Control Structures in C
Part - II

Module – 1

LOOP CONTROL STATEMENTS IN C

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 until a particular condition is being satisfied. There are methods by which we can repeat a part of program. Usually there are circumstances were you want to execute the same statements many times.
For instance you want to print the same words ten times. You could type ten printf functions, but it is easier to use a loop. The only thing you have to do is to setup a loop that executes the same printf function ten times.

In looping, sequences of statements are executed until some conditions are satisfied. A loop control construct therefore consists of two segments:

1. Control statement
2. Body of the loop

The control statements test certain conditions and then direct the repeated execution of the statements contained in the body of the loop.

Depending on the position of control statement in the loop, control structures can be classified into two:

1. Entry controlled loop (pre-test loop)
2. Exit controlled loop (post-test loop)

In entry controlled loop, the conditions are tested before the execution of the body part of the loop. If the conditions are not satisfied the body of the loop will not be executed. In case of an exit controlled loop, the test is performed at the end of the body of the loop and therefore the body is executed unconditionally for the first time.

A looping process normally consists of 4 steps:

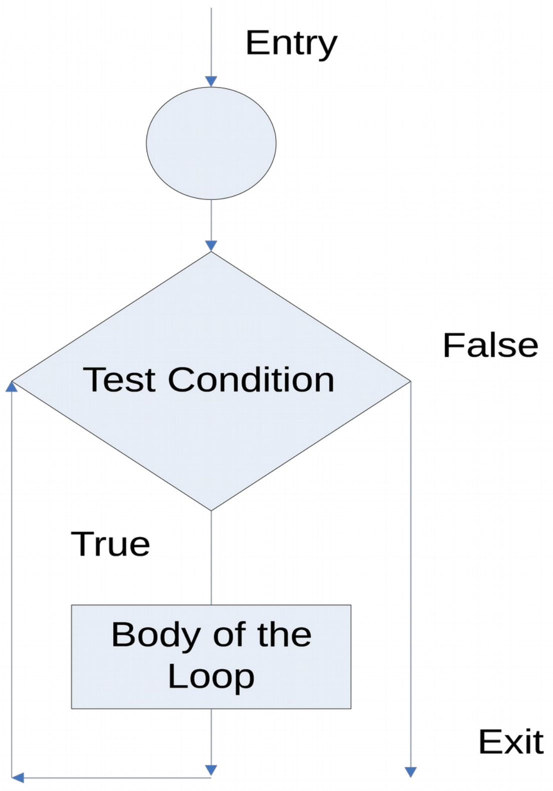
1. Initializing and setting of a conditional variable.
2. Execution of the statements inside the loop.
3. Incrementing or updating the condition variable.
4. Test for a specified value of the condition variable for execution of the loop.

Based on the nature of the control variable and the kind of variable assigned to it the loops can be broadly classified into two categories:

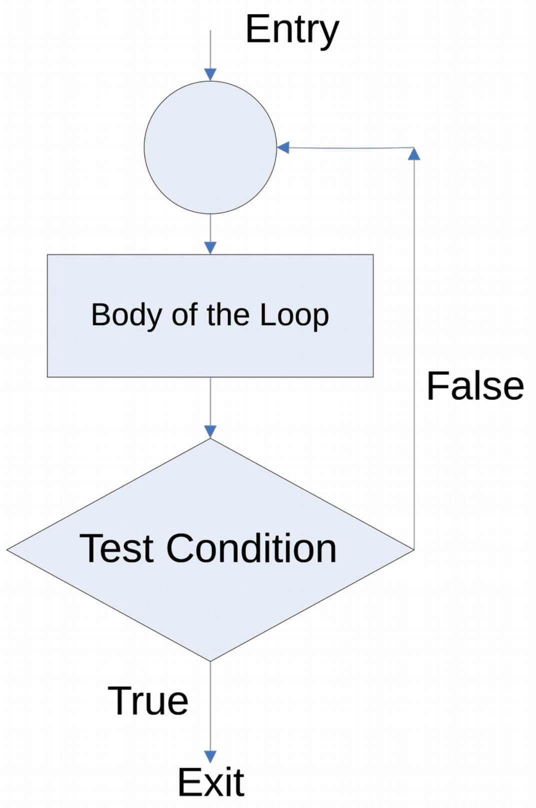
1. **Counter Controlled Loop**: Counter-controlled loop are loops which will repeat a specified number of times where the number of repetitions is fixed prior to entering the loop. Many processing applications have this requirement.

2. **Sentinel Controlled Loop:** In a Sentinel-Controlled loop, a special value called a sentinel value is used to change the loop control expression from true to false. For example, when reading data we may indicate the "end of data" by a special value, like -1 or 999. A sentinel-Controlled loop is often called indefinite repetition loop because the number of repetitions is not known before the loop being executed.

The flow chart for the entry controlled loop is shown in the screen.



Next we will see the flowchart for the exit controlled loop.



Next we will understand the usage of a simple entry controlled loop structure.

Module – 2

The while constructs

The while loop is used when a statement is to be executed repeatedly until a given condition is satisfied. The loop is terminated when the expression yields false value. Perhaps you want to calculate gross salaries of ten different persons, or you want to convert temperatures from centigrade to fahrenheit for 25 different cities. You can easily implement these with the help of **while** loop.

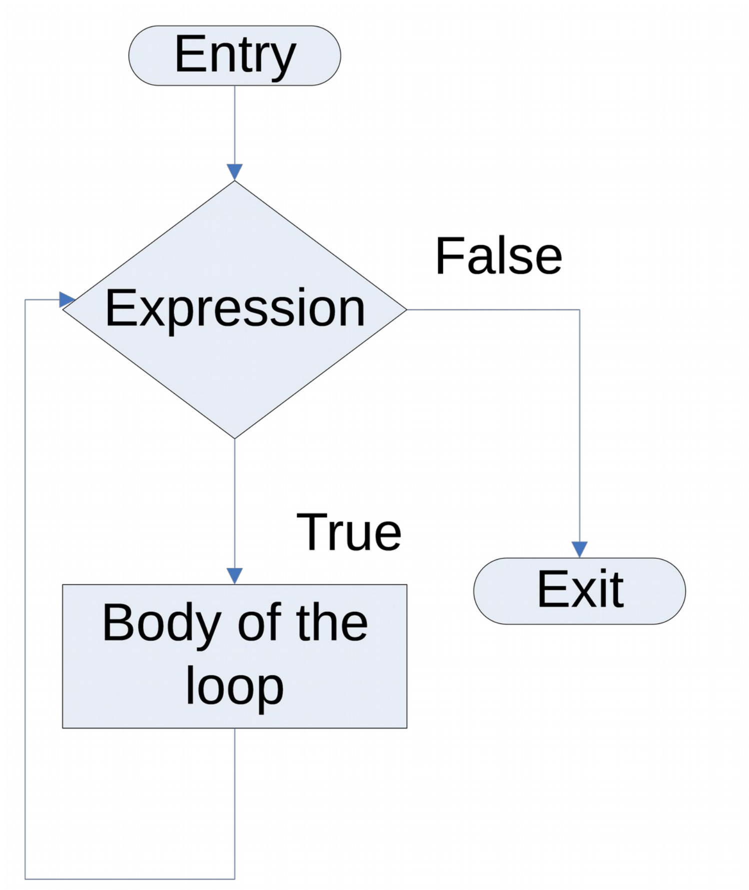
The basic format of the while loop is :

```
while (expression)
{
        // body of the loop

}
```

As I told earlier, the while loop is an entry controlled loop in which the test condition is checked first, and if the condition is satisfied the body of the loop is executed and the conditional parameter is updated ; again the conditional instruction is checked and the above steps are repeated until the condition become false. On exit the program execution is continued with the very next sentence after the end of the loop.

Let us now understand the flow of while loop with the help of the flow chart.



So long as the expression evaluates to a non-zero value the statements within the loop would get executed. The condition being tested may use relational or logical operators as shown in the following examples:

```
while ( i <= 10 )
while ( i >= 10 && j <= 15 )
while ( j > 10 &&( b < 15 || c < 20 ) )
```

the statements within the loop may be a single line or a block of statements. In the first case parenthesis are optional for example

```
while (n>1)
      n- -;
```

is same as

```
while (i>1)
```

```
{
        n- -;
}
```

As a rule the while must test a condition that will eventually become false, otherwise the loop executed forever indefinitely. That is , when an expression returens always true value, then the loop becomes an infinite one. For example,

```
while (1)
{
  Statement;
}
```

is an infinite loop. It is equivalently written as,

```
while (!=0)
{
Statement;
}
```

To form an infinite loop.

Now, let us go through a demonstration of C program which uses a while loop to find the solution of the equation y = x^n.

```
    main()
    {
            int count , n;
            float x,y;

             printf("Enter values of x and n");
             scanf("%f %d",&x,&n);
              y=1.0;
              count=1;
             while(count<=n)
             {
    `                   y=y*x;
                        count++;
             }
    printf("x=%f; x to power n=%f",x,y);

    }
```

The control structure is an entry controlled loop named **for loop structure.**

Module – 3

The for construct

It is the most powerful loop construct used frequently in many programmes. The 'for loop structure' allows us to specify three things about a loop in a single line:

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

General form of the for loop is :

For (initialization; test-condition; increment)

```
            {
                    Body of the loop
            }
```

The execution of the **for** loop is as follows:

1 **control variable initialization**: control variables are initialized using assignment operator.
2 **Test condition checking** : the value of the control variable is tested using the test condition . usually relational expressions are used for conditional checking.
3.Body part is executed and the control is transferred back to the for statement. Now the control variable is incremented using an assignment statement and the new value of the control variable is tested again to see whether the condition is satisfied and the process is continued until the condition become success.

Next we will understand the use of for loop with the help of a program to calculate simple interest for 3 sets of p, n and r.

```
#include<stdio.h>
main ( )
{
        int p, n, count ;
        float r, si ;
        for ( count = 1 ; count <= 3 ; count = count + 1 )
        {
                printf ( "Enter values of p, n, and r " ) ;
                scanf ( "%d %d %f", &p, &n, &r ) ;
                si = p * n * r / 100 ;
                printf ( "Simple Interest = Rs.%f\n", si ) ;
        }
}
```

If this program is compared with the one written using **while**, it can be seen that the three steps—initialization, testing and instrumentation—required for the loop construct have now been incorporated in the **for** statement.

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

- When the **for** statement is executed for the first time, the value of **count** is set to an initial value 1.
- Now the condition **count <= 3** is tested. Since **count** 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 **count** gets incremented by 1.
- Again the test is performed to check whether the new value of **count** exceeds 3.
- If the value of **count** is still within the range 1 to 3, the statements within the braces of **for** are executed again.
- The body of the **for** loop continues to get executed till **count** doesn't exceed the final value 3.
- When **count** reaches the value 4 the control exits from the loop and is transferred to the statement (if any) immediately after the body of **for**.

Additional features of for loop

More than one variable can be initialized at a time in the for statement. Multiple sections in the initialization and increment sections are separated by commas.

For example :

```
for (m=50 ,n=1 ; n<=m ; n=n+1 , m=m-1)
```

```
{
        p=m/n;
        printf ("%d %d %d", n, m, p);
}
```

Nesting of for loops

one **for** loop with in another for loop is known as nesting of for loops.

For example :

```
for (i=1 ; i<m ; i++)
{
    for (j=i ; j<n;j++)
     {
          ……………………
          ……………………
      }
}
```

Let us now move to another control structure named do-while which is a exit control loop.

Module - 4

The do-while construct

The do-while construct is an exit control loop structure in which the loop statement consists of a block of code and a boolean condition. First the code block is executed, and then the condition is evaluated. If the condition is true, the code block is executed again and again until the condition becomes false.

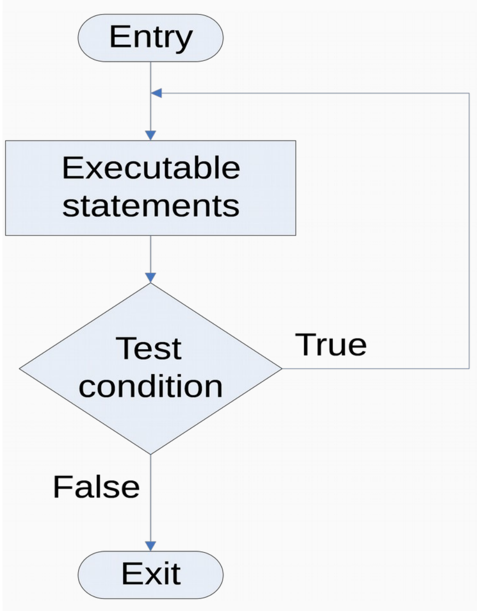
The normal syntax of do–while construct is :

```
do
{

Body of the loop

}while(test condition);
```

Let us see the flow chart representation of the of do-while loop.



Here, at first the block of statements are executed present in the body of the loop and at the end, while statement is executed. If the resultant condition is true then program control goes to evaluate the body of a loop once again. This process continues till condition is true. When it becomes false, then the loop terminates.

It is important to place a semicolon (;) after closing the test condition in do-while construct. Since the loop construct executes the statements within the body of the loop before the test condition is evaluated, the minimum number of times the do-while loop is executed is one. The **while construct**, on the other hand will not execute its statements if the condition fails for the first time. This difference is brought about more clearly with the help of a program.

```
#include<stdio.h>
main( )
{
        while ( 5 < 1 )
        printf ( "Hello world \n") ;
}
```

Here, since the condition fails the first time itself, the **printf( )** will not get executed at all. Let's now write the same program using a **do-while** loop.

```
#include<stdio.h>
main( )
{
        do
        {
         printf ( "Hello world \n") ;
        } while ( 4 < 1 ) ;
}
```

In this program the **printf( )** would be executed once, since first the body of the loop is executed and then the condition is tested.

We have gone through all conditional control structures in C. Let us now discuss the unconditional control structures which include break, continue and goto constructs.

Module – 5

<u>UNCONDITIONAL CONTROL STATEMENTS IN C</u>

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 or in a switch-case control structure it terminate the execution at that point and transfers execution control to the statement immediately following the loop or switch-case construct. Thus it indulges an early exit from any loop or switch-case construct statement.

The general syntax is

break;

Let us consider a C program, to determine whether a number is prime or not. As you aware a prime number is one, which is divisible only by 1 or itself.

The idea that we have, to test whether a number is prime or not, is to divide it successively by all numbers from 2 to one less than itself. If remainder of any of these divisions is zero, the number is not a prime. If no division yields a zero then the number is a prime number. Let us now go through the program that implements this logic.

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

        int num, i ;
        printf ( "Enter a number " ) ;
        scanf ( "%d", &num ) ;
        i = 2 ;
        while ( i <= num - 1 )
        {
                if ( num % i == 0 )
                {
                        printf ( "Not a prime number" ) ;
                        break ;
                }
        i++ ;
}
        if ( i == num )
                printf ( "Prime number" ) ;
}
```

In this program the moment **num % i** turns out to be zero, (i.e. **num** is exactly divisible by **i**) the message “Not a prime number” is printed and the control breaks out of the **while** loop. Why does the program require the **if** statement after the **while** loop at all? Well, there are two ways the control could have reached outside the **while** loop:

1. It jumped out because the number proved to be not a prime.
2. The loop came to an end because the value of **i** became equal to **num**.

When the loop terminates in the second case, it means that there was no number between 2 to **num - 1** that could exactly divide **num**. That is, **num** is indeed a prime. If this is true, the program should print out the message “Prime number”.

The keyword **break**, breaks the control only from the loop in which it is placed. Consider the a program, which illustrates this fact.

```
#include<stdio.h>
main( )
{
         int i = 1 , j = 1 ;
        while ( i++ <= 100 )
        {
                while ( j++ <= 200 )
                {
                        if ( j == 150 )
                                break ;
                        else
                                printf ( "%d %d\n", i, j ) ;
                }
        }
}
```

Here when **j** equals 150, **break** takes the control outside the inner **while** only, since it is placed inside the inner **while**.

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. That means it is used within loop structures to end the execution of the current iteration and proceed to the next iteration.

Format is

continue;

A **continue** is usually associated with an **if**. let's consider a program to clearly understand the usage of **continue** statement.

```
#include<stdio.h>
main( )
{
        int i, j ;
        for ( i = 1 ; i <= 2 ; i++ )
        {
                for ( j = 1 ; j <= 2 ; j++ )
                {
                        if ( i == j )
                        continue ;
                        printf ( "\n%d %d\n", i, j ) ;
                }
        }
}
```

Output of the program will be:

```
1 2
2 1
```

Note that when the value of **i** equals that of **j**, the **continue** statement takes the control to the outer **for** loop bypassing rest of the statements pending execution in the inner **for** loop.

Goto and label construct

The goto construct causes an unconditional transfer of execution. Formally, the goto statement is never necessary, and in practice it is almost always easy to write code without it. General format is:

label:

goto label;

where label is an identifier and not a number.

The identifier following goto is a statement label and it is not declared. The name of the statement label can also be used as a variable name in the same program if it is declared properly. The compiler identifies this name as a label it it appears in a goto statement and as a variable if it appears in an expression.

As far as possible use of goto construct is to be avoided, since it affects structured programming.Nevertheless, there are certain situations where gotos may find a place to make the program simpler. The most common is to abandon processing in some deeply nested structure, such as breaking out of two or more loops at once. The break statement cannot be used directly since it only exits from the innermost loop. Thus:

```
for ( ... )
        for ( ... )
        {
                ...
                if (disaster)
                goto error;
        }
...
error:
```

If the statements follow the label ending with a colon(:) appear before a goto statement, the flow is in the backward direction and it is known as backward goto otherwise it is known as forward goto statement. Statements in different places can not have the same label.

Consider a problem to read a string using goto.

```
#include<stdio.h>
main()
{
        char ch;
        Printf("Enter a string\n");
read: ch=getchar();
        If(ch!='\n')
        {
           putchar(ch);
           goto read;
        }
}
```

There are many more other features in C programming language are remaining to explore. So let us wait for the coming sessions. Till then bye.

Assignments

1. Can the action of a do-while structure be simulated by if or if-else structure? Explain with an example.
2. Under what circumstances will the do-while be more appropriate than the for loop structure?
3. Compare the switch-case structure with the if-else structure. Which is more convenient? Give a suitable example.
4. Write a program that finds and displays the number and sum of all integers greater than 100 and less than 200 that are divisible by 7.
5. Write programs using while loop and for loop to reverse the digits of a number.
6. Write a program using do….while loop to calculate and print the first n fibonacci numbers

Reference

1. *B. W. Kernighan and D. M. Ritchie*, The C Programming Language, Prentice-Hall, Englewood Cliffs, New Jersey, 1978.
2. *Greg Perry,* Absolute Beginners' guide to C 2nd Edition, SAMS publishing, A division of Prentice Hall Computer Publishing, 201 West 103rd Street , Indianapolis, Indiana 46290. April 1994.
3. Yashavant Kanetkar; Let us C, BPB Publications, New Delhi.