Unit-I
Chapter-3

Operators & Expressions: C operators and expressions, Type-conversion methods, Operators Precedence and Associativity.

Operator specifies which operation we need to perform.
Operands are the values that we perform operation on those values.

Precedence:
If there is more than one operator in an expression, which operation we need to perform first is specified by precedence.

Associativity:
If more than one operator has the same precedence which operation we need to perform is specified by associativity.

Types of operators based on number of operands:
We have three types of operators

1. Unary Operators – require one operand.
2. Binary Operators- require two operands.
3. Ternary Operators – require three operands.

- An operator is a symbol that tells the compiler to perform certain mathematical or logical manipulations.
- Operators are used in programs to manipulate data and variables.
- They usually form a part of the mathematical of logical expressions.

'C' operators can be classified into 8 categories.

1. Arithmetic operators.
2. Relational operators.
3. Logical operators.
4. Assignment operators.
5. Increment and decrement operators.
6. Conditional operators.
7. Bitwise operators.
8. Special operators.

Arithmetic operators:

'C' provides all the basic arithmetic operators.

Operator	Meaning
+	Addition or Unary plus
-	Subtraction or Unary minus
*	Multiplication
/	Division
%	Modulo division

Unary minus example: -2, 2*-1

Note: Modulo division operator (%) is used only for integral values and cannot be used on floating point data.

*, /, % ------ highest precedence
+, - ------ lowest precedence

Arithmetic operator's associativity is from Left to Right.

i. In **integer arithmetic**, all operands are integer.
   Example: 15/4=3
ii. In **real arithmetic**, all operands are real.
   Example: 15.0/4.0=3.75
iii. In **mixed mode arithmetic**, one operand is real and another operand is integer then that expression is called mixed mode arithmetic.

If either operand is of the real type, then only the real operation is performed and the result is always a real number.
Example: 15.0/4=3.75 or
15/4.0=3.75

Example program:

```
#include<stdio.h>
main()
{
        int a,b;
        floatc,d;
        printf("enter 2 integer values\n");
```

```
    scanf("%d%d",&a,&b);
    printf("enter 2 float values\n");
    scanf("%f%f",&c,&d);
    printf("Integer Arithmetic %d/%d=%d\n",a,b,a/b);
    printf("real Arithmetic %f/%f=%f\n",c,d,c/d);
    printf("Mixed mode Arithmetic %f/%d=%f\n",c,b,c/b);
}
```

Relational operators:

- These operators are used to compare the quantities.
- These operators are used to identify the relation between quantities.

Operator	Meaning
<	is less than
<=	is less than or equal
>	is greater than
>=	is greater than or equal
==	is equal to
!=	is not equal to

Form:

ae-1 relational operator ae-2

Where **ae-1** and **ae-2** are arithmetic expressions, which may be simple constants, variables, or combination of them.

The expression containing a relational operator is termed as relational expression.
The value of the relational expression is either one or zero.
It is one if the specified relation is true and zero if the relation is false.
Examples:

Relational Expression	Result
10<20	1(true)
20<=10	0(false)
10>5	1(true)
10>=4	1(true)
10==10	1(true)
10!=10	0(false)

<, <=, >, >= ---------- highest precedence
==,!= ---------- lowest precedence

Relational expressions are used in *decision statements* such as, **if** and **while** to decide the course of a running program.

Arithmetic operators have a higher priority than relational operators.
Associativity is from Left to Right.

Example program:

```
#include<stdio.h>
main()
{
        int a,b;
        printf("enter a,b\n");
        scanf("%d%d",&a,&b);
        printf("%d<%d=%d\n",a,b,a<b);
        printf("%d<=%d=%d\n",a,b,a<=b);
        printf("%d>%d=%d\n",a,b,a>b);
        printf("%d>=%d=%d\n",a,b,a>=b);
        printf("%d==%d=%d\n",a,b,a==b);
        printf("%d!=%d=%d\n",a,b,a!=b);
}
```

Logical operators:

Operator	Meaning
&&	Logical AND
||	Logical OR
!	Logical NOT

The logical operators && and || are used when we want to test more than one condition and make decisions.

The logical expression:

An expression which combines 2 or more relational expressions is termed as a logical or compound relational expression.

Logical expression yields a value of one or zero, according to the truth table.

Truth Table:

op1	op2	op1 && op2	op1 || op2
Non-zero	Non-zero	1	1
Non-zero	0	0	1
0	Non-zero	0	1
0	0	0	0

Example program:

```
#include<stdio.h>
main()
{
        int marks,a,b;
        printf("enter marks\n");
        scanf("%d",&marks);
        printf("enter 2 int values\n");
        scanf("%d%d",&a,&b);
        printf("marks>=90 && marks<=100: %d\n",marks>=90 &&
        marks<=100);
        printf("a>=b || b>=a=%d\n",a>=b || b>=a);
        printf("!a=%d\n",!a);
}
```

Local NOT ---------- highest precedence, **Associativity from Right-> Left**
Logical AND ---------- next highest precedence, **Associativity from Left -> Right**
Logical OR ---------- lowest precedence, **Associativity from Left -> Right**

Assignment operators:
Assignment operators are used to assign the result of an expression to a variable.
Assignment operator is '**=**'.

Example:
a=10;
Left hand side must be a variable; Right hand side may be a variable/constant or combination of these two.

'C' has a set of '**shorthand**' assignment operators of the form

v **op=**exp;

Where v is a variable, exp is an expression and op is C binary arithmetic operator.
The operator **op=**is known as the **shorthand assignment operator**.
The assignment statement

vop= exp;

is equivalent to

v = v op (exp);

Associativity is from Right to Left.

Statement with simple assignment operator	Statement with shorthand operator
a =a + 100	a += 100
a = a - 20	a -= 20
a = a * (n+1)	a *= (n+1)
a = a / (n+1)	a /= (n+1)
a = a % b	a %= b

The use of shorthand assignment operators has three advantages:

1. What appears on the left hand side need not be repeated and therefore it becomes easier to write.
2. The statement is more concise and easier to read.
3. The statement is more efficient.

Example program:

```
#include<stdio.h>
main()
{
        int a,n;
        printf("enter a,n values\n");
        scanf("%d%d",&a,&n);
        a += 100;
        printf("a=%d\n",a);
        a -= 20;
        printf("a=%d\n",a);
        a *= (n+1);
        printf("a=%d\n",a);
```

```
        a /= (n+1);
        printf("a=%d\n",a);
        a %= n;
        printf("a=%d\n",a);
}
```

Increment and Decrement operators:

- 'C' has two very useful operators not generally found in other languages.
- These are increment and decrement operators:
  ++ and --
- The operator ++ adds 1 to the operand.
- The operator -- subtracts 1 from the operand.
- Both are unary operators.
- We use the increment and decrement statements in **for** and **while** loops extensively.

Statement with Operator	Meaning
++a	Pre increment
a++	Post increment
--a	Pre decrement
a--	Post decrement

- Both ++a and a++ mean the same thing when they form statements independently.
- They behave differently when they are used in expressions on the R.H.S. of an assignment statement.

Statement with Operator used in assignment	Meaning	Explanation
b = ++a	Pre increment	**First adds1** to a and then the result is assigned value of a to b
b = a++	Post increment	**First assigns** the value of a to b and then add 1 to a
b = --a	Pre decrement	**First subtracts 1** to a and then the result is assigned value of a to b
b = a--	Post decrement	**First assigns** the value of a to b and then subtract 1 to a

Associativity is from **Right -> Left**.

Example Program:

```
#include<stdio.h>
main()
{
        int a,n;
        printf("enter a value\n");
        scanf("%d",&a);
        n=a++;
        printf("a=%d\tn=%d\n",a,n);
        n=++a;
        printf("a=%d\tn=%d\n",a,n);
        n=a--;
        printf("a=%d\tn=%d\n",a,n);
        n=--a;
        printf("a=%d\tn=%d\n",a,n);
}
```

Output:

```
enter a value
4
a=5    n=4
a=6    n=6
a=5    n=6
a=4    n=4
```

Conditional operator:

A ternary operator pair "**?:**"is available in C to construct conditional expressions.
Form is

exp1 ? exp2 : exp3;

Where exp1, exp2, exp3 are expressions.

Working of conditional operator "?:" :

i. exp1 is evaluated first. If it is non-zero (true), then the expression exp2 is evaluated and that is the value of the expression.
ii. If exp1 is zero (false), exp3 is evaluated and it is the value of the expression
iii. Note that only one of the expressions (either exp2 or exp3) is evaluated.

Example:
int a=10, b=15, x;
x=(a<b) ? a : b;

Here, (10<15) evaluates to True, then x=a=10

Result:
x=10

Associativity is from **Right -> Left**.

```
//Program to find whether the given number is even or odd using conditional operator
#include<stdio.h>
main()
{
int n;
printf("enter n value\n");
scanf("%d",&n);
      (n%2==0)?printf("%d is even number\n",n):printf("%d is odd number\n",n);
}
```

Output 1:

```
enter n value
4
4 is even number
```

Output 2:

```
enter n value
5
5 is odd number
```

Example:

```
#include<stdio.h>
main()
{
float a,b,c,d;
printf("enter a,b,c,d values\n");
scanf("%f%f%f%f",&a,&b,&c,&d);
      (c-d!=0)?printf("%f",(a+b)/(c-d)):printf("c-d is 0");
}
```

Output 1:
entera,b,c,d values
3
4
5
2
2.333333
Output 2:
entera,b,c,d values
3
4
5
5
c-d is 0

Bitwise operators:

- For manipulation of data at bit level, a special operators known as bitwise operators are introduced.
- These operators are used for testing the bits or shifting them right or left.
- Bitwise operators may not be applied to float or double.

Operator	Meaning
&	Bitwise AND
|	Bitwise OR
^	Bitwise Exclusive OR
<<	Shift Left
>>	Shift Right
~	One's Complement

Truth table for Bitwise AND, Bitwise OR and Bitwise Exclusive OR:

op1	op2	op1 & op2	op1 | op2	op1 ^ op2
1	1	1	1	0
1	0	0	1	1
0	1	0	1	1
0	0	0	0	0

Truth table for Bitwise NOT:

op1	! op1
1	0
0	1

Bitwise AND (&):

Example 1:
x=9
y=7
First convert given decimal number to binary number and then perform bitwise operation.

x=1001
y=0111
---
z= 0001

Next convert the binary number to decimal number to get final result

z = **9 & 7=1**

Example2:

x=4
y=14

x=0100
y=1110
---
z=0100 = 4

z = **4 & 14=4**

Bitwise OR (|):

Example 1:
x=8
y=7
First convert given decimal number to binary number and then perform bitwise operation.

x=1000
y=0111
---
z=1111

Next convert the binary number to decimal number to get final result

z = **8| 7=15**

Example2:

x=4
y=14

x=0100
y=1110
---
z=1110 = 14
z = **4 | 14=4**

Bitwise Exclusive-OR (^):

Example 1:
x=5
y=7
First convert given decimal number to binary number and then perform bitwise operation.
x=101
y=111
---
z=010

Next convert the binary number to decimal number to get final result

z = **5 ^ 7=2**

Example2:

x=4
y=14

x=0100
y=1110
---
z=1010 = 10

z = **4 ^ 14=4**

Bitwise Complement Operator (~):

Example 1:
Find complement of 5 (~5)

Step1:
Convert decimal 5 into binary and take 8 bits.
0000 0101
First perform bits complement means change 1's as 0's and 0's as 1's (1's complement).

1111 1010
MSB LSB

Step2:
If the MSB bit of result of bits complement is 1, then the final result contains –ve sign. Always negative numbers in the system are represented using 2's complement form.

Finding 2's complement means 1's complement + 1.

0000 0101 ---------- 1's complement
+1 (in binary addition 1+1 = 10 means 1 as carry and 0 as sum)

0000 0110 ---------- 6

Therefore final result is **-6**

Note:
(In binary addition 1+1 = 10 means 1 as carry and 0 as sum, 1+0=1, 0+0=0)

Example 2:
Find complement of -5 (~-5)

Step1:
Always negative numbers in the system are represented using 2's complement form.

First take +5 in binary form, find 2's complement of that number.

0000 0101
1111 1010
+1

1111 1011 ---------- -5 binary equivalent

Step2:

Perform bits complement (1's complement) on step1 result.
0000 0100 ---------- 4
If the MSB bit of the result contains 0, then the resulting number have the + sign.
(~-5)=+4

Bitwise Shift operators:

The shift operators are used to move bit patterns either to the left or to the right.

Left shift: v << n
Right shift: v >> n

Where v is the integer expression (value) that is to be shifted and n is the number of bit positions to be shifted.

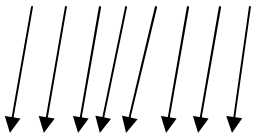
Left shift:
Left shift means given number multiply by 2 per each shift.
Example:
6<<1 or x=6, x<<1

First convert decimal 6 to binary and take 8 bits.

MSB LSB

87654321

0 0 0 0 0 1 1 0



0 0 0 0 0 1 1 0

8 7 6 5 4 3 2 1

Here, 12345678 are bit positions for understanding purpose.

- First move 1st bit as 2nd bit, 2nd bit as 3rd bit, 3rd bit as 4th bit, 4th bit as 5th bit, 5th bit as 6th bit, 6th bit as 7th bit, 7th bit as 8th bit.
- Here, 1st bit is empty, make it as 0. Remove 8th bit because it position exceeds 8 bit positions.

0 0 0 0 1 1 1 0

8 7 6 5 4 3 2 1

This is the final bit result. Convert binary to decimal and it is equal to decimal 12.
6 << 1 = 12

Special operators:

C supports some special operators of interest such as:

i. Comma operator (,)
ii. sizeof operator
iii. pointer operators (&, *)
iv. member selection operators (., ->)
v. function call symbol ()
vi. array subscript operator []

Comma operator:

- This can be used to link the related expressions together.
- Comma liked lists of expressions are evaluated from left to right. And the value of right most expression is the value of the combined expression.
- Comma operator has the lowest precedence of all operators, the parenthesis are necessary.

Example:

value=(x=10, y=5, x+y)
Therefore value=15.

The **sizeof operator**:

- The sizeof operator is a compile time operator.
- When used with an operand, it returns the number of bytes the operand occupies.
- The operand may be a variable, a constant or a data type qualifier.
- This is normally used to determine the lengths of arrays and structures.

& ------- address operator or reference operator, it is unary operator.
***** -------- dereference operator or pointer reference or indirection operator.

Expressions:

Arithmetic expressions:

An arithmetic expression is a combination of variables, constants, and operators arranged as per the syntax of the language.

Examples:

Algebraic expression	C expression
a × b – c	a*b-c
(m + n)(x + y)	(m+n)*(x+y)
$\frac{ab}{c}$	a*b/c
$3x^2+2x+1$	3*x*x+2*x+1
$\frac{x}{y}+c$	x/y+c

Evaluation of expressions:
Expressions are evaluated using an assignment statement.

variable = expression

Where variable is any valid 'C' variable name.

Example:
a=9, b=12, c=3
x = a – b / 3 + c * 2 – 1
x = 9 – 12 / 3 + 3 * 2 – 1

Here, **/, *** having the **same priority** and **highest priority**, so **follow** the **associativity**, it is **Left -> Right**.

Step1: evaluate 12 / 3 = 4

x = 9 – 4 + 3 * 2 – 1

Step 2: evaluate 3 * 2 = 6

x = 9 – 4 + 6 – 1

Next, **-, +** having the **same priority**, so **follow** the **associativity**, it is from **Left -> Right**.

Step3: 9 – 4 = 5

x = 5 + 6 – 1

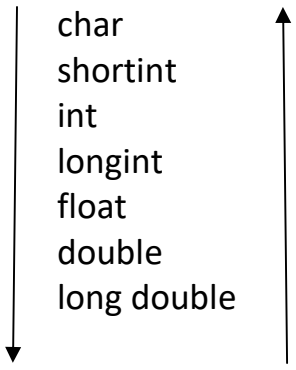
Step4: 5 + 6 = 11

x = 11 – 1

Step5: 11 – 1 = 10

Finally **x = 10**

<u>Type conversion</u>:



Conversion of small data type to big is broadening.
Conversion of big data type to small is narrowing.
(small and big in terms of number of bytes)

Conversions are of two types:

1. Implicit type conversion
2. Explicit type conversion

Implicit type conversion:

- Computer considers one operator at a time, involving 2 operands.
- **If the operands are of different types, the 'lower' type is automatically converted to the 'higher' type before the operation proceeds**.
- **The result is of the 'higher' type**.
- **The final result of an expression is converted to the type of the variable on the left of the assignment sign before assigning the value to it.**

- The following rules are followed:
  i. float to int causes truncation of the fractional part.
  ii. double to float causes rounding of digits.
  iii. longint to int causes dropping of the excess higher order bits.

Example:

```
#include<stdio.h>
main()
{
        int a;
        floatb,c;
        printf("enter a,b values\n");
        scanf("%d%f",&a,&b);
        c=a/b;
        printf("c=%f\n",a/b);
}
```

Output:

```
entera,b values
4
5
c=0.800000
```

Explicit type conversion or casting a value:

For example,

```
#include<stdio.h>
main()
{
        int a,b;
        float c;
        printf("enter a,b values\n");
        scanf("%d%d",&a,&b);
        c=a/b;
        printf("c=%f\n",a/b);
}
Output:
entera,b values
4
5
c=0.000000
```

Consider the statement c=a/b;

- Here a, b are integer data type, so integer division is performed. The result of this division is 0. So, the decimal part of the division would be lost and the result is not correct.
- This problem can be solved by converting locally one of the variables to the floating point.

  c**=(float)**a/b;

- The **operator (float)** converts the 'a' to floating point for the purpose of evaluation of the expression.
- Then automatically conversion is done and then the division is performed in floating point mode.
- This process of such a local conversion is known as casting a value.
- Form is

(type name) expression

Where type name is one of the valid 'C' data types and the expression may be a constant, variable or an expression.

```
// Example program using explicit type conversion
#include<stdio.h>
main()
{
        int a,b;
        float c;
        printf("enter a,b values\n");
        scanf("%d%d",&a,&b);
        c=(float)a/b;
        printf("c=%f\n",c);
}
```

Output:

```
enter a,b values
4
5
c=0.800000
```

Usage of type conversion:

Example	Action
x = (int) 4.78	4.78 is converted to integer by truncation and result would be 4
a=(int)13.4/4	Evaluated as 13/4 and the result would be 3
d=(double)12/5	Division is done in floating point mode and result would be 2.400000

Operators Precedence and Associativity Table:

Operator	Description	Associativity	Rank
( ) [ ]	Function call Array element reference	Left to Right	1
+ - ++ -- ! ~ * & sizeof (type)	Unary plus Unary minus Increment Decrement Logical NOT Ones complement Pointer reference (indirection) Address Size of an object Type cast (conversion)	Right to Left	2
* / %	Multiplication Division Modulo 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
== !=	Is equal to Not equal to	Left to Right	7
&	Bitwise AND	Left to Right	8
^	Bitwise XOR (Exclusive OR)	Left to Right	9
|	Bitwise OR	Left to Right	10
&&	Logical AND	Left to Right	11
||	Logical OR	Left to Right	12
?:			13
= *=, /=, %= +=, -= &=, ^=, |= <<=, >>=	Assignment operators	Right to Left	14
,	Comma operator	Left to Right	15