UNIT -III: Control Flow-Relational Expressions - Logical Operators:
Selection: if-else Statement, nested if, examples, Multi-way selection: switch, else-if, examples. **Repetition:** Basic Loop Structures, Pretest and Posttest Loops, Counter-Controlled and Condition-Controlled Loops, The while Statement, The for Statement, Nested Loops, The do-while Statement.

In computer science, **control flow** (or alternatively, **flow of control**) is the order in which individual statements, instructions or function calls of a program are executed. A **control flow statement** is a statement whose execution results in a choice being made as to which of two or more paths should be followed.

Definitions:

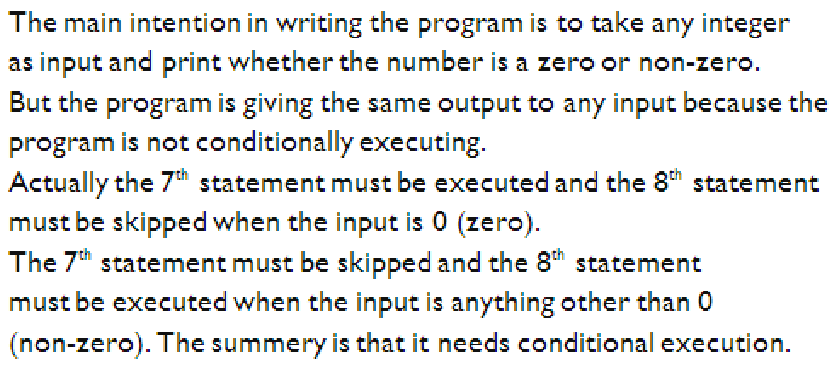
- ✓ A control statement is any statement that alters the linear flow of control of a program. C examples include: if-else, while, for, break, continue and return statements.
- ✓ A control structure is a control statement along with the other statements that it controls

<u>Need of Conditional Execution:</u>

All the programs written so far are sequential programs; execute statement after statement in written order from the beginning of main() to the end of main(). But we can't solve every problem just by writing a sequence of statements. While developing complicated programs, we need to conditionally execute the program. Sometimes a statement or multiple statements need to be executed, in other time the same thing has to skip from the execution. Sometimes we need to execute a statement or a block of statements repeatedly until a condition is attained. In some other case, need to jump from one part of program to other part.

Example:

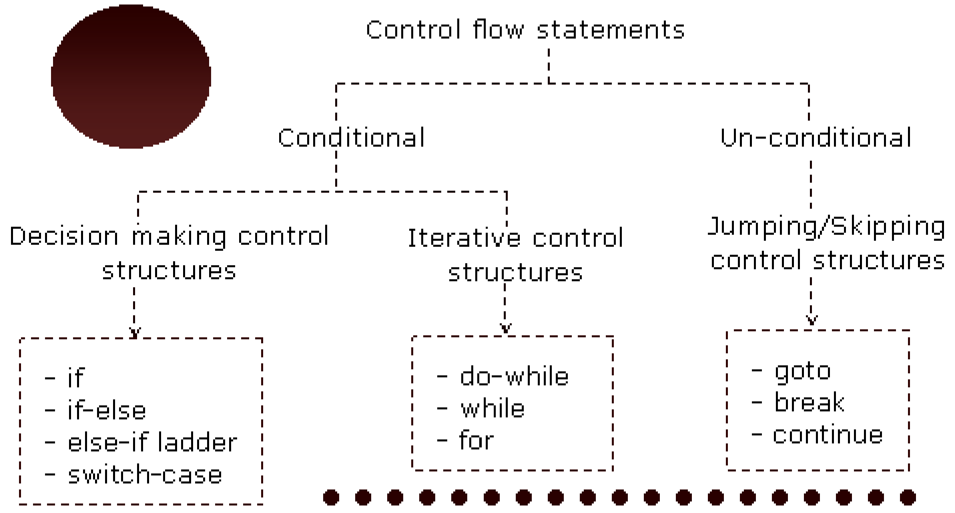
```
#include<stdio.h>
int main()
{
int n;
printf("Enter any integer:");
scanf("%d",&n);
printf("Zero");
printf("\nNon-zero");
return 0;
}
```


The main intention in writing the program is to take any integer as input and print whether the number is a zero or non-zero. But the program is giving the same output to any input because the program is not conditionally executing. Actually the 7th statement must be executed and the 8th statement must be skipped when the input is 0 (zero). The 7th statement must be skipped and the 8th statement must be executed when the input is anything other than 0 (non-zero). The summery is that it needs conditional execution.

Control flow instructions/statements in C language:

Fortunately, C language provides number of control flow instructions/statements to control the flow of program execution conditionally. These are classified as

- Decision making control structures (Selection statements)
- Iteration or Repetitive control structures (Iteration statements)
- Jump statements



How to write a condition:

Both the Decision and iterative control structures control the program execution according to the condition we supply to the conditional statement. Here a condition is either a relational or logical expression that returns either true (1) or false (0). We must first learn how to write a condition before stepping into control flow statements.

✓ **Relational Operators in C language:**

These operators express the relation among any two operands as either true or false. The following are different relational operators provided by the C language.

Operator	Meaning
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not Equal to

Equal to (==) operator in C language:

In C language == is different from =.

= is an assigning operator helps to assign a value or the result of an expression to the variable.

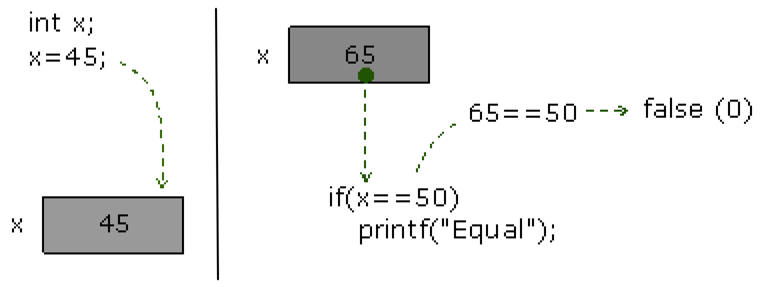
In other hand == is an equal to operator used to compare any two values.

```
int x;
x=45;
```

Here the value 45 is assigning to the variable x.

```
int x=65;
if(x==50)
printf("Equal");
```

Here the value of x 65 is comparing with 50.



```
int x;
x=45;
```

```
if(x==50)
  printf("Equal");
```

Relational Expression:

Relational expression is a combination of constants, variable and relational operators. The result of any relational expression is either 1 (true) or 0 (false). Associativity and precedence of relational operators comparatively with arithmetic operators is as follows.

Operator	**Associativity**
( )	left to right
* / %	left to right
+ –	left to right
< > <= >=	left to right
== !=	left to right
=	right to left

We will see the behavior of relational operators through different example programs.

Example 1:

```
#include<stdio.h>
int main()
{
int a,b,c,d;
a=40<20;/*assigns 0 (false) to a*/
b=40!=20;/*assigns 1 (true) to b*/
c=a!=b;/*assigns 1 (true) to c */
d=b==c;/*assigns 1 (true) to d*/
printf("%d\n%d\n%d\n%d",a,b,c,d);
return 0;
}
```

Output:

```
0
1
1
1
```

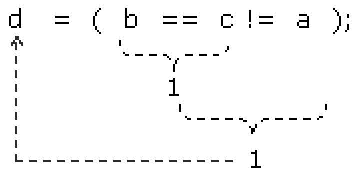
Example 2:

```
#include<stdio.h>
int main()
{
int a=45,b=30,c=30,d;
d=(b==c!=a);
printf("%d",d);
return 0;
}
```

Output:

```
1
```

As the associativity of relational operators is left to right, b==c results 1, 1!=a results 1 and that will be assigned to d.



✓ **Logical operators:**

Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then the condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then the condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.	!(A && B) is true.

Example:

Try the following example to understand all the logical operators available in C −

```
#include <stdio.h>
void main()
{
   int a = 5;
   int b = 20;
   int c ;
     if ( a && b ) {
           printf("Line 1 - Condition is true\n" );
        }
     if ( a || b )    {
          printf("Line 2 - Condition is true\n" );
        }
   /* lets change the value of  a and b */
   a = 0;
   b = 10;
   if ( a && b ) {
      printf("Line 3 - Condition is true\n" );
   }
   else {
      printf("Line 3 - Condition is not true\n" );
   }
   if ( !(a && b) ) {
      printf("Line 4 - Condition is true\n" );
   }   }
```

When you compile and execute the above program, it produces the following result −

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 3 - Condition is not true
Line 4 - Condition is true
```

✓ Selection Statements in C:

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 is made up of a series of statements.

Selection structure is extensively used in programming because it allows the program to decide an action based upon user's input or other processes. The application of selection structure can be seen at systems that have password checking such as ATM Machine. The system allows user access only if the user input password is the same as the corresponding password stored in the system.

Before we go further into the selection structure, you need to know the meaning of True and False in programming. An expression that evaluates to non-zero (1) is considered true expression while false expression evaluates to zero (0)

Decision making is used to specify the order in which statements are executed. In C Programming language we use selection statements for implementing decision making in our program to control its flow of execution.

C programming language provides the following types of decision making statements.

S.No	Statement & Description
1	if statement :An if statement consists of a boolean expression followed by one or more statements.
2	if...else statement : An if statement can be followed by an optional else statement, which executes when the Boolean expression is false.
3	nested if statements :You can use one if or else if statement inside another ifor else if statement(s).
4	switch statement :A switch statement allows a variable to be tested for equality against a list of values.
5	nested switch statements :You can use one switch statement inside another switch statement(s).

Types of selection statements in c are

- ✓ 1-Way selection (if statement)
- ✓ 2-Way selection (if-else statement)
- ✓ Multiple selection (Nested If-else or n-way selection, switch/case)

<u>If Statement</u> (one way Selection):

An **if** statement consists of a Boolean expression followed by one or more statements.

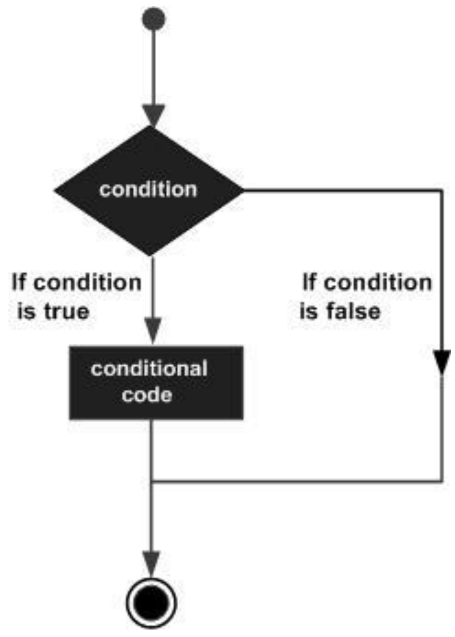
Syntax:

```
if(boolean_expression)
{
   /* statement(s) will execute if the boolean expression is true */
}
```

If the Boolean expression evaluates to **true**, then the block of code inside the 'if' statement will be executed. If the Boolean expression evaluates to **false**, then the first set of code after the end of the 'if' statement (after the closing curly brace) will be executed.

Example:

```
#include <stdio.h>
 int main ( )
{
   /* local variable definition */
   int a = 10;
    /* check the boolean condition using if statement */
   if( a < 20 ) {
      /* if condition is true then print the following */
      printf("a is less than 20\n" );
   }
      printf("value of a is : %d\n", a);
    return 0;
}
```



When the above code is compiled and executed, it produces the following result −

```
a is less than 20;
value of a is : 10
```

<u>if...else statement:</u>

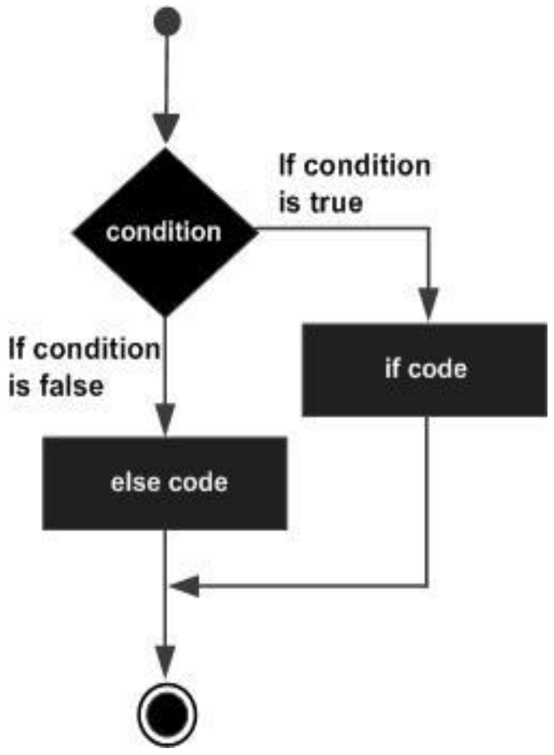
n **if** statement can be followed by an optional **else** statement, which executes when the Boolean expression is false.

Syntax:

```
if(boolean_expression) {
   /* statement(s) will execute if the boolean expression is true */
}
```

```
else {
   /* statement(s) will execute if the boolean expression is false */
}
```

If the Boolean expression evaluates to **true**, then the **if block** will be executed, otherwise, the **else block** will be executed. C programming language assumes any **non-zero** and **non-null** values as **true**, and if it is either **zero** or **null**, then it is assumed as **false** value.



Example:

```
#include <stdio.h>
 int main ( )
{
   /* local variable definition */
   int a = 100;
    /* check the boolean condition */
   if( a < 20 ) {
      /* if condition is true then print the following */
      printf("a is less than 20\n" );
   }
   else {
      /* if condition is false then print the following */
      printf("a is not less than 20\n" );
   }
      printf("value of a is : %d\n", a);
    return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
a is not less than 20;
value of a is : 100
```

<u>If...else if...else Statement:</u>

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if...else if..else statements, there are few points to keep in mind −

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

Syntax:

The syntax of an **if...else if...else** statement in C programming language is −

```
if(boolean_expression 1) {
   /* Executes when the boolean expression 1 is true */
}
else if( boolean_expression 2) {
   /* Executes when the boolean expression 2 is true */
}
else if( boolean_expression 3) {
   /* Executes when the boolean expression 3 is true */
}
else  {
   /* executes when the none of the above condition is true */
}
```

Example

```
#include <stdio.h>
 int main () {
   /* local variable definition */
   int a = 100;
   /* check the boolean condition */
   if( a == 10 ) {
      /* if condition is true then print the following */
      printf("Value of a is 10\n" );
   }   else if( a == 20 ) {
      /* if else if condition is true */
      printf("Value of a is 20\n" );
   }    else if( a == 30 ) {
      /* if else if condition is true  */
      printf("Value of a is 30\n" );
   }
   else {
      /* if none of the conditions is true */
      printf("None of the values is matching\n" );
   }
   printf("Exact value of a is: %d\n", a );
   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
None of the values is matching
Exact value of a is: 100
```

✓ Switch Statement:

Why we should use Switch Case?

1. One of the classic problem encountered in nested if-else /else-if ladder is called Confusion.
2. It occurs when no matching else is available for if
3. As the number of alternatives increases the Complexity of program increases drastically.
4. To overcome this, C Provide a multi-way decision statement called 'Switch Statement'

Switch Statement:

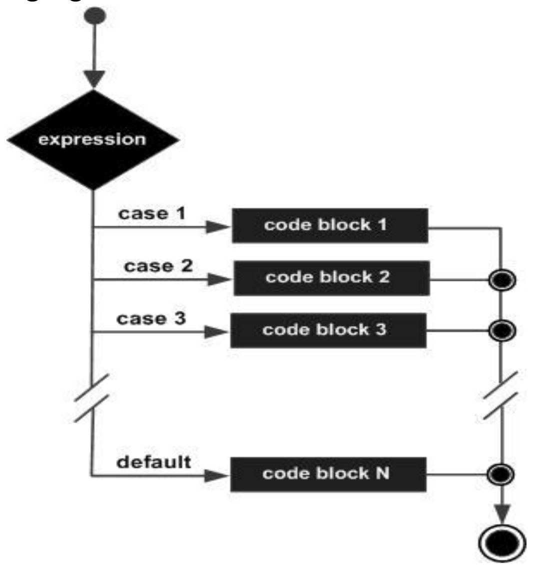
A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each **switch case**.

Syntax: The syntax for a **switch** statement in C programming language is as follows

```
switch(expression)
{
   case constant-expression  :
      statement(s);
      break; /* optional */

   case constant-expression  :
      statement(s);
      break; /* optional */

   /* you can have any number of case statements */
   default : /* Optional */
   statement(s);
}
```



The following rules apply to a **switch** statement −

- The **expression** used in a **switch** statement must have an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a **break** statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a **break**. If no **break** appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No **break** is needed in the default case.

Example

```
#include <stdio.h>
void main () {
  /* local variable definition */
  char grade = 'B';
  switch(grade) {
    case 'A' :        printf("Excellent!\n" );
                      break;
    case 'B' :
    case 'C' :        printf("Well done\n" );
                      break;
    case 'D' :        printf("You passed\n" );
                      break;
    case 'F' :        printf("Better try again\n" );
                      break;
    default :         printf("Invalid grade\n" );
  }
    printf("Your grade is  %c\n", grade );

}
```

When the above code is compiled and executed, it produces the following result −

```
Well done
Your grade is B
```

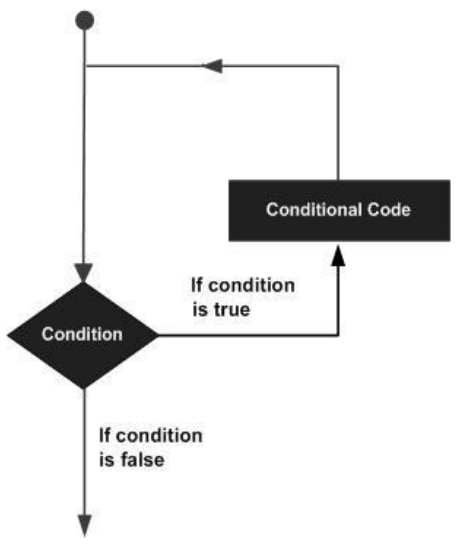
C – Loops:

In Some situations, when a block of code needs to be executed several number of times. In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. Given below is the general form of a loop statement in most of the programming languages −

C programming language provides the following types of loops to handle looping requirements.

S.No	Loop Type & Description
1	**while loop** : Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
2	**for loop** : Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
3	**do...while loop** It is more like a while statement, except that it tests the condition at the end of the loop body.
4	**nested loops**: You can use one or more loops inside any other while, for, or do..while loop.



✓ **Types of Loops:**

We have two types of loops in C. They are
Pre test loop: A loop which checks the condition at the beginning of its execution. Ex:While,For Loops.
Post test loop: A loop which checks the condition after the execution: Ex; do-while

Note:

For Every Loop we have three statements. If we understand those in a better way then we can use the loop in an efficient manner. They are

1. Initialization statement (Starting Point) Ex: a=0,i=10,a=7,i=c (Use Assignment operator only)
2. Test Expression (Ending Point) Ex: a<8,b>j, i!=10 (Use Relational operators only)
3. Update Statement (processing point) Ex: i++, j--,i+2, j+1 (Use Increment/ Decrement only)

Basic Structure of these statements in the Loops are

In for Loop

```
For(Initialization Statement; Test Expression; Update Statement)

{

  Loop Body /* statements in Loop*/

}
```

In while Loop

```
Initialization Statement;
While(Test Expression)
{
   Statements;
   Update statement;
}
```

In do-while Loop

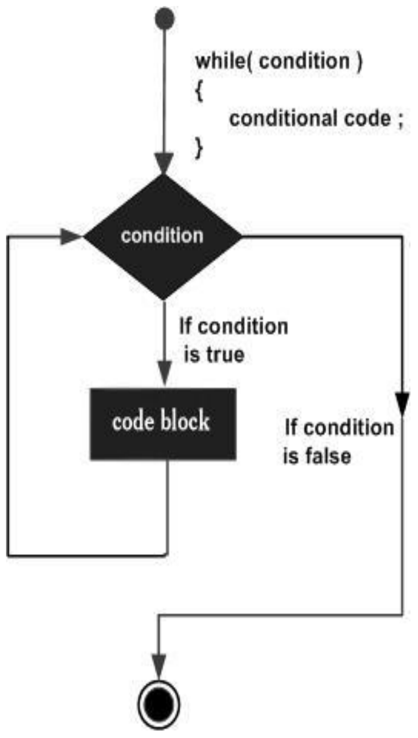
```
Initialization Statement;
do
  {
    Statements;
    Update Statement;
  }
While(Test Expression)
```

- ✓ **Pre- test Loops:**

while loop : A while loop in C programming repeatedly executes a target statement as long as a given condition is true.

Syntax:

```
while (condition)
{
  statement(s);
}
```



Here, **statement(s)** may be a single statement or a block of statements.
The **condition** may be any expression, and true is any nonzero value.
The loop iterates while the condition is true.
When the condition becomes false, the program control passes to the line immediately following the loop.

Here, the key point to note is that a while loop might not execute at all. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
#include <stdio.h>
void main ()
{
  /* local variable definition */
  int a = 10;
  /* while loop execution */
  while( a < 15 )
   {
     printf("value of a: %d\n", a);
      a++;
    }
 }
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

<u>For Loop</u>

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

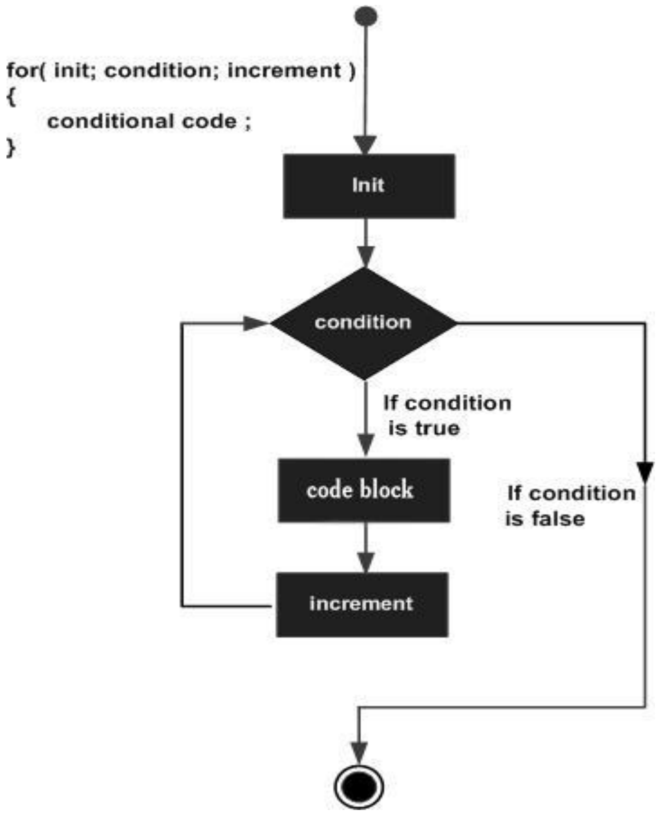
```
for ( init; condition; increment )
{
  statement(s);
}
```

Here is the flow of control in a 'for' loop −

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and the flow of control jumps to the next statement just after the 'for' loop.
- After the body of the 'for' loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the 'for' loop terminates.

Example:

```
#include <stdio.h>
 int main ()
 {
   int   a;
   /* for loop execution */
   for( a = 10; a < 15; a = a + 1 )
    {
       printf("value of a: %d\n", a);
     }
 }
```



Output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

✓ **Post Test Loop:**

Do-while Loop:

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop in C programming checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except the fact that it is guaranteed to execute at least one time.

Syntax:

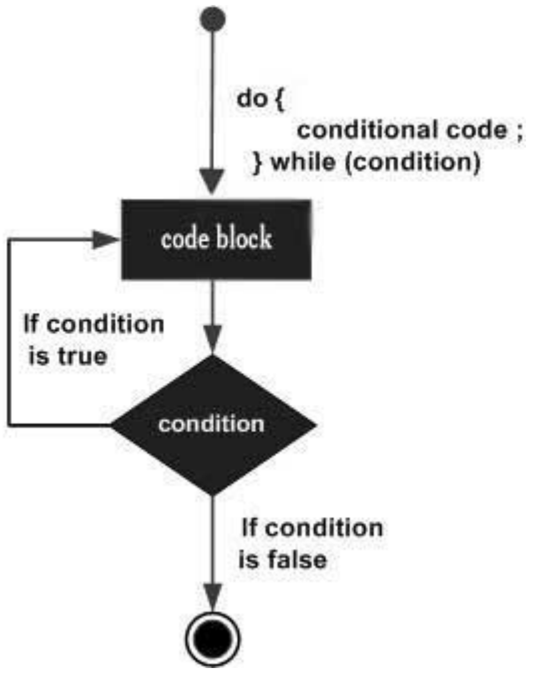
```
Do
 {
   statement(s);
 }
while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop executes once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop executes again. This process repeats until the given condition becomes false.

Example:

```
#include <stdio.h>
 int main () {
   /* local variable definition */
   int a = 10;
   /* do loop execution */
   do {
     printf("value of a: %d\n", a);
     a = a + 1;
   }
   while( a < 15 );
    return 0;
}
```



The above code is produces the following output:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

Nested Loops:

C programming allows to use one loop inside another loop. The following section shows a few examples to illustrate the concept.

Syntax: Nested For-Loop

```
for ( init; condition; increment )
 {
    for ( init; condition; increment )
     {
        statement(s);
     }
   statement(s);
 }
```

Syntax: Nested do-While Loop

```
do
   {
   statement(s);
     do
       {
         statement(s)
       } while(condition);
   } while (condition);
```

Syntax: Nested While Loop

```
while(condition)
 {
   while(condition)
   {
       statement(s);
    }
   statement(s);
 }
```

A final note on loop nesting is that you can put any type of loop inside any other type of loop. For example, a 'for' loop can be inside a 'while' loop or vice versa.

Example

The following program uses a nested for loop to find the prime numbers from 2 to 100 −

```
        #include <stdio.h>
         int main ()
         {
           /* local variable definition */
           int i, j;
              for(i = 2; i<20; i++)
             {
                    for(j = 2; j <= (i/j); j++)
                         if(!(i%j)) break; // if factor found, not prime
                         if(j > (i/j)) printf("%d is prime\n", i);
              }
            return 0;
          }
```

When the above code is compiled and executed, it produces the following result −

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
```

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C supports the following control statements.

S.N.	Control Statement & Description
1	break statement : Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
2	continue statement : Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	goto statement : Transfers control to the labeled statement.

The Infinite Loop

A loop becomes an infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the 'for' loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <stdio.h>

int main () {

   for( ; ; ) {
      printf("This loop will run forever.\n");
   }

   return 0;
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C programmers more commonly use the for(;;) construct to signify an infinite loop.

NOTE − You can terminate an infinite loop by pressing Ctrl + C keys.

Break statement:

The **break** statement in C programming has the following two usages −

- When a **break** statement is encountered inside a loop, the loop is immediately terminated and the program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement (covered in the next chapter).

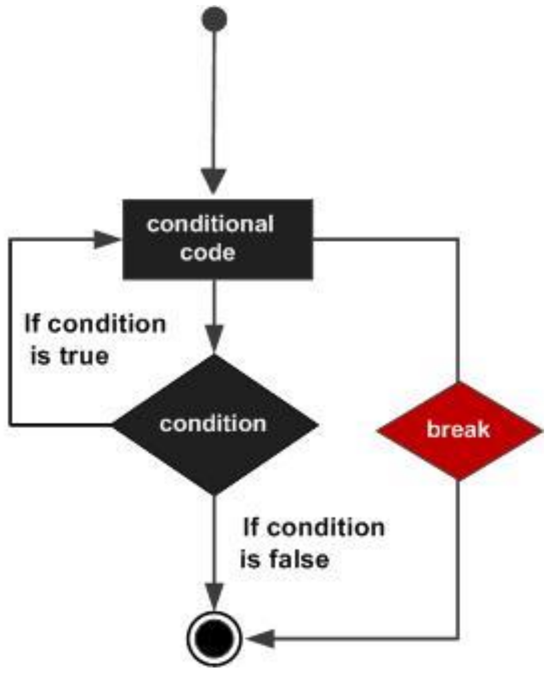
If you are using nested loops, the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

Syntax:

break;

Example:

```
#include <stdio.h>
 int main ()
{
  /* local variable definition */
   int  a = 10;
  /* while loop execution */
  while( a < 20 )
     {
       printf("value of a: %d\n", a);
       a++;
```



```
          if( a > 15)
          {
            /* terminate the loop using break statement */
            break;
          }
        }
      return 0;
    }
```

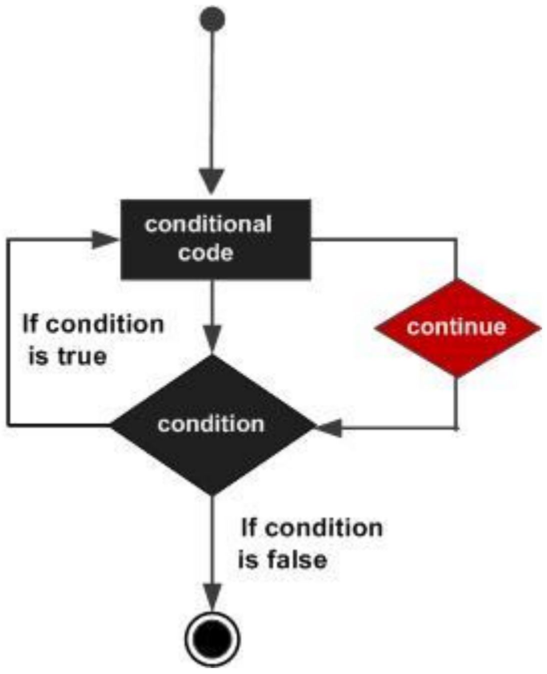
When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

Continue Statement:

The **continue** statement in C programming works somewhat like the **break** statement. Instead of forcing termination, it forces the next iteration of the loop to take place, skipping any code in between. For the **for** loop, **continue** statement causes the conditional test and increment portions of the loop to execute. For the **while** and**do...while** loops, **continue** statement causes the program control to pass to the conditional tests.

Syntax: continue;



Example:

```
    #include <stdio.h>
     int main ()
    {
      /* local variable definition */
      int a = 10;
      /* do loop execution */
      do {
          if( a == 15)
            {
              /* skip the iteration */
              a = a + 1;
              continue;
            }
          printf("value of a: %d\n", a);
```

```
        a++;
            } while( a < 20 );
          return 0;
        }
```

When the above code is compiled and executed, it produces the following result –

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Goto statement:

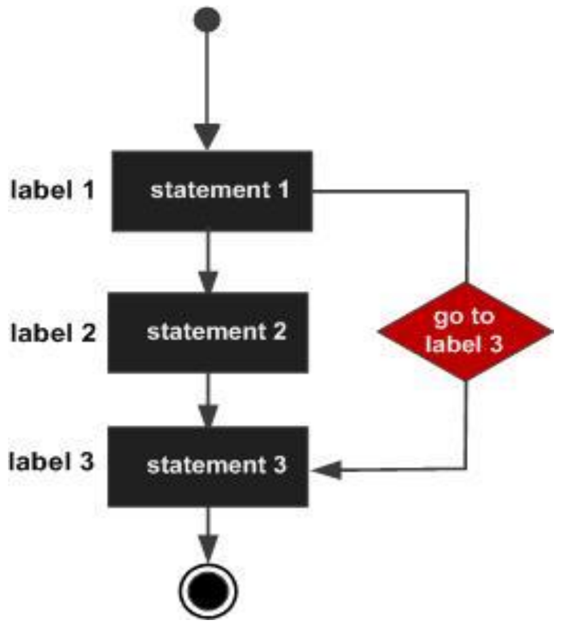
A **goto** statement in C programming provides an unconditional jump from the 'goto' to a labeled statement in the same function.

NOTE – Use of **goto** statement is highly discouraged in any programming language because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten to avoid them.

Syntax:

```
goto label;
..
.
label: statement;
```

Here **label** can be any plain text except C keyword and it can be set anywhere in the C program above or below to **goto** statement.



Example:

```
    #include <stdio.h>
     int main () {
       /* local variable definition */
       int a = 10;
       /* do loop execution */
       LOOP:do {
           if( a == 15) {
           /* skip the iteration */
           a = a + 1;
           goto LOOP;       }
```

```
            printf("value of a: %d\n", a);
            a++;
          } while( a < 15 );
          return 0;
       }
```

When the above code is compiled and executed, it produces the following result −

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
```

Counter controlled loops : The type of loops, where the number of the execution is known in advance are termed by the counter controlled loop. That means, in this case, the value of the variable which controls the execution of the loop is previously known. The control variable is known as counter. A counter controlled loop is also called definite repetition loop.

Example : A while loop is an example of counter controlled loop.

```
sum = 0;
n = 1;
while (n <= 10)
{
sum = sum + n*n;
n = n+ 1;
}
```

Condition Controlled Loops:

A Condition-Controlled loop keeps going until a certain condition is met, like say the user clicks a button, or the world ends or something. A Counter controlled loop keeps going until it has run a certain number of times.

For example if you create a variable x=0. And then every time you look runs you increase x by 1 (x=x+1), you can tell your loop to keep running until x=5. That way the loop would run 5 times until the it reaches 5.

Example:

```
int sum=0, number=1;
 while(number<20)
 {
          sum=sum+number;           /* these statements execute as long as number value
          number=number +1;            less than or equal to 20*/
 }
```