Unit-III : Decision Making & Looping

Decision Making & Looping

Loops are very basic and very useful programming facility that facilitates programmer to execute any block of code lines repeatedly and can be controlled as per conditions added by programmer. It saves writing code several times for same task.

There are three types of loops in C.

For loop

Do while loop

While loop

<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:

The syntax of a **for** loop in C programming language is:

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

Here is the flow of control in a for loop:

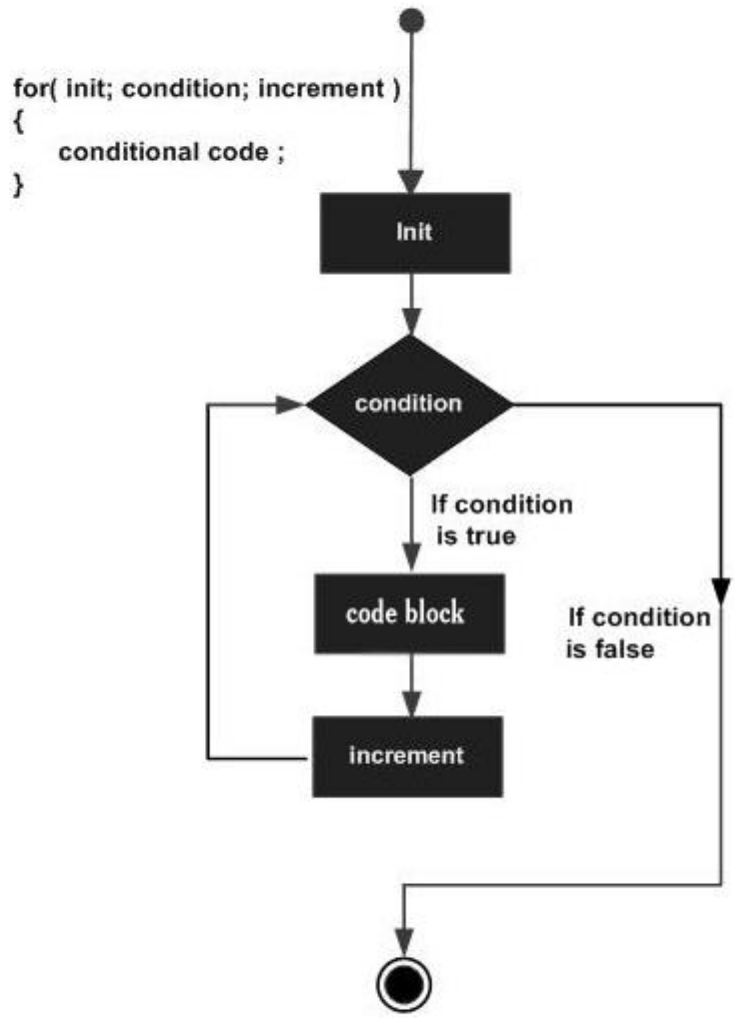
The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.

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 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.

Flow Diagram:



Example:

```
#include <stdio.h>

int main ()
{
  /* for loop execution */
  for( int a = 10; a < 20; a = a + 1 )
  {
     printf("value of a: %d\n", a);
  }

  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
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

while loop in C

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

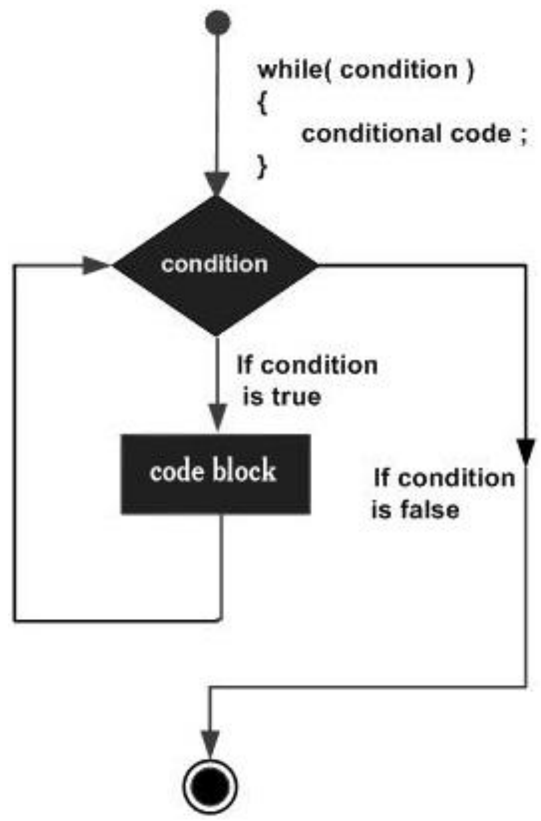
Syntax:

The syntax of a **while** loop in C programming language is:

```
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, program control passes to the line immediately following the loop.

Flow Diagram:



Here, key point of the *while* loop is that the loop might not ever run. 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>

int main ()
{
  /* local variable definition */
  int a = 10;

  /* while loop execution */
  while( a < 20 )
  {
    printf("value of a: %d\n", a);
```

```
      a++;
   }

   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
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

<u>do...while loop in C</u>

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

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

Syntax:

The syntax of a **do...while** loop in C programming language is:

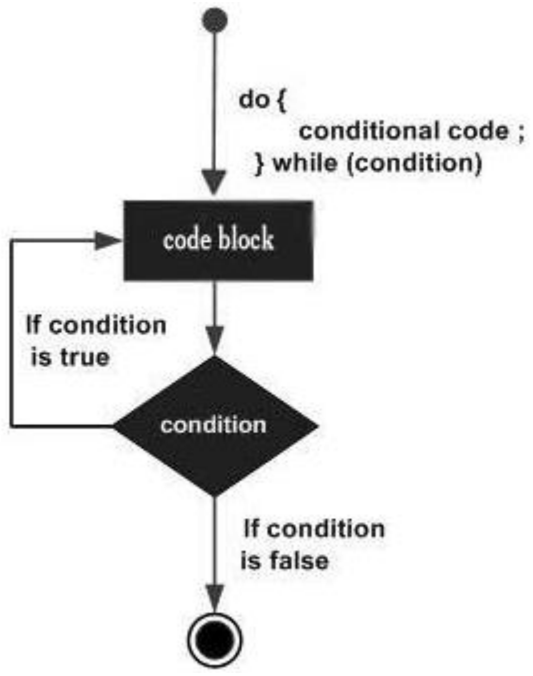
```
do
{
   statement(s);

}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute 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 execute again. This process repeats until the given condition becomes false.

Flow Diagram:



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 < 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: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Loops are group of instructions executed repeatedly while certain condition remains true. There are two types of loops, counter controlled and sentinel controlled loops (repetition).

Counter controlled repetitions are the loops which the number of repetitions needed for the loop is known before the loop begins; these loops have control variables to count repetitions. Counter controlled repetitions need initialized control variable (loop counter), an increment (or decrement) statement and a condition used to terminate the loop (continuation condition).

Sentinel controlled repetitions are the loops with an indefinite repetitions; this type of loops use “sentinel value” to indicate “end of iteration”.

Loops are mostly used to output the data stored in arrays, however they are also used for sorting and searching of data.

‘For’ Loop

“For” loops is a counter controlled repetition; therefore the number iterations must be known before the loop starts.

```
for(control-variable; continuation-condition;increment/decrement-control) {
code to iterate
}
```

Hint: if the code to iterate is only a single line then the braces ({ }) can be ignored.

Diagram 1 illustrates ‘for statement’ operation.

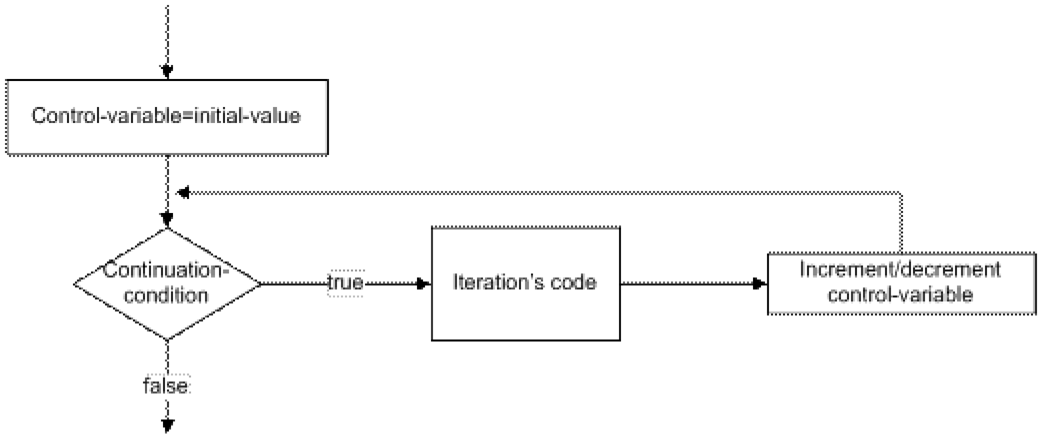
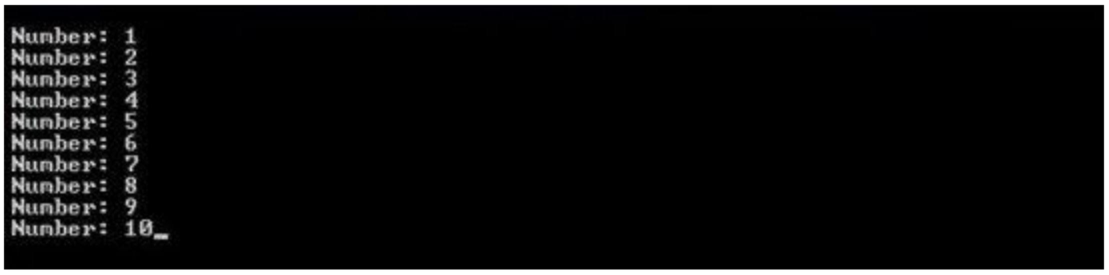

Control-variable=initial-value
Continuation-condition
true
Iteration's code
Increment/decrement control-variable
false

Diagram SEQ Diagram * ARABIC 1 for statement

Example: consider the case of repeating the same line of code for 10 times. We write the code below.

```
int counter;
    for(counter=1; counter<=10; counter++)
    printf("n Number: %d",counter);
```



```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Number: 10_
```

Figure 1 simple program counting from 1 to 10

Description: “statement 1” declares the control variable of this ‘for’ loop. “Statement 2” is the most important part of this code. It initializes the control variable; it sets the continuation condition and it increments the control variable after every successful iteration. Diagram 2 clearly shows how this code works.

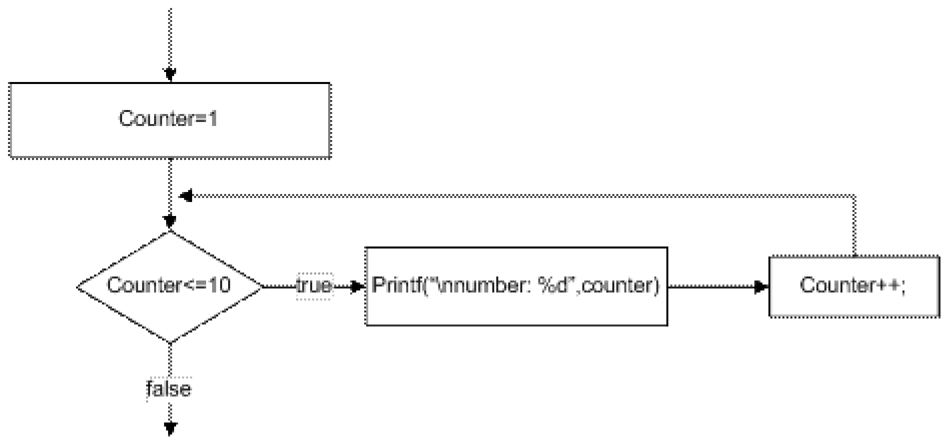


Diagram 2 code 1's operation

The While Statement

"while" statement is a sentinel controlled repetition which can be iterated indefinite number of times. Number of iterations is controlled using a sentinel variable (test expression).

```
while(test-expression)
{
code to execute
}
```

Hint: test-expression must be initialized otherwise errors will be generated when trying to compile the code.

Diagram 3 illustrates how "while" statement operates

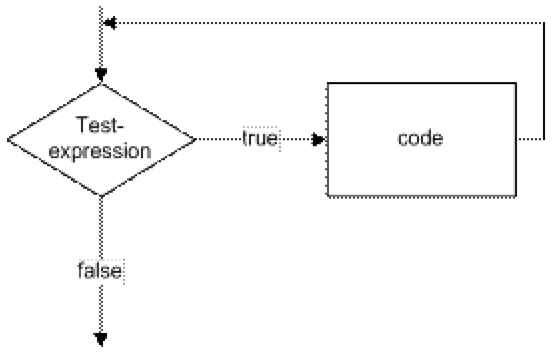
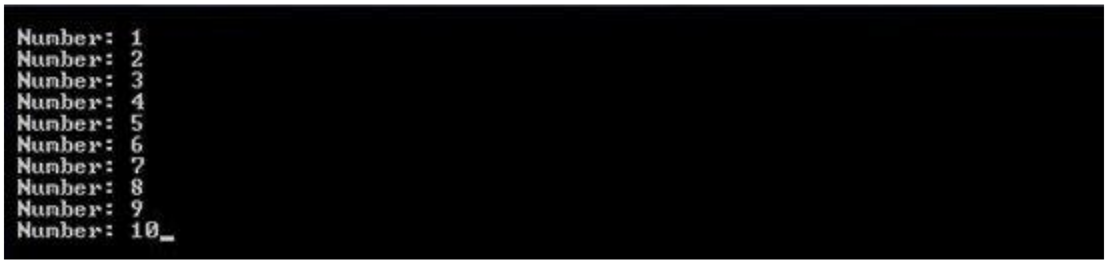


Diagram 3 while statement

Hint: sentinel variable (test expression) must be controlled within the "while" statement, otherwise the loop will run forever.

Example: here we will consider the same example as we did for "for" loop; this is to see how two loops differ from each other.

```
int counter=1;
while(counter <=10)
{
printf("n Number: %d",counter);
counter++;
}
```



```
Number: 1
Number: 2
Number: 3
Number: 4
Number: 5
Number: 6
Number: 7
Number: 8
Number: 9
Number: 10_
```

Figure 2 counting from 1 to 10 using while loop

Description: this code does the same job as the code 1 but using "while" statement instead of "for" statement.

The 'do..while' statement

"do..while" statement is a sentinel controlled repetition which is quite different from the other two statements we covered earlier. This statement runs the code first and then checks the test-expression, so there is always a guarantee that the code runs at least once. This type of loop is generally used for password checks and menus.

```
do{
code to iterate
}while(test-expression)
```

Hint: you must initialize the test-expression to avoid possible errors.

Diagram 4 illustrates the operation of "do..while" statement.

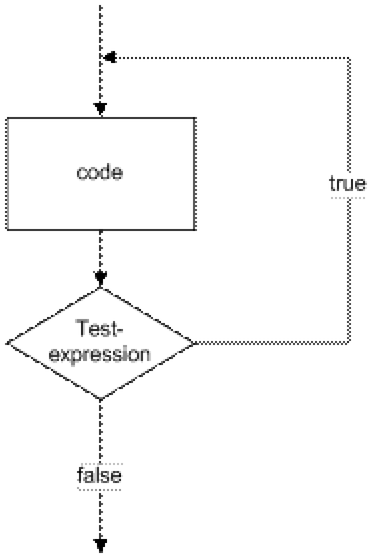
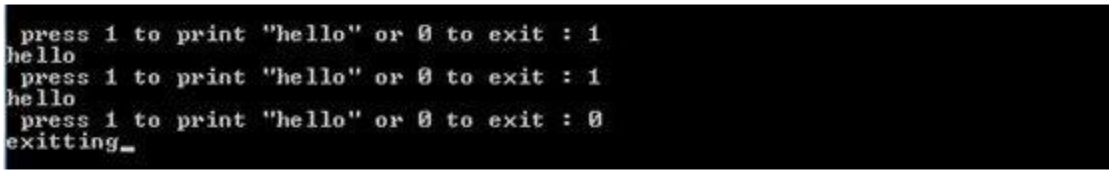


Diagram 4 do..while statement

Example #4: here we will consider a simple menu using "do..while" this code will repeat the menu until "0" is inputted.

```
int input;
do
{
printf("n press 1 to print "hello" or 0 to exit : ");
scanf("%d",&input);
switch(input)
{
case 1: printf("hello"); break;
case 0: printf("exitting");break;
}
}
while(input !=0);
```



```
 press 1 to print "hello" or 0 to exit : 1
hello
 press 1 to print "hello" or 0 to exit : 1
hello
 press 1 to print "hello" or 0 to exit : 0
exitting_
```

Figure 3 simple menu

Description: “switch statement” is used to create a simple menu which allows actions for two different cases. If and input is 0 then the “while” loop will be terminated once the test-expression is reached causing the program to exit.

The Break Statement

“break” statement is used to exit the iteration. This statement is usually used when early escape from the loop is acceptable by the programmer. For example if we are searching for a data, we can use “break” as soon as we find the data to exit the loop. You can simply use “break” by writing “break;” at the point where you want to escape the loop. “break” can be used with all the statements we have covered in this tutorial.

Diagram 5 shows how the program flow changes by “break” statement.

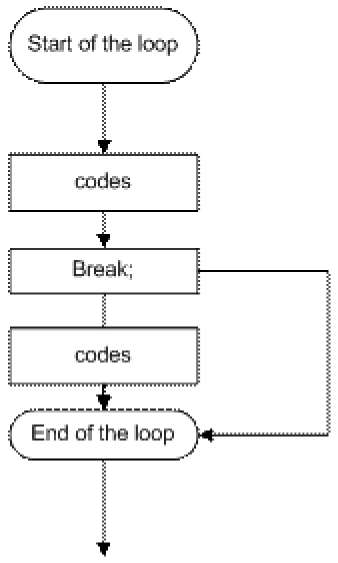
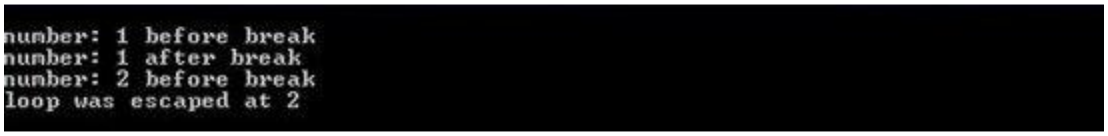


Diagram 5 use of "break"

Example: in this example, we will write a program which will escape the loop using “break” as soon as the number is equal to “2”.

```
int counter;
for(counter=1;counter<=10;counter++)
{
printf("nnumber: %d before break",counter);
if(counter==2)
 break;
 printf("nnumber: %d after break",counter);
}
printf("nloop was escaped at %d",counter);
```



```
number: 1 before break
number: 1 after break
number: 2 before break
loop was escaped at 2
```

Figure 4 use of "break"

Description: counter value is initialized to ‘1’. Therefore “statements 1 and 2“ will all be executed in the first iteration, but when the counter is incremented; “statement 1,2 and 4 “ will be executed because “statement 2” will escape out of the loop and go to the first line after the loop.

The Continue Statement

“continue” statement is used to skip the remaining code of the loop and start the new one. This statement can be implemented by “continue;”.

Diagram 6 shows the flow of the loop.

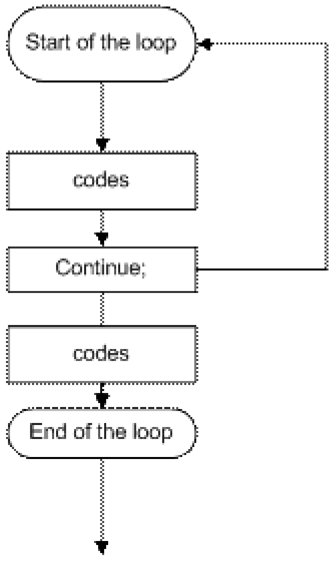


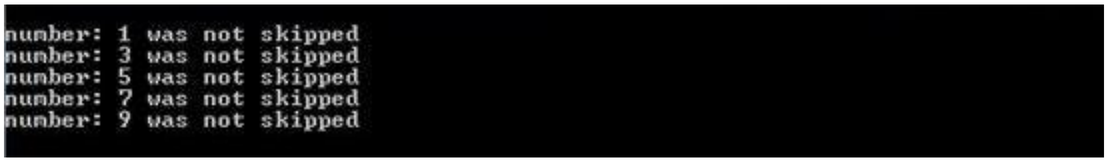
Diagram 6 "continue" statement

Example: we will consider the case in which the loop will skip all the even numbers between 1 and 10.

```
int counter;
for(counter=1;counter<=10;counter++)
{
```

```
if(counter%2==0)
continue;
printf("nnumber: %d was not skipped",counter);

}
```



```
number: 1 was not skipped
number: 3 was not skipped
number: 5 was not skipped
number: 7 was not skipped
number: 9 was not skipped
```

Figure 5 odd numbers between 1 and 10

Description: “for” statement will iterate 10 times counting from 1 to 10. Every time the counter reaches a number which is a multiple of 2 (counter%2==0), “continue” statement will be used to skip to the next number.