The loop control structures

The versatility of the computer lies in its ability to perform a set of instructions repeatedly. This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied. This repetitive operation is done through a loop control instruction.

There are three methods by way of which we can repeat a part of a program. They are:

1) Using a **for** loop.
2) Using a **while** loop.
3) Using a **do-while** loop.

1) The for loop- It is the most popular looping instruction. The **for** allows us to specify three things about a loop in a single line:

(a) Setting a loop counter to an initial value.
(b) Testing the loop counter to determine whether its value has reached the number of repetitions desired.
(c) Increasing the value of loop counter each time the program segment within the loop has been executed.

The general form of **for** statement is as given below :

```
for ( initialise  counter ; test  counter ; increment  counter )
{
    do this ;
    and this ;
    and this ;
}
```

……..

Example-
```
for(c=1;c<=10;c++)
{
  printf("%d", c);
}
```

……....

Let us now examine how the **for** statement gets executed:

- When the **for** statement is executed for the first time, the value of **c** is set to an initial value 1.
- Now the condition **c<= 10** is tested. Since **c** is 1 the condition is satisfied and the body of the loop is executed for the first time.
- Upon reaching the closing brace of **for**, control is sent back to the **for** statement, where the value of **c** gets incremented by 1.

- Again the test is performed to check whether the new value of **c** exceeds 10.

- If the value of **c** is still within the range 1 to 10, the statements within the braces of **for** are executed again.

- The body of the **for** loop continues to get executed till **c** doesn't exceed the final value 10.

- When **c** reaches the value 11 the control exits from the loop and is transferred to the statement (if any) immediately after the body of **for**.

Note: 1. The increment operator **++** increments the value of **i** by 1, every time the statement **i++** gets executed. Similarly, to reduce the value of a variable by 1 a decrement operator **--** is also available.
However, never use **n+++** to increment the value of **n** by 2, since C doesn't recognize the operator **+++**. (use n=n+2)
2. The initialization, testing and incrementation part of a **for** loop can be replaced by any valid expression.

Nesting of Loops

The way **if** statements can be nested, similarly the **for** can also be nested. To understand how nested loops work, look at the program given below:

```
/* Demonstration of nested loops */
        void main( )
          {
            int r, c, sum ;
           for ( r = 1 ; r <= 3 ; r++ )         // outer loop
            {
                for ( c = 1 ; c <= 2 ; c++ )    // inner loop
                {
                 sum = r + c ;
                 printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;
                }
            }
          }
```

When you run this program you will get the following output:

r = 1 c = 1 sum = 2
r = 1 c = 2 sum = 3
r = 2 c = 1 sum = 3
r = 2 c = 2 sum = 4
r = 3 c = 1 sum = 4
r = 3 c = 2 sum = 5

Here, for each value of **r** the inner loop is cycled through twice, with the variable **c** taking values from 1 to 2. The inner loop terminates when the value of **c** exceeds 2, and the outer loop terminates when the value of **r** exceeds 3.

Multiple Initialisations in the for Loop

The initialisation expression of the **for** loop can contain more than one statement separated by a comma. For example,

```
for ( i = 1, j = 2 ; j <= 10 ; j++ )
```

Multiple statements can also be used in the incrementation expression of **for** loop; i.e., we can increment (or decrement) two or more variables at the same time. However, only one expression is allowed in the test expression. This expression may contain several conditions linked together using logical operators.

2) The *while* Loop – The general form of **while** is as given below:

```
initialize loop counter ;
while ( test loop counter using a condition )
  {
     do this ;
     and this ;
     increment loop counter ;
  }
```

Note:

- The statements within the **while** loop would keep on getting executed till the condition being tested remains true. When the condition becomes false, the control passes to the first statement that follows the body of the **while** loop. In place of the condition there can be any other valid expression. So long as the expression evaluates to a non-zero value the statements within the loop would get executed.

- The statements within the loop may be a single line or a block of statements. In the first case the parentheses are optional.

- As a rule the while must test a condition that will eventually become false, otherwise the loop would be executed forever, indefinitely.

- Instead of incrementing the loop counter, we can even decrement it and still manage to get the body of the loop executed repeatedly.

- It is not necessary that a loop counter must only be an int. It can even be a float.

❖ **Remember:**

```
void  main()
  {
    int i=1;
    while(i<=32767)
     {
        printf("%d\n",i);
         i=i+1;
     }
    }
```

It's an indefinite loop. To begin with, it prints out numbers from 1 to 32767. After that value of **i** is incremented by 1, therefore it tries to become 32768, which falls outside the valid integer range, so it goes to other side and becomes -32768 which would certainly satisfy the condition in the **while**. This process goes on indefinitely.

```
void  main()
  {
    int i=1;
    while(i<=10);
     {
        printf("%d\n",i);
         i=i+1;
     }
   }
```

This is another indefinite loop, and it doesn't give any output at all. The reason is, we have carelessly given a **;** after the **while**. Since the value of **i** is not getting incremented the control would keep rotating within the loop, eternally.

Note: The **+=** is a compound assignment operator. It increments the value of **i** by 1. Similarly, **j = j + 10** can also be written as **j += 10**. Other compound assignment operators are **-=, *=, / =** and **%=**.

Post increment operator- Consider the following code fragment –

```
void  main()
{
   int i=0;
   while(i++<10);
    printf("%d\n",i);
  }
```

In the statement **while (i++ < 10)**, firstly the comparison of value of **i** with 10 is performed, and then the incrementation of **i** takes lace. Since the incrementation of **i** happens after its usage, here the ++ operator is called a post-incrementation operator. When the control reaches **printf( )**, **i** has already been incremented, hence **i** must be initialized to 0.

Pre increment operator- Consider the following code fragment –

```
void  main()
{
   int i=0;
   while(++i<=10);
    printf("%d\n",i);
  }
```

In the statement **while (++i <= 10)**, firstly incrementation of **i** takes place, then the comparison of value of **i** with 10 is performed. Since the incrementation of **i** happens before its usage, here the ++ operator is called a pre-incrementation operator.

3) The *do-while* Loop-

The **do-while** loop looks like this:

```
do
{
     this ;
     and  this  ;
     and  this  ;
     and this ;
} while ( this condition is true ) ;
```

Note: There is a minor difference between the working of **while** and **do-while** loops. This difference is the place where the condition is tested. The **while** tests the condition before executing any of the statements within the **while** loop. As against this, the **do-while** tests the condition after having executed the statements within the loop.

This means that **do-while** would execute its statements at least once, even if the condition fails for the first time. The **while**, on the other hand will not execute its statements if the condition fails for the first time.

The Odd Loop- The loops that we have used so far executed the statements within them a finite number of times. However, in real life programming one comes across a situation when it is not known beforehand how many times the statements in the loop are to be executed. This situation can be programmed as shown below:

```
/* Execution of a loop an unknown number of times*/
void main()
{
  char another;
  int num;
  do
  {
    printf ( "Enter a number " ) ;
    scanf ( "%d", &num ) ;
    printf ( "square of %d is %d", num, num * num ) ;
    printf ( "\nWant to enter another number y/n " ) ;
    scanf ( " %c", &another ) ;
  } while (another==’y’);
}
```

In this program the **do-while** loop would keep getting executed till the user continues to answer y. The moment he answers n, the loop terminates, since the condition (**another == 'y'**) fails. Note that this loop ensures that statements within it are executed at least once even if **n** is supplied first time itself.

The *break* Statement- We often come across situations where we want to jump out of a loop instantly, without waiting to get back to the conditional test. The keyword **break** allows us to do this. When **break** is encountered inside any loop, control automatically passes to the first statement after the loop. A **break** is usually associated with an **if**.

The *continue* Statement- In some programming situations we want to take the control to the beginning of the loop, bypassing the statements inside the loop, which have not yet been executed. The keyword **continue** allows us to do this. When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop. A **continue** is usually associated with an **if**.