Unit 4. Operators and Expression

- 4.1 Arithmetic operator,
- 4.2 Relational operator,
- 4.3 Logical or Boolean operator,
- 4.4 Assignment Operator,
- 4.5 Ternary operator,
- 4.6 Bitwise operator,
- 4.7 Increment or Decrement operator,
- 4.8 Conditional operator,
- 4.9 Special Operators(sizeof and comma),
- 4.10 Evaluation of Expression (implicit and explicit type conversion),
- 4.11 Operator Precedence and Associativity.

Arithmetic Operator

- Arithmetic operators in C are used to perform mathematical operations such as addition ,subtraction,multiplication etc.
- Arithmetic operators are divided in two parts:
  - *Unary arithmetic operators : operators taking only one operand*
  - *Binary arithmetic operators* : operators taking two operands.
- Unary arithmetic operators are

Operator	**Symbol**	**Action**	**Examples**
Increment	++	Increments the operand by one	**++x, x++**
Decrement	**--**	**Decrements the operand by one**	**--x, x--**

Increment /Decrement Operator

- The increment and decrement operators can be used only with integer variables, not with constants.

- The operation performed is to add one to or subtract one from the operand.
- In other words, the statements : ++x; --y;

  are the equivalent of these statements: x = x + 1;
  y = y - 1;

- Note that either unary operator can be placed before its operand (prefix mode) or after its operand (postfix mode).

- These two modes are not equivalent. They differ in terms of when the increment or decrement is performed:

Increment /Decrement Operator

- When used in prefix mode, the increment and decrement operators modify their operand before it's used.

  x = 10;   y = ++x;

  Here value of x is incremented first i.e. x becomes 11 and the incremented value is assigned to y.
- So after execution of those statement x and y both have value 11.
- When used in postfix mode, the increment and decrement operators modify their operand after it's used.
- For example : Consider following statements:

  x = 10; y = x++;
- After these statements are executed, x has the value 11, and y has the value 10.
- The value of x was assigned to y, and then x was incremented.

Increment /Decrement Operator

Example:

```
#include <stdio.h>
int a, b;
main()
{
     /* Set a and b both equal to 5 */
     a = b = 5;
     /* Print them, decrementing each time. */
     /* Use prefix mode for b, postfix mode for a */
   printf("\n%d   %d", a--, --b);
   printf("\n%d   %d", a--, --b);
   printf("\n%d   %d", a--, --b);
   printf("\n%d   %d", a--, --b);
   printf("\n%d   %d\n", a--, --b);
  return 0;
}
```

output:

```
5   4
4   3
3   2
2   1
1   0
```

Binary Arithmetic Operators

- C language has five binary arithmetic operators. Following table includes the binary arithmetic operators.

Operator	Symbol	Action	Example
Addition	+	Adds two operands	x + y
Subtraction	-	Subtracts the second operand from the first	x – y
ultiplication	*	Multiplies two operands	x * y
Division	/	Divides the first operand by the second operand	x / y
Modulus	%	Gives the remainder when the first operand is divided by the second operand	x % y

- The first four operators listed in Table should be familiar to us in general mathematics. The fifth operator, modulus operator returns the remainder when the first operand is divided by the second operand. For example, 11 modulus 4 equals 3 . Here are some more examples: 100 %9 equals 1 10 %5 equals 0   40 % 6 equals 4

```
#include <stdio.h>              /* Example of Binary arithmetic operators */
 main()
 {
        int a,b,r;   float r1;
        printf("Enter First integers:");       /* Input two integers */
        scanf("%d",&a);
         printf("Enter Second integer:");
        scanf("%d",&b);
        r=a+b;
        printf(“\nSum of %d and %d = %d",a,b,r);
        r=a-b;
        printf("\nDifference of %d and %d = %d",a,b,r);
        r=a*b;
        printf("\nProduct of %d and %d= %d",a,b,r);
        r1=a/b;
        printf("\nThe Quotient = %f",r1);
   r=a%b;
   printf("\nThe remainder = %d",r);
   return 0;
}
```

Output

```
Enter First integers:34
Enter Second integer:7

Sum of 34 and 7 = 41
Difference of 34 and 7 = 27
Product of 34 and 7= 238
The Quotient = 4.000000
The remainder = 6
```

Operator Precedence and and use od Parentheses

- Operator Precedence is the order in which operations are performed.
- In an expression that contains more than one operator, then the precedence plays the important role for evaluation.

  e.g. x = 4 + 5 * 3;

- Performing the addition first results in the following, and x is assigned the value 27 as below:

  x = 9 * 3;

- if the multiplication is performed first, x is assigned the value 19:

  x = 4 + 15;

- So some rules are needed about the order in which operations are performed.
- When an expression is evaluated, operators with higher precedence are performed first. Following Table lists the precedence of C's mathematical operators.

Operators	Relative Precedence
++ --	1
* / %	2
+ -	3

Precedence of operators: continued…….

In any C expression, operations are performed in the following order:

Unary increment and decrement

Multiplication, division, and modulus

Addition and subtraction

- If an expression contains more than one operator with the same precedence level, the operators are performed in left-to-right order as they appear in the expression called associativity of operator. For example, in the following expression, the % and * have the same precedence level, but the % is the leftmost operator, so it is performed first:

  12 % 5 * 2
- A sub-expression enclosed in parentheses is evaluated first, ignoring to operator precedence. Thus, we could **write x = (4 + 5) * 3;**
- The expression 4 + 5 inside parentheses is evaluated first, so the value assigned to x is 27.
- We can use multiple and nested parentheses in an expression. When parentheses are nested, evaluation proceeds from the innermost expression outward. Look at the following complex expression:

  x = 25 - (2 * (10 + (8 / 2)));
- The evaluation of this expression proceeds as follows:

1. The innermost expression, 8 / 2, is evaluated first, yielding the value 4: **25 - (2 * (10 + 4))**
2. Moving outward, the next, 10 + 4, is evaluated, yielding the value 14: **25 - (2 * 14)**
3. The last, or outermost, expression, 2 * 14, is evaluated, yielding the value 28: **25 - 28**
4. The final expression, 25 - 28, is evaluated, assigning the value -3 to the variable x**: x = -3**

Relational Operators

- C's relational operators are used to compare expressions, asking questions such as, "Is x greater than 100?" or "Is y equal to 0?"
- An expression containing a relational operator evaluates to either true (1) or false (0).

Table 1: C's relational operators.

Operator	Symbol	Question Asked	Example
Equal	==	Is operand 1 equal to operand 2?	x == y
Greater than	>	Is operand 1 greater than operand 2?	x > y
Less than	<	Is operand 1 less than operand 2?	x < y
Greater than or equal to	>=	Is operand 1 greater than or equal to operand 2?	x >= y
Less than or equal to	<=	Is operand 1 less than or equal to operand 2?	x <= y
Not equal	!=	Is operand 1 not equal to operand 2?	x != y

Table 2: Relational operators in use.

Expression	How It Reads	What It Evaluates To
5 == 1	Is 5 equal to 1?	0 (false)
5 > 1	Is 5 greater than 1?	1 (true)
5 != 1	Is 5 not equal to 1?	1 (true)
(5 + 10) == (3 * 5)	Is (5 + 10) equal to (3 * 5)?	1 (true)

An example: Demonstrates the evaluation of relational expressions

```
#include <stdio.h>
int a;
main()
{
        a = (5 == 5);                    /* Evaluates to 1 */
        printf("\na = (5 == 5) evaluates a = %d", a);

        a = (5 != 5);                    /* Evaluates to 0 */
        printf("\na = (5 != 5) evaluatesa = %d", a);

        a = (12 == 12) + (5 != 1);  /* Evaluates to 1 + 1 */
        printf("\na = (12 == 12) + (5 != 1) evaluates a = %d\n", a);
        return 0;
}
```

Output:

```
a = (5 == 5) evaluates  a = 1
a = (5 != 5) evaluates a = 0
a = (12 == 12) + (5 != 1)  evaluates a = 2
```

The Precedence of Relational Operators

- The precedence of relational operators determines the order in which they are performed in a multiple-operator expression.
- However, We can use parentheses to modify precedence in expressions that use relational operators.
- First, all the relational operators have a lower precedence than the arithmetic operators.
- Thus, if we write the following, 2 is added to x, and the result is compared to y: **`(x + 2 > y)`** `equivalent to` **`((x + 2) > y)`** which is a good example of using parentheses for the clarity:
- There is also a two-level precedence within the relational operators, as shown in Table below:

Operators	Relative Precedence
< <= > >=	1
!= ==	2

- Thus, if we write **`x==y>z`** it is the same as **`x==(y>z)`** because C first evaluates the expression y > z, resulting in a value of 0 or 1.
- Next, C determines whether x is equal to the 1 or 0 obtained in the first step.

Logical Operators

- Sometimes we need to ask more than one relational question at once. The logical operators let us combine two or more relational expressions into a single expression that evaluates to either true or false.
- Below table shows the logical operators in C.

Operator	Symbol	Example
AND	&&	exp1 && exp2 e.g. (num>100)&&(num%2==0)
OR	||	exp1 || exp2 e.g. (num<100)||(num%2!=0)
NOT	!	!exp1 e.g. num!=5

- **Table: C's logical operators in use.**

Expression	What It Evaluates To
(exp1 && exp2)	True (1) only if both exp1 and exp2 are true; false (0) otherwise
(exp1 || exp2)	True (1) if either exp1 or exp2 is true; false (0) only if both are false
(!exp1)	False (0) if exp1 is true; true (1) if exp1 is false

Logical Operators

Table Code examples of C's logical operators.

Expression	What It Evaluates To
(5 == 5) && (6 != 2)	True (1), because both operands are true
(5 > 1) || (6 < 1)	True (1), because one operand is true
(2 == 1) && (5 == 5)	False (0), because one operand is false
!(5 == 4)	True (1), because the operand is false

- We can create expressions that use multiple logical operators. For example, to ask the question "Is x equal to 2, 3, or 4?" We can write : **(x == 2) || (x == 3) || (x == 4)**

- The logical operators often provide more than one way to ask a question.
- If x is an integer variable, the preceding question also could be written in either of the following ways:
    1. (x > 1) && (x < 5)
    2. (x >= 2) && (x <= 4)

Precedence of Logical Operators

- Among the logical operators Not (!) has the highest precedence, AND (&&) has next highest and OR(||) has lowest precedence.
- The ! operator has a precedence equal to the unary mathematical operators ++ and --.
- Thus, ! has a higher precedence than all the relational operators and all the binary mathematical operators.
- The && and || operators have much lower precedence, lower than all the mathematical and relational operators, although, && has a higher precedence than ||.
- As with all of C's operators, parentheses can be used to modify the evaluation order when using the logical operators. Consider the following example:
- We want to write a logical expression that makes three individual comparisons:

  1. Is a less than b? **2. Is a less than c?** **3. Is c less than d?**
- **w**e want the entire logical expression to evaluate to true if condition 3 is true and if either condition 1 or condition 2 is true. We can write as **a < b || a < c && c < d**
- **H**owever, this won't do what we intended. Because the && operator has higher precedence than ||, the expression is equivalent to **a < b || (a < c && c < d)** and evaluates to true if (a < b) is true or both a<c and c<d are true.
- So we need to write **(a < b || a < c) && c < d** which forces the || to be evaluated before the &&.

An example to demostrate Logical operator precedence.

```
 #include <stdio.h>

 /* Initialize variables. Note that c is not less than d, */
 /* which is one of the conditions to test for. */
 /* Therefore, the entire expression should evaluate as false.*/

 int a = 5, b = 6, c = 5, d = 1;
 int x;
main()
{
   /* Evaluate the expression without parentheses */
   x = a < b || a < c && c < d;
   printf("\nWithout parentheses the expression evaluates as %d", x);
   /* Evaluate the expression with parentheses */
  x = (a < b || a < c) && c < d;
   printf("\nWith parentheses the expression evaluates as %d\n", x);
   return 0;
}
```

Output

```
Without parentheses the expression evaluates as 1
With parentheses the expression evaluates as 0
```

The Assignment Operator

- The assignment operator is the equal sign (=). Its use in programming is somewhat different from its use in regular math.
- If we write x = y; in a C program, it doesn't mean "x is equal to y." Instead, it means "assign the value of y to x."
- In a C assignment statement, the right side can be any expression, and the left side must be a variable name. Thus, the form is as follows:

    variable = expression;

- When executed, expression is evaluated, and the resulting value is assigned to variable.
- x= x+2; assigns value evaluated by expression x+2 to variable x itself.

Compound Assignment Operators

- C's compound assignment operators provide a shorthand method for combining a binary arithmetic operation with an assignment operation.
- For example, if we want to increase the value of x by 5, or, in other words, add 5 to x and assign the result to x. we can write

```
x = x + 5;
```

- Using a compound assignment operator, which we can think of as a shorthand method of assignment, we can write **x += 5;**
- In more general notation, the compound assignment operators have the following syntax (where op represents a binary operator):

  `exp1 op= exp2` This is equivalent to writing

  `exp1 = exp1 op exp2;`
- **Below table shows the use of all binary arithmetic operators with compound assignment**

When we Write This.	It Is Equivalent To
x *= y	x = x * y
y -= z + 1	y = y - z + 1
a /= b	a = a / b
x += y / 8	x = x + y / 8
y %= 3	y = y % 3

The Conditional Operator

- The conditional operator is C's only ternary operator, meaning that it takes three operands. Its syntax is

  exp1 ? exp2 : exp3;

- If exp1 evaluates to true (that is, nonzero), the entire expression evaluates to the value of exp2. If exp1 evaluates to false (that is, zero), the entire expression evaluates as the value of exp3.
- For example, the following statement assigns the value 1 to x if *exp* is true and assigns 100 to x if *exp* is false:

  x = exp ? 1 : 100;

- Likewise, to make z equal to the larger of x and y, we can write

  z = (x > y) ? x : y;

Conditional operator functions somewhat like an if statement. The preceding statements could also be written like this:

if(exp)x = 1; else x=100; **and** if (x > y) z = x; else z = y;

Special Operator: *The Comma Operator*

- The comma is frequently used in C as a simple punctuation mark, serving to separate variable declarations, function arguments, and so on.
- In certain situations, the comma acts as an operator as we can form an expression by separating two sub-expressions with a comma. The result is as follows:
- Both expressions are evaluated, with the left expression being evaluated first. The entire expression evaluates to the value of the right expression.
- For example, the following statement assigns the value of b to x, then increments a, and then increments b:

  x = (a++ , b++);
- Because the ++ operator is used in postfix mode, the value of b, before it is incremented, is assigned to x. Using parentheses is necessary because the comma operator has low precedence, even lower than the assignment operator.
- The use of comma operator is used in for looping construct as:

  for(n=1,m=10;n<=m;n++,n++)
- exchanging values: **t = x,x = y,y = t;**
- The other types of special operators used in C are pointer operators(&,*), member operators(.and ->),function operator (), array operator []etc.

```
/* An example for Comma operator */
#include<stdio.h>
main()
{
        int x,y,a=12,b=15;
        int t;
        x=(a++,b++);
        printf("a=%d,b=%d,x=%d",a,b,x);
        y=(b++,a++,x++);
        printf("\na=%d,b=%d,x=%d,y=%d",a,b,x,y);
        t = x,x = y,y = t;
        printf("\nx=%d,y=%d",x,y);
        getch();
        return 0;
}
```

Outpur:

```
a=13,b=16,x=15
a=14,b=17,x=16,y=15
x=15,y=16
```

The Cast operator: Explicit type conversion

- C supports another type of operator for data conversion. The cast operator are used to type cast for the data.
- Once a data are defined as one type and need to convert in to another during calculation rather changing the actual value of the variable , we need to cast operators.
- The cast operator is nothing but just the type keyboard of the dada type as int,float,double etc.
- The syntax for type casting is : **(type-name)expression** where type-name is one of the C data type.
  - **For example: int x,y ; float z;**

    z= float(x)/y; or z = x / (float)y;
- The expression may be constant, variable or any expression.

The Cast operator: Explicit type conversion

- Some example of casts and their action are shown in table below

Example	Action
x = (int)7.5	7.5 is converted to integer by truncation
a = (int)21/(int)4.5	Evaluates as 21/4 result is 5
b = (float)sum/n	Division is done in floating point mode
y = (int)(a+b)	The result of a+b is converted in to integer
c=(char)87	returns the character whose ASCII code is 87

Bitwise Operators

- C has special operators for bit oriented programming known as bitwise operators.

- Bitwise operators in C are used for manipulating data in bit level. These are used to testing the bits or shifting the bits left or right.

- Bitwise operators are not applied to float and double.

Bitwise Operators

- Following table shows the bitwise operators and their meanings.

Operator	Meaning
&	bitwise AND
|	bitwise OR
^	bitwise X-OR
<<	shift left
>>	shift right
~	one’s complement

Bitwise Logical operators

- The three operators & , | , ^ act on integral expressions only.
- The two operands of these operators are operated bit by bit.
- The following table shows the bitwise operator acting on 1 bit fields.

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

- **Bitwise AND (&):** The & operator is surrounded by its two integral operands. The result of AND operations is 1 if both the bit is 1 otherwise 0.

Example: For 4 byte word machine:

a = **00000000 00000000 00000000 00001101** (13 DEC)

b= **00000000 00000000 00000000 00011001** (25 DEC.)

- if we execute statement: **c=a&b;**   then

c= **00000000 00000000 00000000 00001001** (DEC 9)

Bitwise OR (|)

- The | operator is also surrounded by its two integral operands. The result of ORing operations is 1 if at least one of the bit of its operand is 1 otherwise 0.
- Example:

a = 00000000 00000000 00000000 000**01101** (13 D)

b= 00000000 00000000 00000000 000**11001** (25 D)

- if we execute statement: **c=a| b;** then

c= 00000000 00000000 00000000 000**11101** (29 D)

Bitwise Exclusive OR (^)

- The ^ operator is also surrounded by its two integral operands. The result of X-OR operations is 1 if only one of the bit of its operand is 1 otherwise 0.
- Example below:

a = 00000000 00000000 00000000 00001101 (13 D)

b= 00000000 00000000 00000000 00011001 (25 D)

- if we execute statement: **c=a^b;** then

c= 00000000 00000000 00000000 00010100 (20 D)

```
/*Program to demo bitwise logical operators*/
#include<stdio.h>
main()
{
        int a=13,b=25,c;
        c=a&b;
        printf("%d&%d=%d",a,b,c);
        c=a|b;
        printf("\n%d|%d=%d",a,b,c);
        c=a^b;
        printf("\n%d^%d=%d",a,b,c);
        return 0;
}
```

Output:

```
13&25=9
13|25=29
13^25=20
```

Bitwise Shift operators

The shift operators are used to shift the bit of any integral expression to the left or right as indicated by the number of bits.

- **The left shift operators(<<) :** The left shift operator takes the form: **expr<< n**
- Which causes the bits representation of expr to be shifted to the left by the number of places specified by **n.**
- The leftmost n bits in the original bit pattern will be lost and rightmost n bit positions that are vacated will be filled with 0s
- e.g. **0000 1111<< 2** produces **0011 1100**
- The decimal value of 0000 1111 is 15 and the decimal value of 0011 1100 is 60.
- So left shift by 1, multiply the number by 2. left shift by 2 , multiply the number by 4 and so on ..

The right shift operator (>>)

- The left shift operator takes the form: **expr1 >> n** which causes the bits representation of expr1 to be shifted to the right by the number of places specified by **n.**
- The rightmost n bits in the original bit pattern will be lost and leftmost n bit positions that are vacated will be filled with 0s

e.g.

0000 1111>> 2 produces **0000 0011**

- The decimal value of **0000 1111** is **15** and the decimal value of **0000 0011** is **3** .

- So right shift by 1 ,divides the number by 2, (integer division) , right shift by 2 bits divides the number by 4.

The one’s complement operator(~)

- The complement operator ~ is unary complementary operator. It inverts the bit string representation of its argument, the 0s becomes 1s and the 1s become 0s.
- Consider the example ,
- a= 9;
- The bit representation is :

a = 00000000 00001001

- **b= ~a;** becomes: 11111111 11110110

END

of

UNIT 4