Operators in C

An operator is defined as a symbol that operates on operands and does something. The something may be mathematical, relational or logical operation. C language supports a lot of operators to be used in expressions. These operators can be categorized into the following groups:

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

Arithmetic operators

- ✓ These are used to perform mathematical operations.
- ✓ These are binary operators since they operate on two operands at a time.
- ✓ They can be applied to any integers, floating-point number or characters.
- ✓ C supports 5 arithmetic operators. They are +, -, *, /, %.
- ✓ The modulo (%) operator can only be applied to integer operands and cannot be used on float or double values.

Consider three variables declared as,

int a=9,b=3,result;

We will use these variables to explain arithmetic operators. The table shows the arithmetic operators, their syntax, and usage in C language.

Operation	Operator	Syntax	Statement	Result
Addition	+	a+b	result=a+b	12
Subtraction	-	a-b	result=a-b	6
Multiply	*	a*b	result=a*b	27
Divide	/	a/b	result=a/b	3
Modulo	%	a%b	result=a%b	0

a and *b* are called operands.

An Example ‘C’ program that illustrates the usage of arithmetic operators.

```
#include<stdio.h>
int main()
{
        int a=9,b=3;
        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);
         return 0;
}
```

Output
9+3=12
9-3=6
9*3=27
9/3=3
9%3=0

Relational operators

A relational operator, also known as a comparison operator, is an operator that compares two operands. The operands can be variables, constants or expressions. Relational operators always return either *true* or *false* depending on whether the conditional relationship between the two operands holds or not.

C has six relational operators. The following table shows these operators along with their meanings

Operator	Meaning	Example
<	Less than	3<5 gives 1
>	Greater than	7<9 gives 0
<=	Less than or equal to	100<=100 gives 1
>=	Greater than or equal to	50>=100 gives 0
==	Equal to	10==9 gives 0
!=	Not equal to	10!=9 gives 1

An example program that illustrates the use of arithmetic operators.

```
#include<stdio.h>
int main()
{
        int a=9,b=3;
        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);
        return 0;
}
```

Output
9 > 3 = 1
9 >= 3 = 1
9 < 3 = 0
9 <= 3 = 0
9= =3 = 0
9 != 3 = 1

Logical operators

Operators which are used to combine two or more relational expressions are known as logical operators. C language supports three logical operators – logical AND(&&), logical OR(||), logical NOT(!).

- ✓ Logical&& and logical || are binary operators whereas *logical!*is an unary operator.
- ✓ All of these operators when applied to expressions yield either integer value 0 (false) or an integer value 1(true).

Logical AND

It is used to simultaneously evaluate two conditions or expressions with relational operators. If expressions on the both sides (left and right side) of logical operators is true then the whole expression is *true* otherwise *false*. The truth table of logical AND operator is given below:

A	B	A&&B
0	0	0
0	1	0
1	0	0
1	1	1

Logical OR

It is used to simultaneously evaluate two conditions or expressions with relational operators. If one or both the expressions on the left side and right side of logical operators is true then the whole expression is *true* otherwise *false*. The truth table of logical OR operator is given below:

A	B	A||B
0	0	0
0	1	1
1	0	1
1	1	1

Logical NOT

It takes single expression and negates the value of expression. The truth table of logical NOT operator is given below

A	!A
0	1
1	0

For example: x and y are two variables. The following equations explain the use of logical operators.

```
z1 = x&&y    …1
z2 = x||y    …2
z3 =!x       …3
```

Equation1 indicates that z1 is true if both x and y is true. Equation2 indicates z2 is true when x or y or both true. Equation3 indicates z3 is true when x is not true.

An example 'C' program that illustrates the use of logical operators

```
#include<stdio.h>
int main()
{
        int z1,z2,z3;
        z1=7>5 && 10>15;
        z2=7>5 || 10>15;
        z3=!(7>5);
        printf(" z1=%d\n z2=%d\n z3=%d \n",z1,z2,z3);
        return 0;
}
```

Output

```
z1=0
 z2=1
 z3=0
```

Unary operators

Unary operators act on single operands. C language supports three unary operators. They are:

1. Unary minus
2. Increment operator
3. Decrement operator

Unary minus

The unary minus operator returns the operand multiplied by –1, effectively changing its sign. When an operand is preceded by a minus sign, the unary minus operator negates its value. For example,

```
int a, b=10;
a=-(b);
```

the result of this expression is a=-10.

Increment (++) and decrement (− −) operators

The increment operator is unary operator that increases value of its operand by 1. Similarly, the decrement operator decreases value of its operand by 1. These operators are unique in that they work only on variables not on constants.These are used in loops like while, do-while and for statements.

There are two ways to use increment or decrement operators in expressions. If you put the operator in front of the operand *(prefix),* it returns the new value of the operand (incremented or decremented). If you put the operator after the operand *(postfix),* it returns the original value of the operand (before the increment or decrement).

Operator	Name	Value returned	Effect on variable
x++	Post-increment	x	Incremented
++x	Pre-increment	x=x+1	Incremented
x− −	Post-decrement	x	Decremented
− −x	Pre-decrement	x=x - 1	Decremented

An example program that illustrates the use of unary operators

```
#include<stdio.h>
int main()
{
        int x=5;
        printf("x=%d\n", x++);
        printf("x=%d\n",++x);
        printf("x=%d\n",x--);
        printf("x=%d\n",--x);
        printf("x=%d\n",-x);
         return 0;
}
```

Output

```
x=5
x=7
x=7
x=5
x=-5
```

Bitwise operators

As the name suggests, the bitwise operators operate on bits. These operations include:

1. bitwise AND(&)
2. bitwise OR(|)
3. bitwise X-OR(^)
4. bitwise NOT(~)
5. shift left(<<)
6. shift right(>>)

Bitwise AND (&)

When we use the bitwise AND operator, the bit in the first operand is ANDed with the corresponding bit in the second operand. If both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example:

In a C program, the & operator is used as follows.

```
int a=4,b=2,c;
c=a&b;
printf("%d",c);          //prints 0
```

```
  1 0 0    //binary equivalent of decimal 4
& 0 1 0    //binary equivalent of decimal 2
-------
  0 0 0
```

Bitwise OR (|):

When we use the bitwise OR operator, the bit in the first operand is ORed with the corresponding bit in the second operand. If one or both bits are 1, the corresponding bit in the result is 1 and 0 otherwise. For example:

In a C program, the | operator is used as follows.

```
int a=4,b=2,c;
c=a|b;
printf("%d",c);          //prints 6
```

```
  1 0 0
| 0 1 0    //binary equivalent of decimal 4
-------    //binary equivalent of decimal 2
  1 1 0
```

Bitwise XOR (^)

When we use the bitwise XOR operator, the bit in the first operand is XORed with the corresponding bit in the second operand. If both bits are different, the corresponding bit in the result is 1 and 0 otherwise. The truth table operator is shown below:

A	B	A&&B
0	0	0
0	1	1
1	0	1
1	1	0

For example:

In a C program, the | operator is used as follows.

```
int a=4,b=2,c;
c=a^b;
printf("%d",c);          //prints 6
```

```
  1 0 0      (binary equivalent of decimal 4)
| 0 1 0      (binary equivalent of decimal 2)
-------
  1 1 0
```

Bitwise NOT (~)

One's complement operator (Bitwise NOT) is used to convert each *1* to *0* and *0* to *1*, in the given binary pattern. It is a unary operator i.e. it takes only one operand.

For example, In a C program, the ~ operator is used as follows.

```
int a=4,c;
c=~a;
```

printf("%d",c); //prints -5

~ 1 0 0 (binary equivalent of decimal 4)
1 0 1

Shift operators

C supports two bitwise shift operators. They are shift-left (<<) and shift-right (>>). These operations are simple and are responsible for shifting bits either to left or to the right. The syntax for shift operation can be given as:

operand operator num

where the bits in *operand* are shifted left or right depending on the *operator* (<<, >>) by the number of places denoted by *num*.

Left shift operator

When we apply left-shift, every bit in *x* is shifted to left by one place. So, the MSB (Most Significant Bit) of x is lost, the LSB (Least Significant Bit) of x is set to 0.

Let us consider int x=4;
Now shifting the bits towards left for 1 time, will give the following result

	MSB			LSB	
	2^3	2^2	2^1	2^0	
	8	4	2	1	
x=4	0	1	0	0	
	1	←	0	←	0
x<<1 =	1	0	0	0	the result of x<<1 is 8

Left-shift is equals to multiplication by 2.

Right shift operator

When we apply right-shift, every bit in *x* is shifted to right by one place. So, the LSB (Least Significant Bit) of x is lost, the MSB (Most Significant Bit) of x is set to 0. Let us consider *int x=4;*
Now shifting the bits towards right for 1 time, will give the following result.

	2^3	2^2	2^1	2^0	
	8	4	2	1	
x=4	0	→	1	→	0	→	0
x>>1 =	0	0	1	0	the result of x>>1 is 2

Right-shift is equals to division by 2.

An example 'C' program that illustrates the use of bitwise operators

```
#include <stdio.h>
int main()
```

```
{
        int a=4,b=2;
        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\n",a,~a);
        printf("%d<<1 = %d\n",a,a<<1);
        printf("%d>>1 = %d\n",a,a>>1);
        return 0;
    }
```

Output:

```
4 | 2 = 6
4 & 2 = 0
4 ^ 2 = 6
~4 = -5
4 << 1 = 8
4 >> 1 = 2
```

Assignment operators

Assignment operator (=)

In C, the assignment operator (=) is responsible for assigning values to variables. When equal sign is encountered in in an expression, the compiler processes the statement on the right side of the sign and assigns the result to variable on the left side. For example,

```
int x;
x=10;
```

Assigns the value 10 to variable x. if we have,

```
int x=2,y=3,sum=0;
sum = x + y;
```

then sum=5.

The assignment has right-to-left associativity, so the expression

int a=b=c=10;

is evaluated as

(a=(b=(c=10)))

Consider the following set of examples:

```
a = 5;        // value of variable 'a' becomes 5
a = 5+10; // value of variable 'a' becomes 15
a = 5 + b; // value of variable 'a' becomes 5 + value of b
a = b + c; // value of variable 'a' becomes value of b + value of c
```

Short hand/Compound assignment operators

The assignment operator can be combined with the major arithmetic operators such operators are called compound assignment operators. C language supports a set of compound assignment operators of the form:

variable op = expression;

where *op* is binary arithmetic operator.

The following table lists the assignment operators supported by the C:

Op	Description	Example
+=	Add AND assignment operator. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
−=	Subtract AND assignment operator. It subtracts the right operand from the left operand and assigns the result to the left operand.	C −= A is equivalent to C = C − A
*=	Multiply AND assignment operator. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator. It takes modulus using two operands and assigns the result to the 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

An example 'C' program that illustrates the use of assignment operators

```
#include <stdio.h>
int main()
{
    int a = 21,c;
    c = a;
    printf("Operator is = and c = %d\n", c );
     c += a;
    printf("Operator is += and c=%d\n",c);
    c −= a;
    printf("Operator is −= and c=%d\n",c);
     c *= a;
    printf("Operator is *= and c=%d\n",c);
    c /= a;
    printf("Operator is /= and c=%d\n", c);
    c  = 200;
    c %= a;
    printf("Operator is %= and c=%d\n",c);
    c <<= 2;
```

```
    printf("Operator is<<=and    c=%d\n",c);
    c >>= 2;
    printf("Operator is >>= and   c=%d\n",c);
   c &=  2;
   printf("Operator is &= and    c = %d\n", c );
   c ^= 2;
   printf("Operator is ^= and   c = %d\n", c );
   c |= 2;
   printf("Operator is |= and   c = %d\n", c );
   return 0;
}
```

Output

```
Operator is = and c = 21
Operator is += and  c = 42
Operator is −= and  c = 21
Operator is *= and c = 441
Operator is /= and c = 21
Operator is %= and  c = 11
Operator is <<= and c = 44
Operator is >>= and c = 11
Operator is &= and c = 2
Operator is ^= and c = 0
Operator is |= and c = 2
```

Advantages:

1. Short hand expressions are easier to write.
2. The statement involving short hand operators are easier to read as they are more concise.
3. The statement involving short hand operators are more efficient and easy to understand.

Conditional operator (? :)

It is also called *ternary* operator because it takes three operands. It has the general form:

variable = expression1 ? expression2 : expression3;

If the *expression1* is evaluated to*true* then it evaluates *expression2* and its value is assigned to variable, otherwise it evaluates *expression3* and its value is assigned to variable.

It is an alternative to simple *if..else* statement

For example

```
#include<stdio.h>
int main()
{
        int a=5,b=6,big;
        big = a>b ? a : b;
```

```
        printf("%d is big\n", big);
        return 0;
}
```

Output

6 is big

sizeof Operator

The operator *sizeof* is a unary operator used calculates the size of data types and variables. The operator returns the size of the variable, data type in bytes. i.e. the *sizeof* operator is used to determine the amount of memory space that the variable/data type will take. The outcome is totally machine-dependent. For example:

```
#include<stdio.h>
int main()
{
        printf("char occupies %d bytes\n",sizeof(char)); printf("int
        occupies %d bytes\n",sizeof(int)); printf("float occupies %d
        bytes\n",sizeof(float)); printf("double occupies %d
        bytes\n",sizeof(double)); printf("long double occupies %d
        bytes\n",sizeof(long double)); return 0;

    }
```

Output

char occupies 1 bytes
int occupies 2 bytes
float occupies 4 bytes
double occupies 8 bytes
long double occupies 10 bytes

Control Statements in C

Statements

- ✓ The statements of a C program control the flow of program execution.
- ✓ A statement is a command given to the computer that instructs the computer to take a specific action, such as display to the screen, or collect input.
- ✓ A computer program consists of a number of statements that are usually executed in sequence. Statements in a program are normally executed one after another.

Control statements enable us to specify the flow of program control; ie, the order in which the instructions in a program must be executed. They make it possible to make decisions, to perform tasks repeatedly or to jump from one section of code to another.

Control statements in C programming language are divided into two types.

1. Selection/ Branching statements (also called Decision making statements) and
2. Iteration/ Looping statements (also called Repetitive statements).

- ✓ Branching is deciding what actions to take and
- ✓ Looping is deciding how many times to take a certain action.

Selection/ Branching statements are again divided into two types.

1. **Conditional statements:**(if,if-else, nested if-else, else-if ladder, switch and conditional expression/conditional operator)
2. **Unconditional statements** (break, continue,goto and exit)

Iteration/ Looping(Repetitive) statements are as follows:There are three types of loops in C programming:

1. while loop
2. do..while loop
3. for loop

Specifying Test Condition for Selection and Iteration

A test condition used for selection and iteration is expressed as a *test expression*. If an expression evaluates to true, it is given the value of 1. If a test expression evaluates to false, it is given the value of 0.

- ✓ Relational and Logical operators are available to specify the test condition used in the control statements of C.
- ✓ Relational operators are used to specify individual test expression.
- ✓ Logical operators are used to connect more than one test expression.

Relational Operators

To Specify	Symbol Used
Less than	<
Less than or equal to	<=
Greater than	>
Greater than or equal to	>=
Equal to	==

Logical Operators

To Specify	Symbol Used
Logical AND	&&
Logical OR	||
Logical NOT	!

Not equal to	!=

- **Writing Test Expression**
- Test condition is specified with test expression. The syntax of the test expression for specifying the test condition is as follows:

(Variable/ Expression/ Constant) operator (Variable/ Expression/ Constant)

Some examples of expressions are given below.

1. (num%2==0)
2. (year%4==0)
3. (a>b)
4. (sum==num)
5. (per>=80)
6. (num!=0)
7. (num>10)
8. (num<=10).

3.1 Selection/ Branching statements

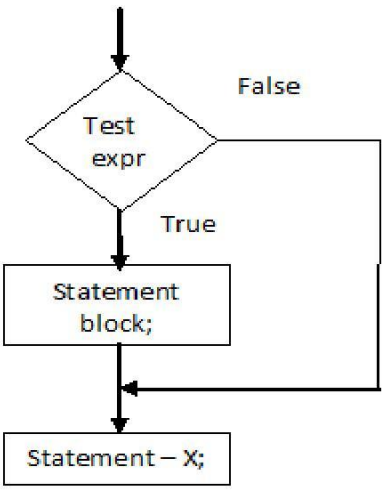
The simple if

This is a powerful decision-making statement and is used to control the flow of execution of statements. It is basically a two-way decision statement and is used in conjunction with an expression. It takes the following form:

Syntax:

```
if(test expression )
{
        statement-block;
}
statement - x;
```

Flowchart



If the *test expression* is true then the statement-block will be executed; otherwise the statement block will be skipped and the execution will jump to the statement-x. Remember, when the condition is true both the statement-block and the statement-x are executed in sequence.

Ex 1: Write a program „C‟ to check whether the given number is +ve or –ve using simple **if** statement.

```
#include<stdio.h>
int main()
{
```

```
        int num;
        printf("Enter a number:");
        scanf("%d", &num);
        if(num>0)

                printf("%d is a positive number\n", num);
        }
        return 0;
}
```

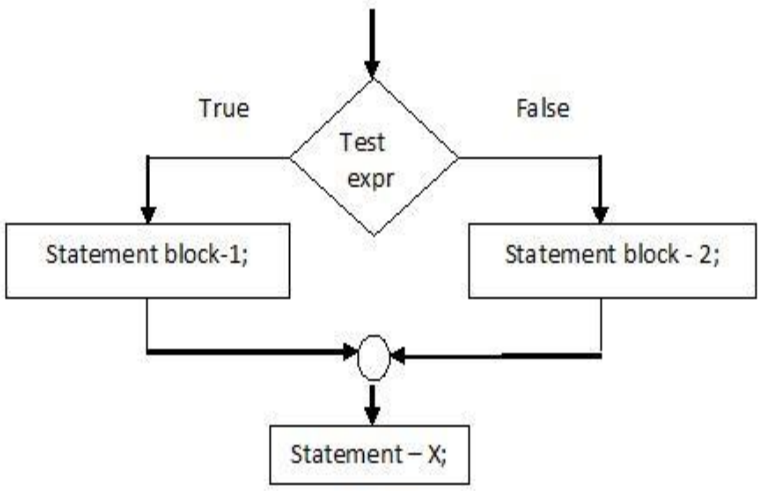
Output:
Enter a number: 9
9 is a positive number.

The if..else statement
The if..else statement is an extension of the simple if statement. The general form is:

Syntax

```
if(test expression )
{
      statement block - 1;
}
else
{
      statement block - 2;
}
statement-x;
```

Flowchart



If the *test expression* is true, then statement block – 1 is executed and statement block-2 is skipped. Otherwise, the statement-2 is executed. In both the cases, the control is transferred subsequently to the statement-x.

Ex 2: Write a program „C" to check whether the given number is even or odd using if..else statement.

```
#include<stdio.h>
int main()
{
   int num;
   printf("Enter a number:");
   scanf("%d",&num);
   if(num%2==0)
      printf("%d is even number\n",num);
   else
      printf("%d is odd number\n",num);
```

```
    return 0;
}
```

Output:
Enter a number: 8
8 is even number.

Ex 3: Write a „C" program to check whether the given year is leap or not using *if..else* statement.

```
.#include<stdio.h>
int main()
{
        int year;
        printf("Enter any year:");
        scanf("%d",&year);
        if(year%4==0)
                printf("%d is leap year\n", year);
        else
                printf("%d is non leap year\n", year);
        return 0;
}
```

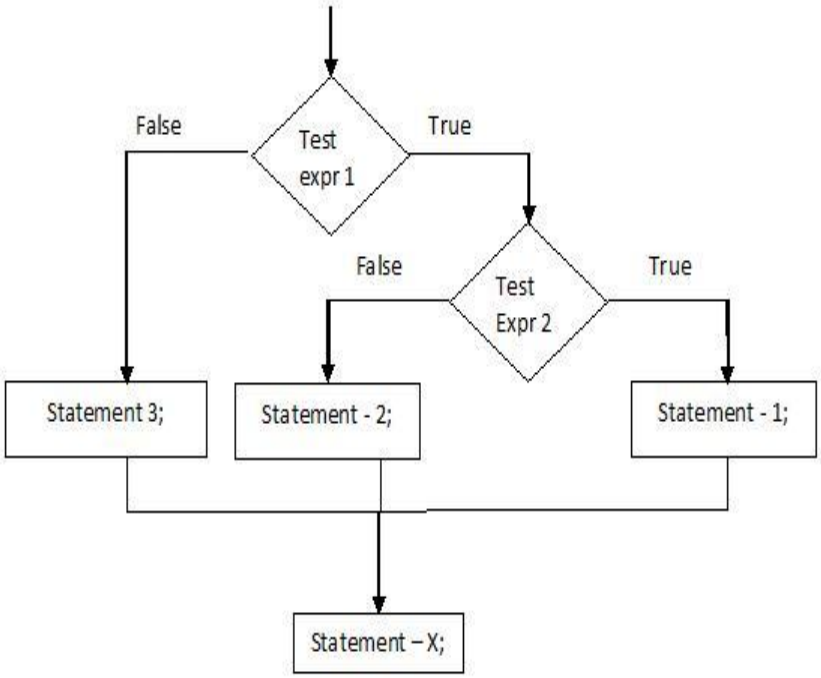
Output:
Enter any year: 1996
1996 is leap year.

Nested if..else statement
When a series of decisions are involved, we may have to use more than one *if..else* statement in nested form. Nested if means one *if* statement with in another *if* statement.

Syntax:

```
if (test expression 1)
{
    if (test expression 2)
    {
       statement-1;
    }
    else
    {
        statement-2;
    }
}
else
{
       statement-3;
}
statement-x;
```

Flowchart:



If the *test expression1* is false, the statement-3 will be executed; otherwise it continues to perform the second test. If the *test expression-2* is true, the *statement-1* will be executed. Otherwise the *statement-2* will be executed and then the control is transferred to the statement-x.

Ex 4: Write a C program to find the biggest among three numbers using nested-if statement.

```
#include<stdio.h>
int main()
{
        int a, b, c;
        printf("Enter a,b,c values:");
        scanf("%d%d%d",&a,&b,&c);
        if(a>b)
        {
                if(a>c)
                        printf("%d is the big\n",a);
                else
                        printf("%d is the big\n",c);
        }
        else
        {
                if(b>c)
                        printf("%d is the big\n",b);
                else
                        printf("%d is the big\n",c);
```

```
    }
    return 0;
}
```

Output:

Enter a, b, c values: 9 5 17
17 is the big

Multi-way Branching statements

In some situations we may like to have more than two possible paths for program flow. In such cases *else..if* or *switch* statement may be used.

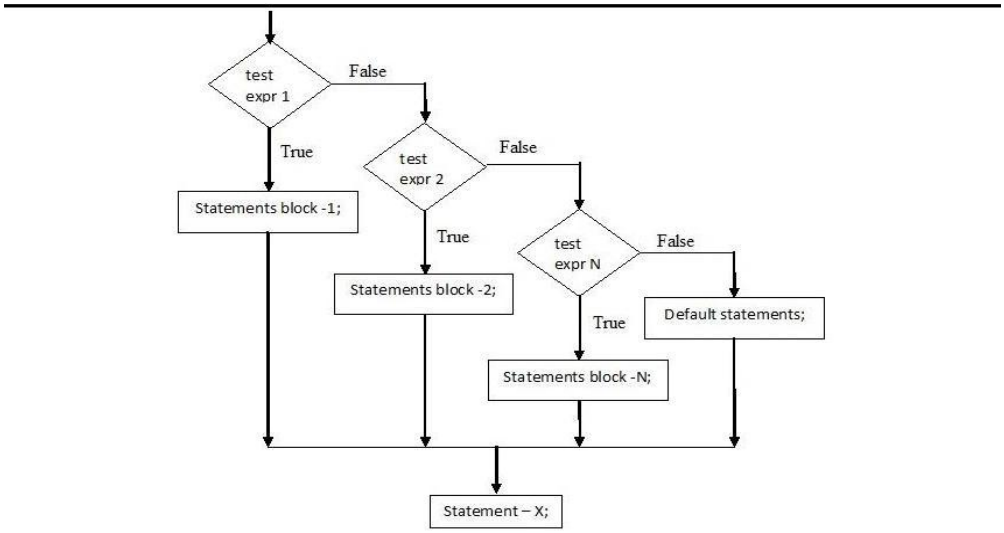
The if..else..if/Ladder if statement

This is the multi-way selection statement (i.e. used to test multiple conditions). The ***syntax*** is:

```
if(test expression 1)
{
        statements block-1;
}
else if (test expression 2)
{
        statements block-2;
 }
      .....
      .....
else if (test expression N)
{
        statement block-N;
}
else
{
      default statements;
}
```

All the conditions are evaluated one by one starting from top to bottom, if any one of the condition evaluating to true then statement group associated with it are executed and skip other conditions. If none of expression is evaluated to true, then the statement or group of statement associated with the final else is executed.

Flowchart



The following program demonstrates a legal if-else if statement: In this program we assign grades to students according to the marks they obtained, using else-if ladder statement.

Ex 5: Write a C program to display student grades using else..if statement.

```
#include<stdio.h>
int main( )
{
        int s1,s2,s3,s4,s5,tot;
        int aggr;
        printf("Enter marks of 5 subjects:");
        scanf("%d%d%d%d%d",&s1,&s2,&s3,&s4,&s5);
        tot=s1+s2+s3+s4+s5;
        printf("Total marks = %d\n",tot);
        aggr=tot/5;
        printf("Aggregate = %d\n",aggr);
        if(aggr>=80)
                printf("Grade: Excellent\n");
        else if (aggr>=75)
                printf("Grade: First class with Distinction");
        else if(aggr>=60)
                printf(" Grade : First Class");
        else if(aggr>=50)
                printf("Grade : Second Class");
        else if(aggr>=40)
                printf("Grade : Pass class");
        else
                printf("Grade: Fail");
        return 0;
}
```

Output

Enter marks of 5 subjects: 90 90 88 86 84
Total marks = 438
Aggregate = 87
Grade: Excellent

The *switch* statement

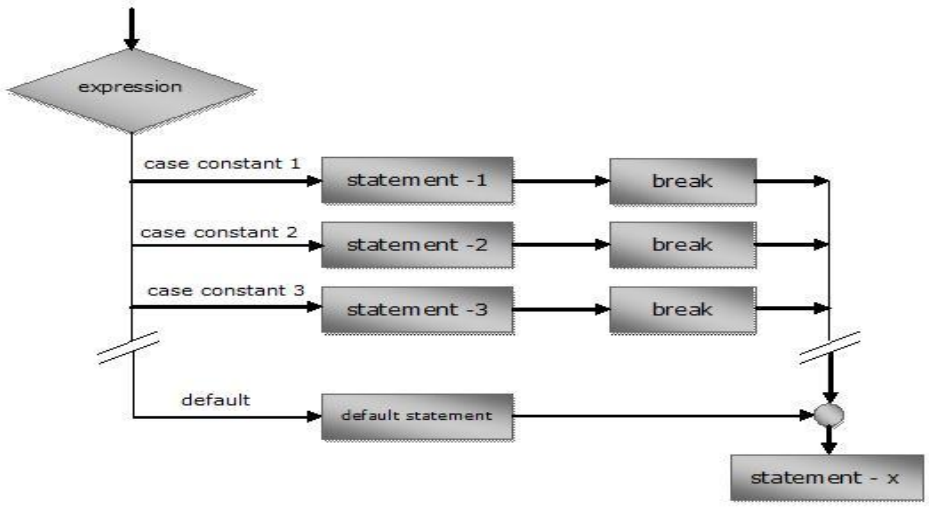
A switch statement is also a multi-way decision making statement that is a simplified version of *if..else..if* block that evaluates a single variable. The ***syntax*** of *switch* statement is:

```
switch(variable)
{
    case constant 1:
        statements;
        break;
    case constant 2:
        statements;
        break;
    case constant N:
        statements;
        break;
    default:
          default statement;
}
```

When switch statement is executed, A variable successively matched against a list of integer or character constants. When a match is found, a statement or block of statements is executed where the default is executed if no matches are found. The default is optional and, if not present, no action takes place if all matches fail.

When a match is found, the statements associated with that case is executed until the break is encountered.

A switch statement work with only *char* and *int* primitive data types, it also works with enumerated types and string.



Ex 6: Write a C program to implement simple calculator using switch statement.

```
#include<stdio.h>
int main()
{
   int choice;
   int a,b;
   printf("Simple Calculator\n");
   printf("---------------------\n");
   printf("1.Addition\n2.Subtraction\n3.Multiplication\n4.Division\n");
   printf("Enter your choice(1..4):");
   scanf("%d",&choice);
   printf("Enter a,b values:");
   scanf("%d%d",&a,&b);
   switch(choice)
   {
      case 1:
         printf("%d+%d=%d\n",a,b,a+b);
         break;
      case 2:
         printf("%d-%d=%d\n",a,b,a-b);
         break;
      case 3:
         printf("%d*%d=%d\n",a,b,a*b);
         break;
      case 4:
         printf("%d/%d=%d\n",a,b,a/b);
         break;
      default:
         printf("Invalid choice....!");
   }
   return 0;
}
```

Output

```
Simple Calculator
---------------------
1. Addition
2. Subtraction
3. Multiplication
4. Division
Enter your choice(1..4): 2
Enter a,b values: 12 7
12 – 7 = 5
```

3.2 Looping Structures

The programmer sometimes interested to repeat a set statements a number of times. This is known as looping in a computer program. A loop is a group of instructions that computer executes repeatedly while some loop continuation conditions remains true. Two ways of repetitions are possible:

1. Counter controlled loops
2. Sentinel controlled loops

(In computer programming, data values used to signal either the start or end of a data series are called sentinels)

Counter controlled loops are sometimes called „definite loops" because we know in advance exactly how many times the loop will be executed.

Sentinel controlled loops are sometimes called „indefinite loops" because it is not known in advance how many times the loop will be executed.

Repetition structure has four required elements:

1. Repetition statement (while, for, do..while)
2. Condition to be evaluated
3. Initial value for the condition
4. Loop termination

C supports the following types of loops:

1. The while loop
2. The do..while loop
3. The for loop

Advantages

- ✓ Reduce length of Code
- ✓ Take less memory space.
- ✓ Burden on the developer is reducing.
- ✓ Time consuming process to execute the program is reduced.

The While Loop

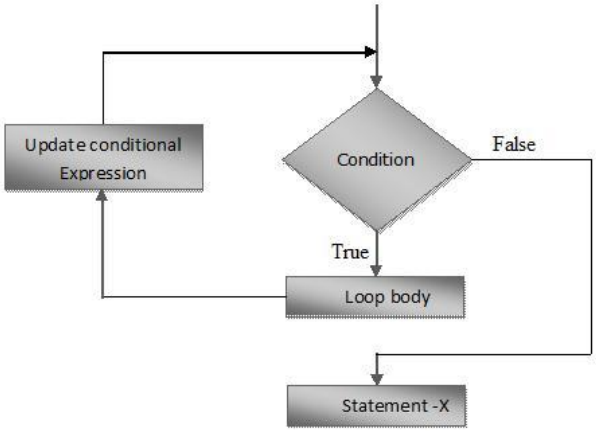
The *while* loop is a pre-test or entry-controlled loop. It uses test expression to control the loop. The while loop evaluates (checking) the test expression before every iteration of the loop, so it can execute zero times if the condition is initially false.

The body of the loop will be executed repeatedly till the value of *test expression* becomes false (0). The initialization of a loop control variable is generally done before the loop separately.

syntax:

```
while (test expression)
{
        body of the loop;
}
statement-x;
```

Flowchart



How while loop works?

- ✓ Initially the while loop evaluates (checking condition) the test expression.
- ✓ If the test expression is true (nonzero i.e.,1), statements inside the body of while loop is evaluated. Then, again the test expression is evaluated. The process goes on until the test expression is false.
- ✓ When the test expression is false, the while loop is terminated.

Ex 7: Write a C program to print 1 to 10 numbers using while loop.

```
#include<stdio.h>
int main( )
{
        int n=1;
        while(n<=10)
        {
                printf("%d\t",n);
                n++;
        }
        return 0;
}
```

Output: 1 2 3 4 5 6 7 8 9 10

Ex 8: Write a C program to print reverse of given number using while loop.

```
#include<stdio.h>
int main( )
{
   int n,rem;
   printf("Enter a number to reversed:");
   scanf("%d",&n);
   printf("The reverse number is ");
   while(n != 0)
   {
        rem = n%10;
```

```
        printf("%d",rem);
        n = n/10;
    }
    printf("\n");
    return 0;
}
```

Output

Enter a number to be reversed: 1234
The reverse number is: 4321

Ex 9: Write a C program to check whether the given number is Armstrong or not.

```
#include<stdio.h>
int main( )
{
    int n,rem,sum=0,temp;
    printf("Enter a number:");
    scanf("%d",&n);
    temp=n;
    while(n != 0)
    {
        rem = n%10;
        sum = sum+(rem*rem*rem);
        n = n/10;
    }
    if(sum == temp)
        printf("%d is Armstrong\n", temp);
    else
        printf("%d is not an Armstrong\n", temp);
    return 0;
}
```

Output

Enter a number: 153
153 is Armstrong

Ex 10: Write a C program for sum of individual digits of a number.

```
#include<stdio.h>
int main()
{
    int n,rem,sum=0;
    printf("enter a number:\n");
    scanf("%d", &n);
    while(n != 0)
    {
        rem=n%10;
```

```
        sum=sum+rem;
        n=n/10;
    }
    printf("The sum of individual digits is %d", sum);
    return 0;
}
```

Output:

Enter a number: 1234
The sum of individual digits is 10

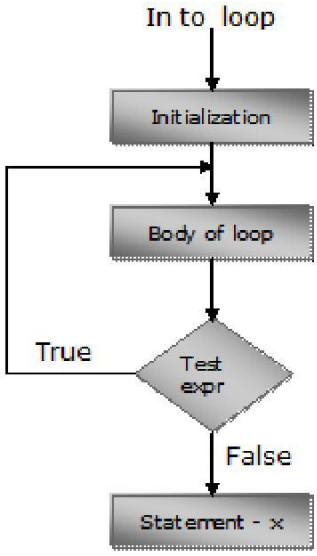
do..while Statement

do-while loop is similar to *while* loop, however there is one basic difference between them. *do-while* runs at least once even if the test condition is false at first time. Syntax of do while loop is:

Syntax

```
do
{
     body of the loop;
 }
 while (expression);
statement -x;
```

Flowchart



How do-while loop works?

- ✓ First the code block (loop body) inside the braces ({….}) is executed once.
- ✓ Then, the test expression is evaluated (checking condition). If the test expression is true, the loop body is executed again. This process goes on until the test expression is evaluated to false (0).
- ✓ When the test expression is false, the **do...while** loop is terminated.

The following example illustrates the use of do-while loop.

```
#include<stdio.h>
int main()
{
        int num=1;
        do
        {
```

```
            printf( " %d\t", num);
            num++;
        }while(num<=5);
        return 0;
}
```

Output: 1 2 3 4 5

Comparison between do and while loops

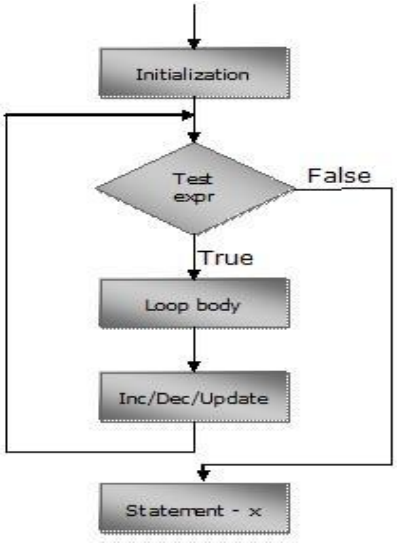
S.No	While loop	Do-while loop
1	Condition is checked before entering into loop	Condition is checked after executing statements in loop
2	Top-tested /entry-controlled loop	Bottom-tested /exit-controlled loop
3	Minimum iterations are zero	Minimum iterations are one
4	Maximum iterations are N+1	Maximum iterations are N
5	General loop statement	General loop statement but well suited for menu-driven applications
6	Non-deterministic loop	Non-deterministic loop

The For Statement

It most general looping construct in C. The *for* loop is commonly used when the number of iterations are exactly known. The ***syntax*** of a *for* loop is

```
for(initialization; test expression; increment /decrement/ update)
{
        body of the loop;
}
```

Flowchart



The loop header contains three parts:

- an initialization,
- a test condition, and
- incrementation(++) / decrementation(- -) /update.

Initialization: This part is executed only once when we are entering into the loop first time. This part allows us to declare and initialize any loop control variables.

Condition: if it is true, the body of the loop is executed otherwise program control goes outside the *for* loop.

Increment or Decrement: After completion of initialization and condition steps loop body code is executed and then increment or decrements steps is execute. This statement allows to us to update any loop control variables.

How for loop works?

- ✓ First the loop initialization statement is executed.
- ✓ Then, the test expression is evaluated. If the test expression is false, for loop is terminated. But if the test expression is true, codes inside the body of for loop is executed and the update expression is executed. This process repeats until the test expression becomes false.

Note: In for loop everything is optional but mandatory to place two semicolons **(; ;)**

Ex : Write a C program to check whether the given number is prime or not.

```
#include<stdio.h>
int main( )
{
   int n,i,fact=0;
   printf("Enter a number:");
   scanf("%d",&n);
   for (i=2;i<n/2;i++)
   {
        if(n%i == 0)
                fact++;
   }
   if(fact == 0)
        printf("%d is Prime\n",n);
   else
        printf("%d is not a Prime\n",n);
   return 0;
}
```

Output

```
Enter a number: 13
13 is  Prime.
```

Ex : Write a C program to print the factorial of a given number.

```
#include<stdio.h>
int main( )
{
   int n,i;
   long int fact=1;
   printf("Enter a number:");
   scanf("%d",&n);
   for (i=1;i<=n;i++)
   {
         fact  = fact*i;
   }
   printf("Factorial of %d is %ld\n", n, fact);
   return 0;
}
```

Output

Enter a number: 5
Factorial of 5 is 120

Comparison between for and do, while loops

S.No	For loop	Do-while loop
1	Generally deterministic in nature	Non-deterministic in nature
2	Loop index can be initialized and altered with in loop	Index will be initialized outside the loop
3	Very flexible in nature	Not so flexible when compared with for loop
4	Condition is checked at beginning of the Loop	Same in case of while loop. In do loop, condition is checked after end of the loop
5	Minimum iterations are zero	Minimum iterations are zero (while) Minimum iterations are one (do-while)

Pre-test loop: The condition can be tested- At the beginning is called as pre-test or entry-controlled loop. Example: while and for loops.

Post-test loop: The condition can be tested - At the end is called as post-test or exit-controlled loop. Example: do-while loop.

Nested loops

Insertion of one loop statement inside another loop statement is called nested loop. Generally *for* loops are nested. In C loops can be nested to any desired level. General syntax for nested *for* loops are:

```
for (initialization; test expression; updation)          //outer loop
{
        for (initialization; test expression; updation) //inner loop {
                        body of inner for loop;
        }
}
```

Note: generally nested for loops are used to generate patterns.

Here is an example program that uses nested loops

```
#include<stdio.h>
int main()
{
        int  n, i, j;       //i and j represents rows and columns respectively


        printf("Enter number of rows:" );
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=i;j++)
                {
                        printf("*  ");
                }
                printf("\n");
        }
        return 0;
}
```

Output:

```
Enter number of rows: 5
 *
 *      *
 *      *      *
 *      *      *      *
 *      *      *      *      *
```

Ex 13: Write a C program to print the following pattern

```
#include<stdio.h>
int main()
{
        int n, i, j; //i and j represents rows and columns respectively
        printf("Enter number of rows:" );
        scanf("%d",&n);
```

```
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=i;j++)
                {
                        printf("%d   ",j);
                }
                printf("\n");
        }
        return 0;
}
```

Output

```
Enter number of rows: 5
1
1       2
1       2       3
1       2       3       4
1       2       3       4       5
```

Ex 14: Write a C program to print the Floyd‟s triangle.

```
#include<stdio.h>
int main()
{
        int  n, i, j,k=1; //i and j represents rows and columns respectively
        printf("Enter number of rows:" );
        scanf("%d",&n);
        for(i=1;i<=n;i++)
        {
                for(j=1;j<=i;j++)
                {
                        printf("%d   ",k++);
                }
                printf("\n");
        }
        return 0;
}
```

Output

```
Enter number of rows: 5
1
2       3
4       5       6
7       8       9       10
11      12      13      14      15
```

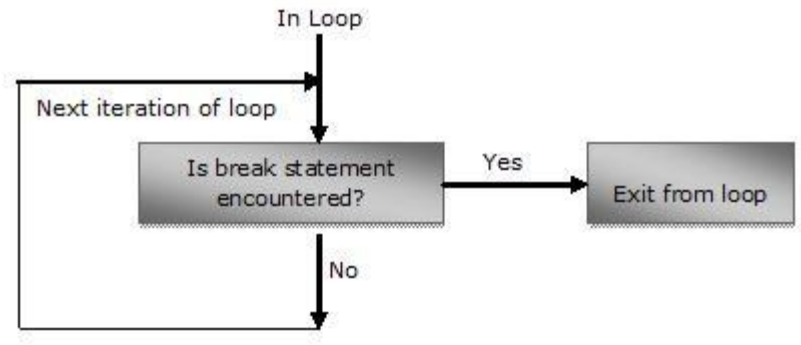
3.3. Unconditional branching statements

The break statement

In C, when *break* statement is encountered inside a loop, the loop is immediately terminated, and program control is transferred to nest statement following the loop. The *break* statement is widely used with *for* loop, *while* loop, *do-while* loop and *switch* statement. Its syntax is quite simple, just type keyword *break* followed with a semicolon.

break;

Following figure shows the sequences in break statements



The following example illustrates the use of break;

```
#include<stdio.h>
int main()
{
    int i=1;
    while( i <= 5)
    {
            if (i==3)
                    break;
            printf("%d   ", i);
            i = i + 1;
    }
    return 0;
}
```

Output

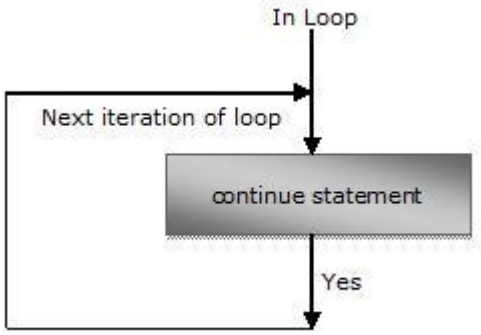
1 2

The continue statement

In C, when *continue* statement is encountered inside a loop, the loop is immediately terminated, and program control is transferred to nest statement following the loop. The *break* statement is widely used with *for* loop, *while* loop, *do-while* loop and *switch* statement. Its syntax is quite simple, just type keyword *continue* followed with a semicolon.

continue;

Following figure shows the sequence of actions in continue statement



The following example illustrates the use of continue;

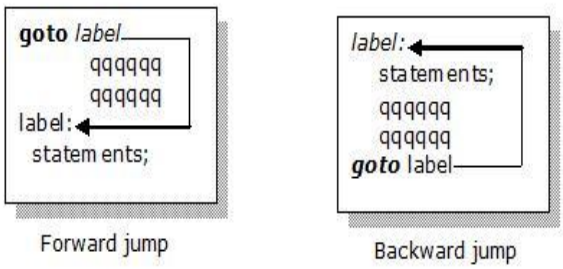
```
#include<stdio.h>
int main()
{
    int i=0;
    while(i<5)
    {
            i=i+1;
            if (i==3)
               continue;
            printf("%d ",i);
    }
    return 0;
}
```

Output

1 2 4 5

The goto statement

The *goto* statement is used to transfers control to a specified *label*. Here *label* is an identifier that specifies the place where the branch is to be made. *Label* can be any valid variable name that is followed by a colon (:). The *label* is placed immediately before the statement where the control has to be transferred. The ***syntax*** of goto statement is:



The following example illustrates the use of goto;

```
#include<stdio.h>
int main( )
{
   int n,sum=0;
   read:
   printf("\nEnter a number (999 for exit):");
   scanf("%d",&n);
   if(n != 999)
   {
if(n<0)
goto read; //jump to label - read
sum += n;
goto read; //jump to label – read
   }
   printf("Sum of the numbers entered by the user is %d\n",sum);
   return 0;
}
```

Output

```
Enter a number (999 for exit): 1
Enter a number (999 for exit): 3
Enter a number (999 for exit): -1
Enter a number (999 for exit): 7
Enter a number (999 for exit): 999
Sum of the numbers entered by the user is 11
```