CL-101 INTRODUCTION TO COMPUTING

LAB-07

CONDITIONAL STATEMENTS (cont.) AND ITERATIONS

NATIONAL UNIVERISTY OF COMPUTER AND EMERGING SCIENCES

SWITCH STATEMENT

A series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken. This is called multiple selection. C provides the switch multiple-selection statement to handle such decision making.
The switch statement consists of a series of case labels, an optional default case and statements to execute for each case.
The switch statement is especially useful when the selection is based on the value of a single variable or of a simple expression (called the controlling expression). The value of this expression may be of type int or char, but not of type double.

C – SYNTAX

```
switch (controlling expression)
{
label set 1:
statements1;
break;
label set 2:
statements 2;
break;
.
.
.
label set n:
statements n;
break;
default:
statements d;
}
```

INTERPRETATION

• The controlling expression, an expression with a value of type int or type char, is evaluated and compared to each of the case labels in the label sets until a match is found. A label set is made of one or more labels of the form case followed by a constant value and a colon.

• When a match between the value of the controlling expression and a case label value is found, the statements following the case label are executed until a break statement is encountered. Then the rest of the switch statement is skipped.

• The statements following a case label may be one or more C statements, so it is not need to make multiple statements into a single compound statement using braces.

• If no case label value matches the controlling expression, the entire switch statement body is skipped unless it contains a default label. If so, the statements following the default label are executed when no other case label value matches the controlling expression

SWITCH CASE STATEMENT AS A SUBSTITUTE FOR LONG IF STATEMENTS

EXAMPLE USING IF - ELSE	EXAMPLE USING SWITCH
#include <stdio.h> int main () { char grade = 'E'; if(grade == 'A') printf("Superb!\n"); else if(grade == 'B') printf("Very good!\n"); else if(grade == 'C') printf("Good.\n"); else if(grade == 'D') printf("Passed.\n"); else if(grade == 'F') printf("Try again.\n"); else printf("Invalid grade.\n"); } printf("Your grade is %c.\n", grade); return 0;	**#include <stdio.h> int main () { char grade = 'F'; switch(grade) { case 'A' : printf("Superb!\n"); break; case 'B' : printf("Very good!\n"); break; case 'C' : printf("Good\n"); break; case 'D' : printf("Passed\n"); break; case 'F' : printf("Try again\n"); break; default : printf("Invalid grade\n"); } printf("Your grade is %c.\n", grade); return 0; }**

C – LOOP CONTROL STATEMENTS

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.

TYPES OF LOOP CONTROL STATEMENTS IN C

There are 3 types of loop control statements in C language. They are,

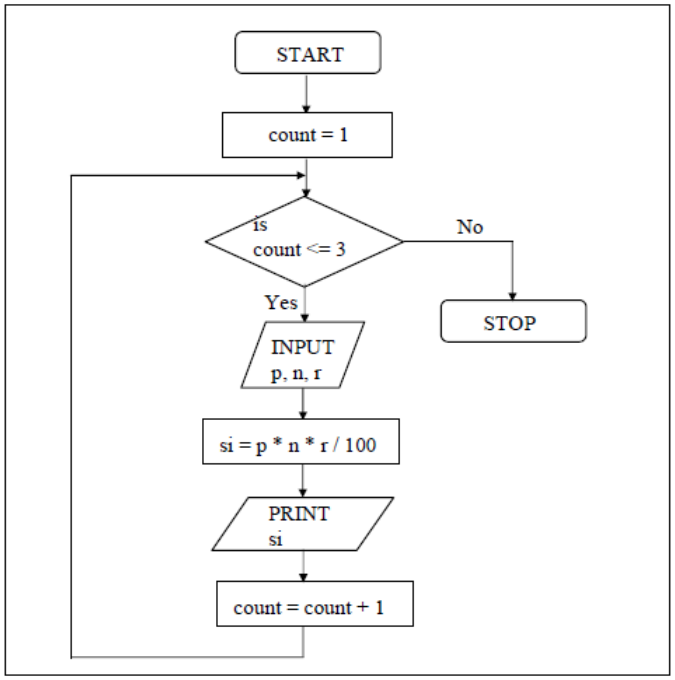
1. while
2. for

3. do-while

Syntax for each C loop control statements are explained below with description.

WHILE LOOP

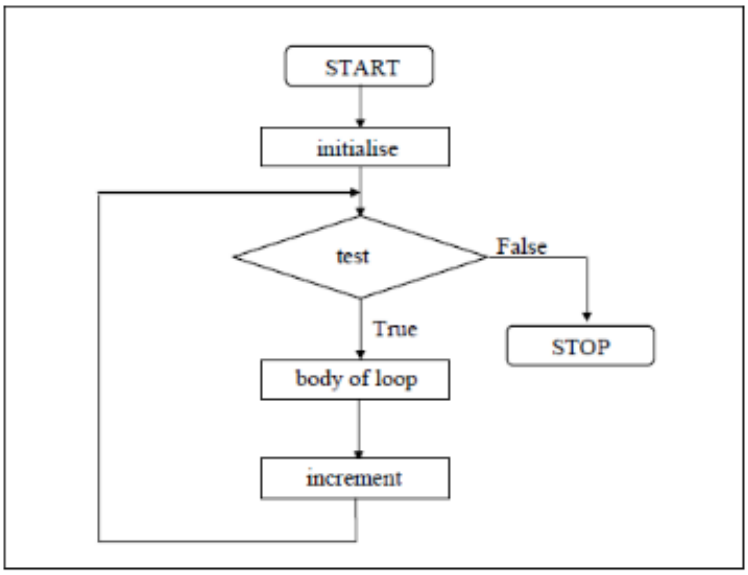
It is often the case in programming that you want to do something a fixed number of times. Perhaps you want to calculate gross salaries of ten different persons, or you want to convert temperatures from centigrade to Fahrenheit for 15 different cities.
The **while** loop is ideally suited for such cases. Let us look at a simple example, which uses a **while** loop. The flowchart shown below would help you to understand the operation of the **while** loop.




```
/* Calculation of simple interest for 3 sets of p, n and r */

main()
{
        int p, n, count;
        float r, si;
        count = 1;
        while (count <= 3)
        {
                printf ("\nEnter values of p, n and r ");
                scanf ("%d %d %f", &p, &n, &r);
                si = p * n * r / 100;
                printf ("Simple interest = Rs. %f", si);
                count = count + 1;
        }
}
```

The program executes all statements after the **while** 3 times. The logic for calculating the simple interest is written within a pair of braces immediately after the **while** keyword. These statements form what is called the 'body' of the while loop. The parentheses after the **while** contain a condition. So long as this condition remains true all statements within the body of the **while** loop keep getting executed repeatedly. To begin with the variable **count** is initialized to 1 and every time the simple interest logic is executed the value of **count** is incremented by one. The variable **count** is many a times called either a 'loop counter' or an 'index variable'.
The operation of WHILE loop is as follows:



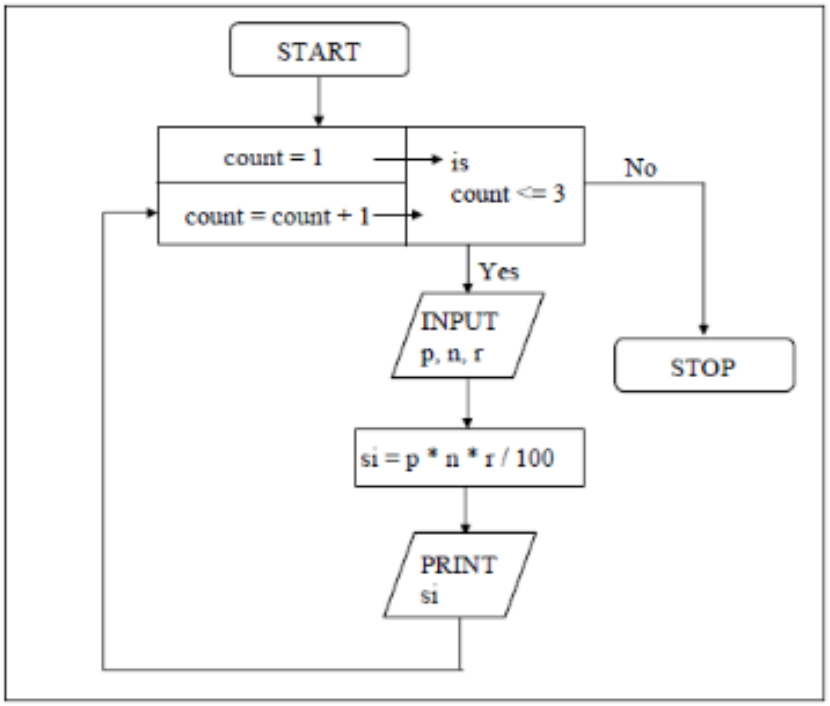
FOR LOOP

Perhaps one reason why few programmers use **while** is that they are too busy using the **for**, which is probably 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 under:

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

Let us write down the simple interest program using **for**. Compare this program with the one, which we wrote using **while**. The flowchart is also given below for a better understanding.



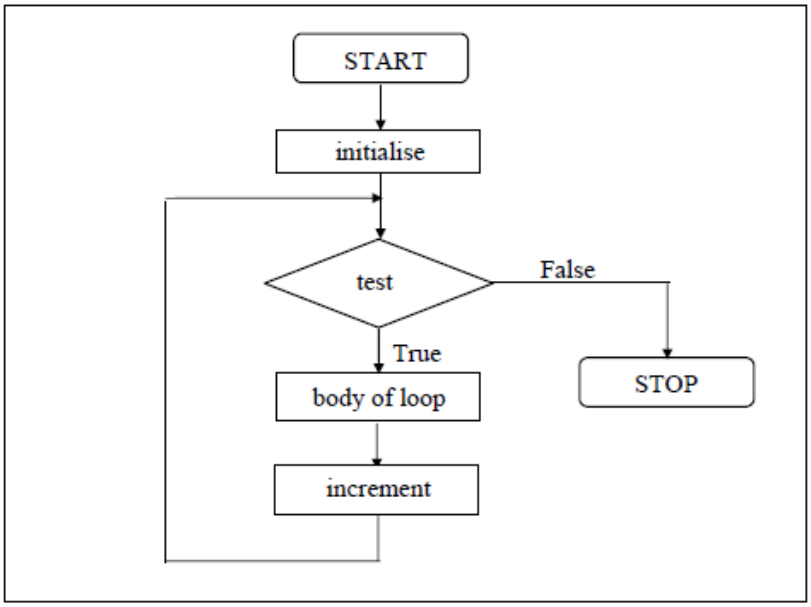

```
/* Calculation of simple interest for 3 sets of p, n and r */
main ()
{
    int p, n, count;
    float r, si;
    for (count = 1; count <= 3; count = count + 1)
    {
        printf ("Enter values of p, n, and r ");
        scanf ("%d %d %f", &p, &n, &r);
        si = p * n * r / 100;
        printf ("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 incrementation — required for the loop construct have now been incorporated in the **for** statement. Let us now examine how the **for** statement gets executed:

- 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**.

The following figure would help in further clarifying the concept of execution of the **for** loop.

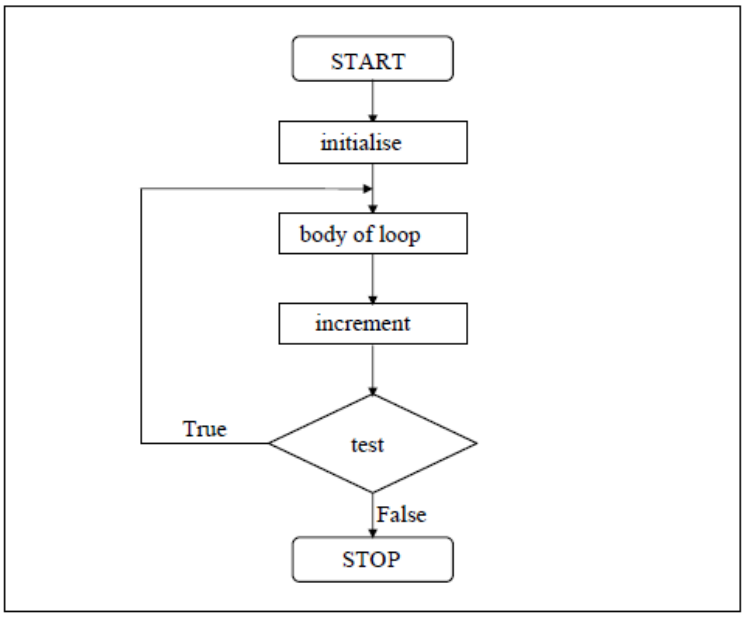


DO-WHILE LOOP

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

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

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. Following figure would clarify the execution of **do-while** loop still further.



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. This difference is brought about more clearly by the following program.

break and **continue** are used with **do-while** just as they would be in a **while** or a **for** loop. A **break** takes you out of the **do-while** bypassing the conditional test. A **continue** sends you straight to the test at the end of the loop (skipping the current iteration).

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**. As an example, let's consider the following example.

Example: Write a program to determine whether a number is prime or not. A prime number is one, which is divisible only by 1 or itself.

All we have to do 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. Following program implements this logic.

```
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");
}
```

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**. As an example, let's consider the following program.

```
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);
        }
    }
}
```

The output of the above program would be...

```
1 2
2 1
```

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