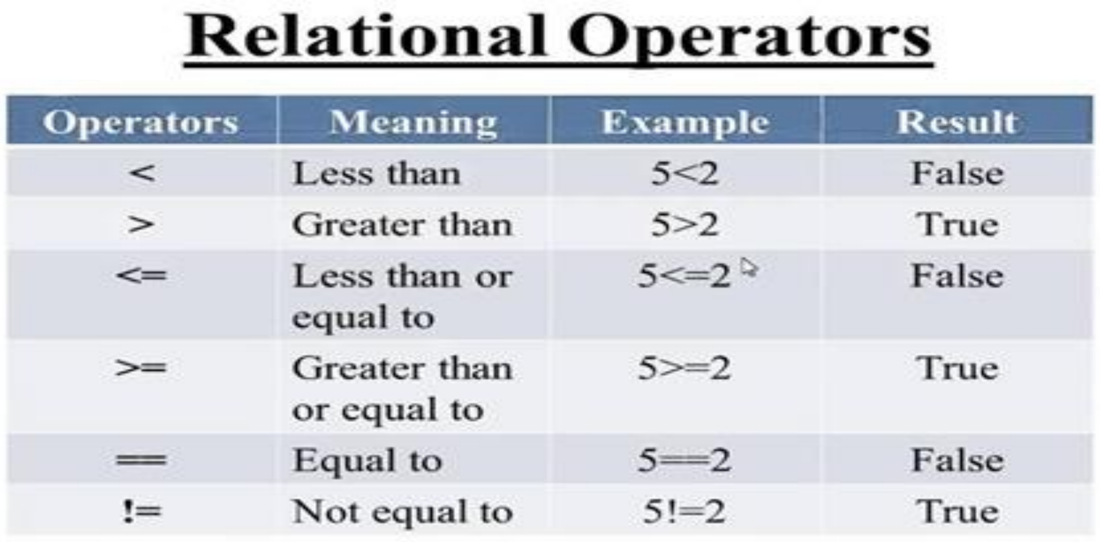
Here is an example for such expression:

int a=25; float x=4,z; z=a/x;

printf("%f",z);

Here ‘a’ is of integer data type and ‘x’ is of float type, therefore ‘a’ automatically gets converted to float type and output is: 6.25

2. Relational operator: Relational operators are often used to compare two quantities on their relation, take certain decision. An expression that uses relational operators are called relational expressions. The value of relational expression is either 1 or 0. 1 means true, 0 means false. C supports 6 relational operators



Relational Operators

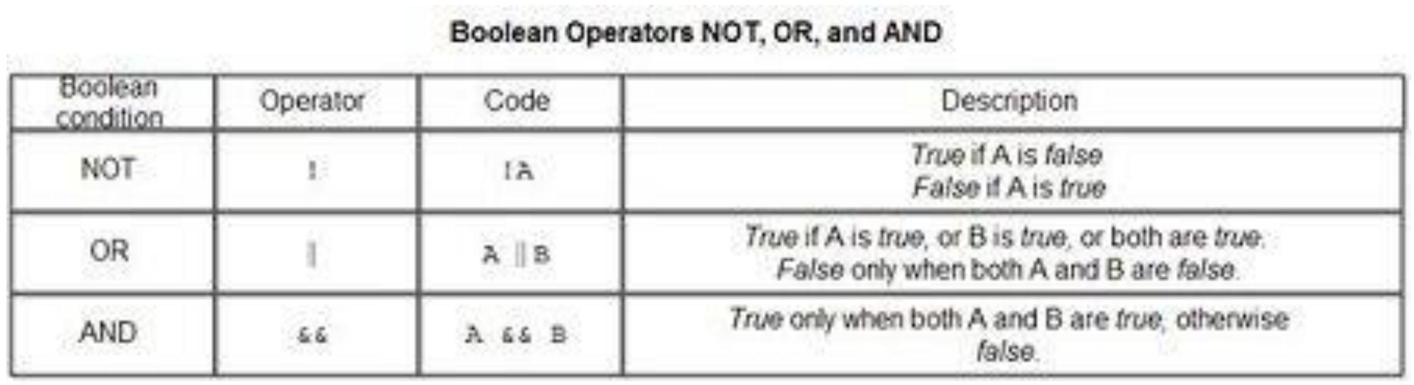
Operators	Meaning	Example	Result
<	Less than	5<2	False
>	Greater than	5>2	True
<=	Less than or equal to	5<=2	False
>=	Greater than or equal to	5>=2	True
==	Equal to	5==2	False
!=	Not equal to	5!=2	True

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

   printf("%d == %d is %d \n", a, b, a == b);
   printf("%d == %d is %d \n", a, c, a == c);
   printf("%d > %d is %d \n", a, b, a > b);
   printf("%d > %d is %d \n", a, c, a > c);
   printf("%d < %d is %d \n", a, b, a < b);
   printf("%d < %d is %d \n", a, c, a < c);
   printf("%d != %d is %d \n", a, b, a != b);
   printf("%d != %d is %d \n", a, c, a != c);
   printf("%d >= %d is %d \n", a, b, a >= b);
   printf("%d >= %d is %d \n", a, c, a >= c);
   printf("%d <= %d is %d \n", a, b, a <= b);
   printf("%d <= %d is %d \n", a, c, a <= c);

   return 0;
}
```

Output

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

3. Logical Operator: Logical operators are used to test more than one condition mad make decision. C provides 3 logical operators as shown below.



Boolean Operators NOT, OR, and AND

Boolean condition	Operator	Code	Description
NOT	!	!A	*True* if A is *false* *False* if A is *true*
OR	||	A || B	*True* if A is *true*, or B is *true*, or both are *true*. *False* only when both A and B are *false*.
AND	&&	A && B	*True* only when both A and B are *true*, otherwise *false*.

Note:
1. In Logical AND operator output is TRUE only if both inputs are TRUE.
2. In Logical OR operator output is TRUE if any of the inputs are TRUE.
3. In Logical NOT operator output is TRUE if input is FALSE and vice versa. Here are some examples on logical expressions:

Example1: int A=5, B=10, X=20, Y=5;
What is truth value of expression: (A) && (B)?
(5) && (10) both are non-zero number: 1 && 1 = 1 (True) (Note: Any non-zero number is treated as 1)

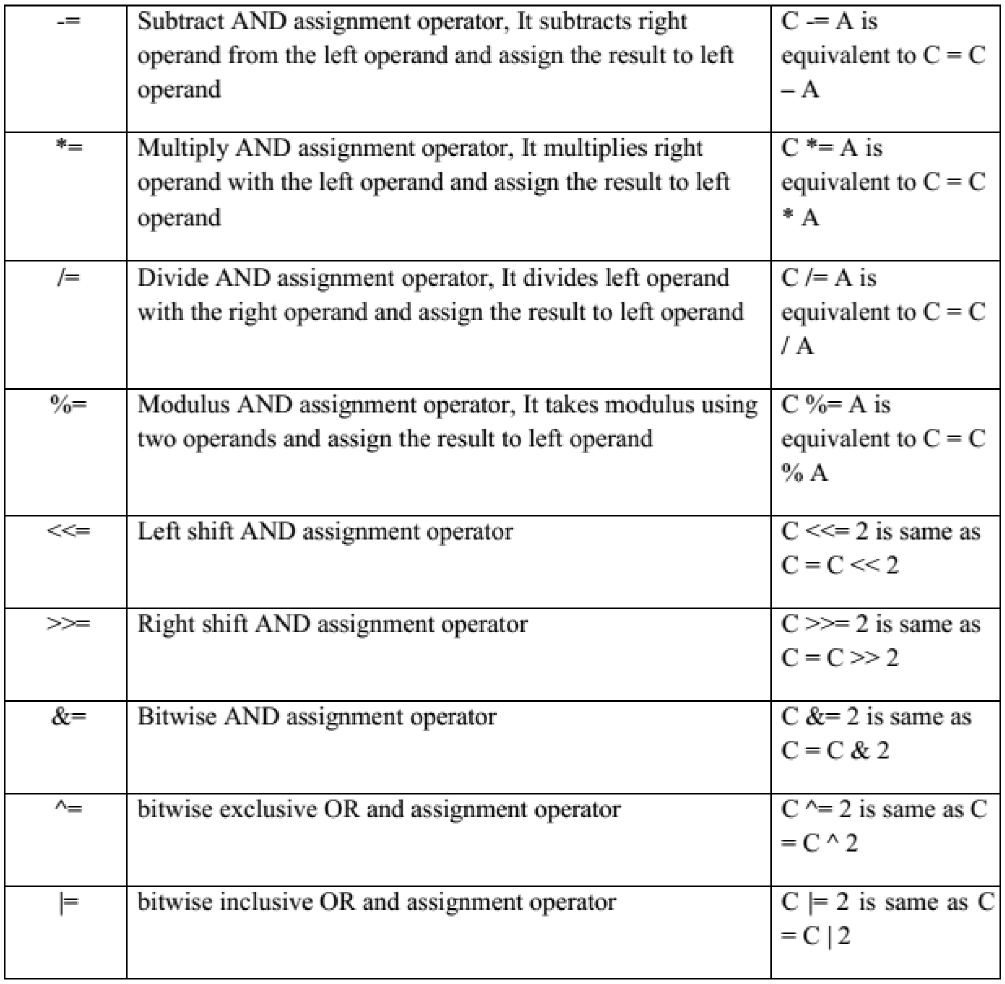
```
// Working of logical operators
#include <stdio.h>
int main()
{
   int a = 5, b = 5, c = 10, result;

   result = (a == b) && (c > b);
```

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

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

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

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

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

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

    return 0;
}
```

Output

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

4. Assignment Operator (=): Assignment operators are used to assign the result of an expression to a variable.



-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand	C -= A is equivalent to C = C – A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and assign the result to left operand	C %= A is equivalent to C = C % A
<<=	Left shift AND assignment operator	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator	C &= 2 is same as C = C & 2
^=	bitwise exclusive OR and assignment operator	C ^= 2 is same as C = C ^ 2
|=	bitwise inclusive OR and assignment operator	C |= 2 is same as C = C | 2

There are three types of assignment

1. Simple assignment. Ex a=5;
2. Shorthand assignment. Ex a+=5;
3. Multiple assignments Ex a=b=c=100;

An expression such as

i = i + 2;

in which the variable on the left side is repeated immediately on the right, can be written in the compressed form

i += 2;

The operator +=is called a shorthand assignment operator.

Note: the left-hand side of the assignment operator should always have variable A+b=5 is illegal

```
// Working of assignment operators
#include <stdio.h>
```

```
int main()
{
   int a = 5, c;

   c = a;      // c is 5
   printf("c = %d\n", c);
   c += a;     // c is 10
   printf("c = %d\n", c);
   c -= a;     // c is 5
   printf("c = %d\n", c);
   c *= a;     // c is 25
   printf("c = %d\n", c);
   c /= a;     // c is 5
   printf("c = %d\n", c);
   c %= a;     // c = 0
   printf("c = %d\n", c);

   return 0;
}
```

Output

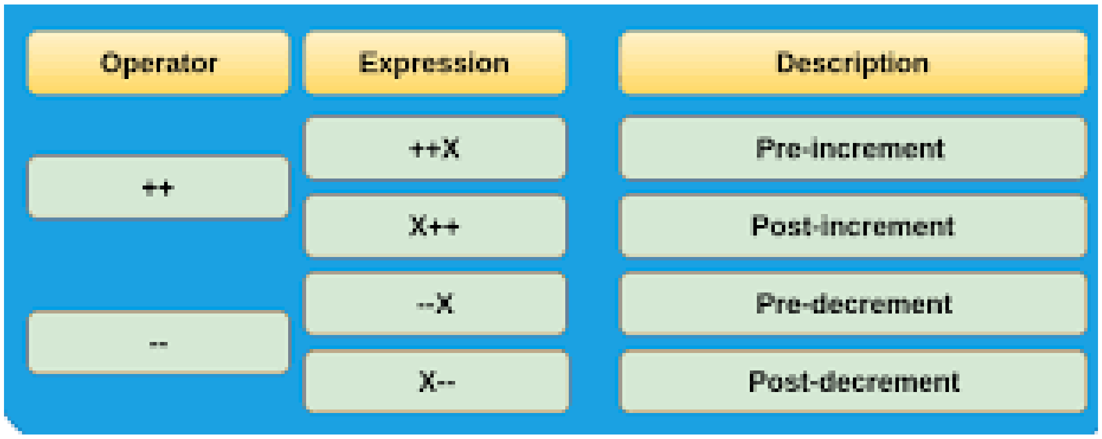
```
c = 5
c = 10
c = 5
c = 25
c = 5
c = 0
```

5. Increment and Decrement Operators: C uses two useful operators generally not found in other languages. These are increment and decrement operator. The operator ++ adds 1 to the operand and - - subtract 1. Both are unary operator and takes the form

++m or m++; equivalent to m=m+1

--m or m--; equivalent to m=m-1

These are extensively used in loops(while, for, do while)



Operator	Expression	Description
++	++X	Pre-increment
	X++	Post-increment
--	--X	Pre-decrement
	X--	Post-decrement

The following example illustrates the working principle behind post-increment/decrement and pre-increment/decrement operators:

Example-1:

int a=5,b; b=a++;

Here first the current value of a (i.e. 5) is copied to b then 'a' is incremented

$b=\boxed{5} \qquad a=\boxed{6}$

Example-2:

int x=7, y; y=++x;

Here first the value of x is incremented to 8 and then assigned/copied to y

$x=\boxed{8} \qquad y=\boxed{8}$

Example-3:

int a=6, b;

b= (a++) + (a++);

Steps of evaluation:

•First the current values of a is used to calculate the expression that is 6 +6 is calculated and stored in b.

•Then a gets incremented twice that is a=14 b= $\boxed{12}$ a= $\boxed{14}$

Example-4:

int a=6, b;

b= (++a) + (++a);

Steps of evaluation:

- First the values of a is incremented to 7
- Next a is again incremented to 8
- Final value of expression is 8+8
- This value is stored in b (i.e.16)

b= $\boxed{16}$ a= $\boxed{16}$

Example-5:
int a=5,b;

b=a- -;

Here first the current value of a (i.e. 5) is copied to b then ‘a’ is decremented

b= $\boxed{5}$ a= $\boxed{4}$

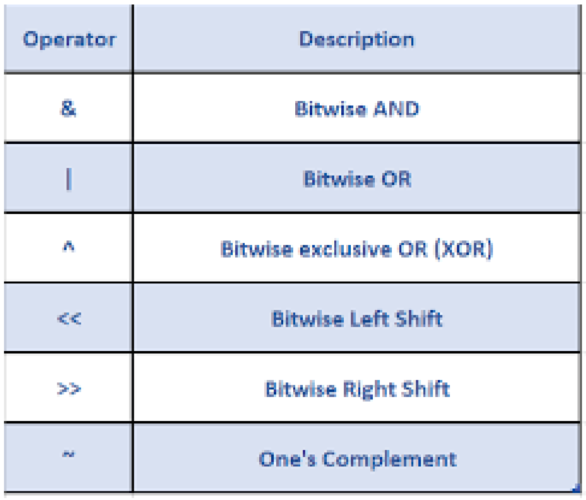
Example-6:

int x=7, y; y= - -x;

Here first the value of x is decremented to 6 and then assigned/copied to y

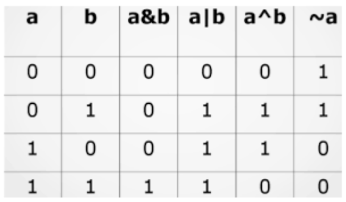
x= 6 y= 6

6. Bitwise Operators: C has a distinction of supporting special operator known as bitwise operators for manipulation of data at bit levels. These operators are used for testing the bits, or shifting them right or left. Bitwise operators may not be applied to float or double. Table below shows bitwise operators and their meaning.

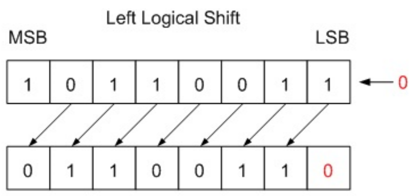


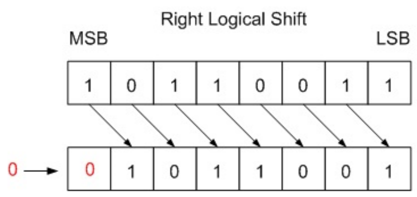
Operator	Description
&	Bitwise AND
|	Bitwise OR
^	Bitwise exclusive OR (XOR)
<<	Bitwise Left Shift
>>	Bitwise Right Shift
~	One's Complement

Examples of Bitwise operators

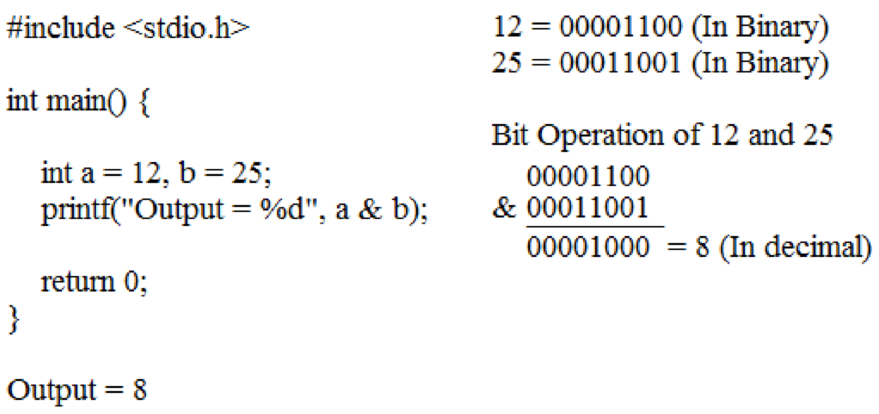


a	b	a&b	a|b	a^b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0





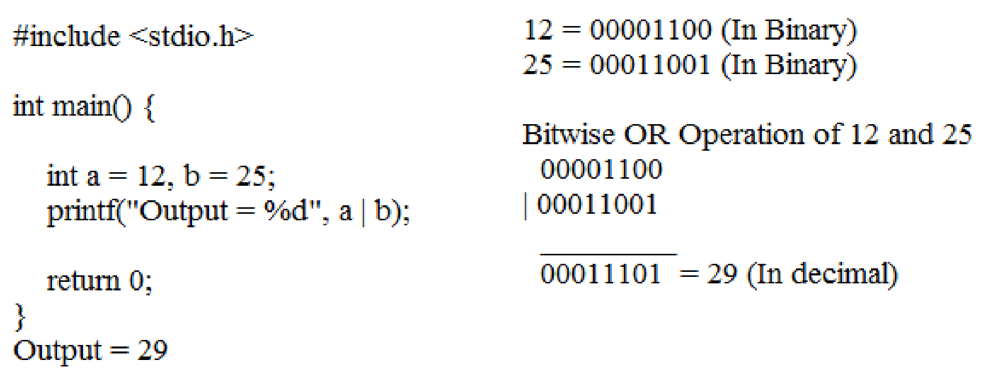
c code on Bitwise AND



```
#include <stdio.h>

int main() {

    int a = 12, b = 25;
    printf("Output = %d", a & b);

    return 0;
}

Output = 8
```

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

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

C code for Bitwise OR



```
#include <stdio.h>

int main() {

    int a = 12, b = 25;
    printf("Output = %d", a | b);

    return 0;
}
Output = 29
```

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

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

7. Conditional Operators [?:] Or Ternary Operator: They are also called as Ternary Operator. It takes 3 Arguments as shown in the syntax

$$\boxed{\textbf{Expression 1 ? Expression 2 : Expression 3}}$$

Where

Expression1 is Condition
Expression2 is Statement Followed if Condition is True
Expression2 is Statement Followed if Condition is False

Meaning of Syntax:

1. Expression1 is nothing but Boolean Condition i.e it results into either TRUE or FALSE
2. If result of expression1 is TRUE then expression2 is executed
3. Expression1 is said to be TRUE if its result is NON-ZERO
4. If result of expression1 is FALSE then expression3 is executed
5. Expression1 is said to be FALSE if its result is ZERO

Example : Check whether Number is Odd or Even

```
#include<stdio.h>
void main()
{
int num;
printf("Enter the Number : ");
scanf("%d",&num);

flag = ((num%2==0)?1:0);
if(flag==0)
printf("\nEven");
else
printf("\nOdd");
}
```

8 Special operator:
a) Comma Operator b) sizeof operator

1. Comma operator:
1.Comma operator has lowest precedence. [Priority] i.e., evaluated at last.
2.Comma operator returns the value of the rightmost operand.
3.Comma operator can acts as
- Operator: In the Expression
- Separator: Declaring Variable, In Function Call Parameter List

```
#include<stdio.h>
void main()
{
int a=1,b=2; int k;
k = (a,b); printf("%d",k);
}
Output: 2
```

In the Above Example –
1 : Comma as Separator
It can acts as Separator in –
Function calls
Function definations
Variable declarations
Enum declarations

2 : Comma as Operator k = (a , b);
Different Typical Ways of Comma as Operator :
int a=1,b=2,c;
Way 1 :
c = (a , b);
c = Value Stores in b = 2

Way 2 :
c = a , b;
c = Value Stores in a = 1

sizeof( ) operator: It is a compile time operator and when used with an operand it returns the number of bytes the operand occupies the operand may be variable or constant or data type qualifier.

Example:
```
int sum;
M=sizeof(sum);          //M has 2
N=sizeof(long int);   //N has 4
```

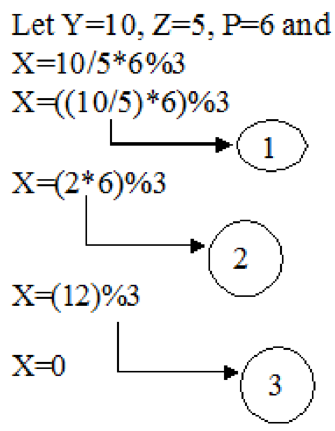
2.1.1 Operator Precedence and Associatively:

In previous section we came across expressions and various operators used in an expression. For instance if an expression consists of different category of operators such expression is evaluated using Precedence (priority) of operators and Associatively of operators.

Precedence: The order in which the operators in a complex expression are evaluated is determined by set of priorities known as precedence. If an expression contains Arithmetic operators '+' , '*' and '/ 'operators as shown below:
A=B+C*D/F
In this expression RHS of expression contains B+C*D/F, here first preference is given to '*' and '/ 'and then to '+'. This is decided by operator precedence given in C Language.

Let us say: B=5, C=7, D=9 and F=3, then value of A is:
A= 5+7*9/3
This expression is evaluated in following steps:
A = 5 + (7*9)/3 A= 5+(63/3) A=5+21
A=26

Associativity: If two operators with same precedence accur in a complex expression, another attribute of an operator called associativity takes control. Associativity is the parsing direction used to evaluate an expression. It can be either left to right or right to left .
For example: X= Y/Z*P%Q
Here '/', '*', and '%' are operators at same level. But we evaluate this expression from LEFT to RIGHT (i.e. Associativity is from Left to Right).


Let Y=10, Z=5, P=6 and Q=3
X=10/5*6%3
X=((10/5)*6)%3
X=(2*6)%3
X=(12)%3
X=0

Following table provides the Precedence and Associativity of operators discussed in this unit.

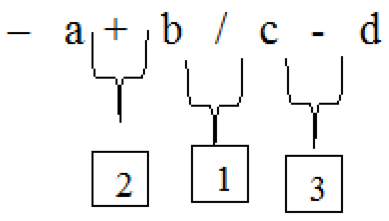
Operator	Description	Associativity	Precedence(Rank)
( ) [ ]	Function call Array element reference	Left to right	1
+ - ++	Unary plus Unary minus Increment Decrement	Right to left	2

-- ! ~ * & sizeof(type)	Logical negation Ones complement Pointer to reference Address Size of an object Type cast (conversion)		
* / %	Multiplication Division Modulus	Left to right	3
+ -	Addition Subtraction	Left to right	4
<< >>	Left shift Right Shift	Left to right	5
< <= > >=	Less than Less than or equal to Greater than Greater than or equal to	Left to right	6
== |=	Equality Inequality	Left to right	7
&	Bitwise AND	Left to right	8
^	Bitwise XOR	Left to right	9
|	Bitwise OR	Left to right	10
&&	Logical AND	Left to right	11
||	Logical OR	Left to right	12
?:	Conditional expression		13
= *= /= %= += -= &= ^= |= <<= >>=	Assignment operators		14
,	Comma operator	Left to right	15

Evaluation of Expressions

Based on the Precedence and Associativity of operators here we discuss evaluating various types of expressions.

int a=10, b=7, c=8, d=2;

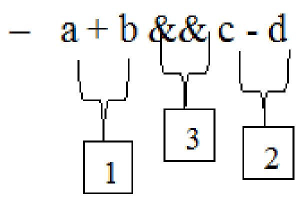


10 + (7/8) – 2
(10+ 0) - 2
(10 – 2)
8

Note: In this example 7/8 gives 0 as answer as it is integer division the decimal portion is truncated.

Example-2:.

int a=10, b=7, c=8, d=2;



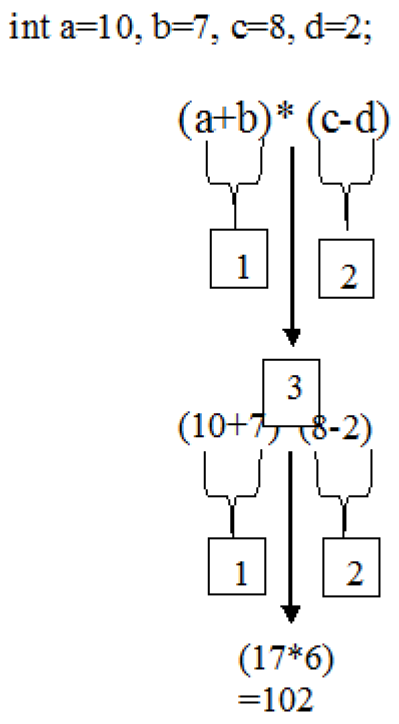
(10+7) && 8-2

(17) && (8-2)

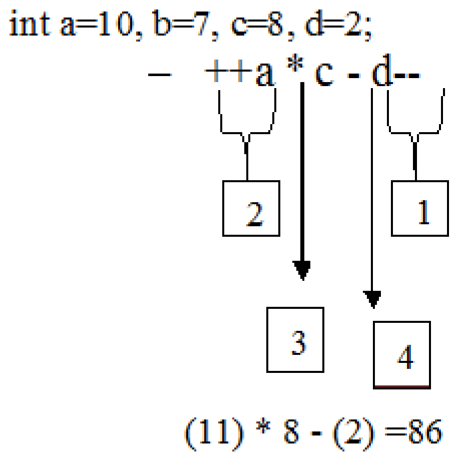
(17) && (6) Answer:
TRUE (1)

Note: Any non-zero number is treated as 1 (TRUE) whenever Logical operators are used.

Example-3:



Example-4:



Note: Here d - - is post decrement so its initial value (2) is used in the expression.

2.2 Types of Conversions

Whenever mixed data occurs ‘type conversion’ comes into picture. Two types of data type conversions are:

1. Automatic type conversions (Implicit conversions)
2. Manual type conversions (Explicit conversions)

In automatic conversion the operand/variable of smaller data type is automatically converted to data type of larger size.

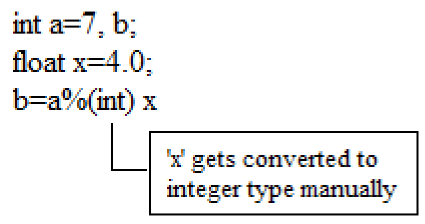
That is:

$$\begin{tikzcd}\text{char} \arrow[r, Rightarrow] & \text{int} \arrow[r, Rightarrow] & \text{long int} \arrow[r, Rightarrow] & \text{float} \arrow[r, Rightarrow] & \text{double} \arrow[r, Rightarrow] & \text{long double}\end{tikzcd}$$

Example:
int a=25;
float x=5,z;
z=x/a;
printf("%f",z);

In this example 'x' is float and 'a' is integer, a gets automatically converted to float and answer is: z=(5.0/25.0) 0.2

Side effects of automatic conversion:
int a=7, b;
float x;
b=a%x
In the example given above we are using modulus or remainder (%) operator.
Here 'a' is integer type and 'x' is float. But the catch is % can be used only with integers. So 'x' has to be automatically converted to float. But it is impossible as float is bigger than integer. As a result Complier gives syntax error.
If we have to convert a variable of bigger size to smaller type we have to use manual conversion (explicit conversion).
Here is how we can use explicit conversion and overcome the syntax error in previous example:
Example:



```
int a=7, b;
float x=4.0;
b=a%(int) x
```

'x' gets converted to integer type manually

Output is: b=7%4 □3

Note: This type of explicit conversion is also referred to as TYPE CASTING in 'C'

Another example of side effects in type conversions is here: Example:
float a=25, b=4;
int x; x=a/b;
The problem here is though 25.0/4.0 will result in 6.25, it is stored in an integer variable 'x'. As a result 6.25 is truncated to 6! Therefore we have to be careful when mixed data types are used.