6.2 Loop or Iteration or Repeating Construct

Loop may be defined as block of statements which are repeatedly executed for a certain number of times or until a particular condition is satisfied. When an identical task is to be performed for a number of times, then loop is used. A loop allows to execute certain block of statements repeatedly till a conditional expression is true. When the expression becomes false, the loop terminates and the control passes on to the statement following the loop. A loop consists of two segments, one is known as the `control statement` and the other is the `body` of the loop. The control statement in loop decides whether the body is to be executed or not.

Example 6.6

Write a program to display the message "Welcome to my college" ten times using loop and without using loop

Method I without using loop

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i;
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
printf("\nWelcome to my college");
getch();
return 0;
}
```

Method II using loop

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

```
int i;
for(i=0;i<10;i++)
  printf("\nWelcome to my college");
getch();
return 0;
}
```

Output

```
Welcome to my college
Welcome to my college
Welcome to my college
Welcome to my college
Welcome to my college
Welcome to my college
Welcome to my college
Welcome to my college
Welcome to my college
Welcome to my college
```

The output of both versions of above program is the same. But in first method, we are not using loop so that we have to write `printf()` statement 10 times whereas in second method,the same function or job is being done by writing the `printf()` statement once which is being repeated for 10 times itself with the help of a of loop. Here, same job (i.e. job of displaying the message) is to be repeated for ten times. In this type of situation, a loop seems to be more useful. Thus, second method is more efficient than the first.

Types of Loop

i. `for` loop
ii. `while` loop
iii. `do–while` loop

6.2.1 `for` Loop

The `for` loop allows to execute block of statements for a number of times. When the number of repetitions is known in advance, the use of this loop will be more efficient. Thus, this loop is also known as a `determinate` or `definite` loop.

Syntax:

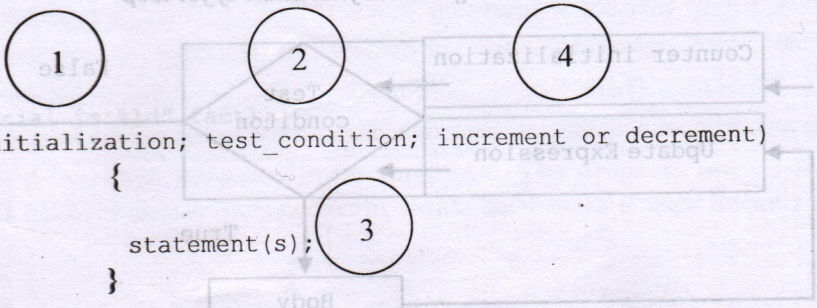
```
for(counter_initialization; test_condition; increment or decrement)
 {
  statements; or body of  loop
 }
```

Different components in for loop

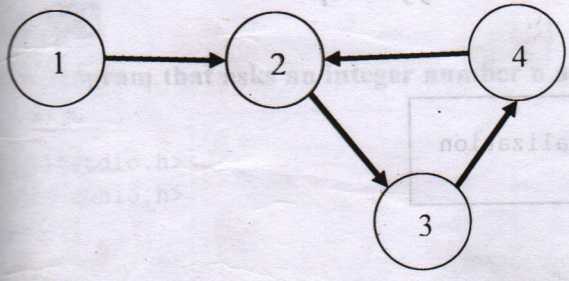
Counter initialization: The counter variable is initialized using assignment statement such as `i=1` , `j=10` and `count=0`. Here, the variable `i`, `j` and `count` are known as `counter or loop-controlled` variables whose value controls the loop execution. The initialization of counter variable is done only once prior to execution of the statements within the `for` loop.

Test condition: The value of `counter` variable is tested using test condition. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution continues with the statement that immediately follows the loop.

Update expression or increment/ decrement expression: When the body of the loop is executed, the control is transferred back to the `for` statement after evaluating the last statement in the body of loop. The counter variable is updated either incremented or decremented and then only it is to be tested with test condition again and same process is repeated until the condition fails. Working rule for this loop can be illustrated from following figure:



The different components in **for** loop are executed in directed sequence as shown figure.



At first; counter is initialized to some value (i.e. step 1). Then counter variable is tested with test condition (i.e. step 2). If test is true, body of loop is executed (i.e. step 3). After finishing body, the counter variable is incremented or decremented (i.e. step 4) and then again updated counter variable is tested with test condition (i.e. step 2). The same process is repeated as long as the condition is true. If the result with test condition is false, the control passes outside the loop i.e. the statement following the loop.

Flowchart

The `for` loop can be represented with following different methods in flow chart.

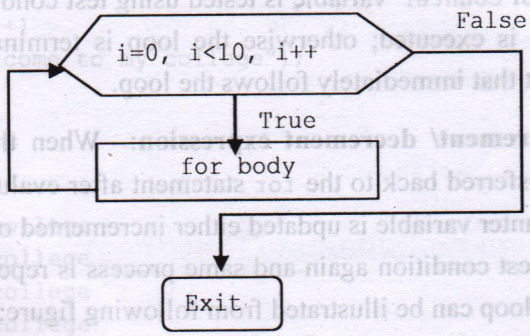


Fig. 6.2.1-a flow chart of for loop

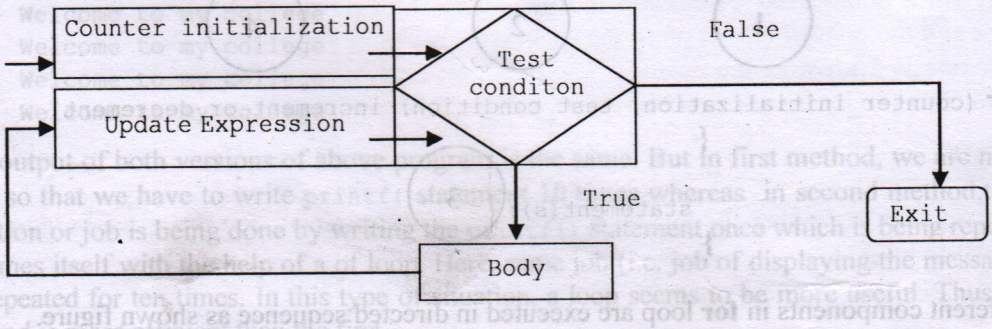


Fig 6.2.1-b: Flowchart of for loop

The next version of the flow chart is

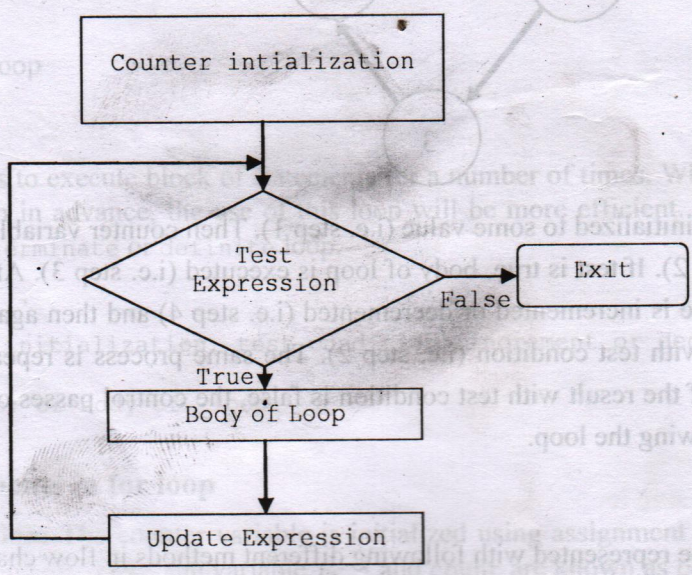


Fig. 6.1.1-c: flow chart of the for loop

Example 6.7

Write a program to calculate factorial of a number.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int num,i;
long fact=1;
printf("\nEnter a number whose factorial is to be calculated:");
scanf("%d",&num);
for(i=1;i<=num;i++)
  fact*=i;
printf("\nThe factorial is:%ld",fact);
getch();
return 0;
}
```

Output

```
Enter a number whose factorial is to be calculated:5
The factorial is:120
```

Example 6.8

Write a program that asks an integer number n and calculate sum of all natural numbers from 1 to n.

```
#include<stdio.h>
#include<conio.h>
int main()
{
 int num,i,sum=0;
 printf("\nEnter a number n \t");
 scanf("%d",&num);
 for(i=1;i<=num;i++)
    sum+=i;
 printf("\nThe sum is:%d",sum);
 getch();
 return 0;
}
```

Output

```
Enter a number n: 10
The sum is:55
```

6.2.2 while Loop

Syntax	Example

`while(test_condition)`
`{`
`body of loop`
`}`
	`int i=0;`
	`while(i<10)`
	`{`
	`printf("%d ",i);`
	`i++;`
	`}`

The `test_condition` is evaluated first and if the condition is true, then the body of loop is executed. After execution of body of loop once, the `test_condition` is again evaluated to determine the body of loop is to be executed for next time or not. If `test_condition` produces true value, the body is executed once again. This process of repeated execution of the body continues until the `test_condition` finally becomes false and then the control is transferred out of the loop. On exit, the program continues with the statement immediately after the body of the loop.

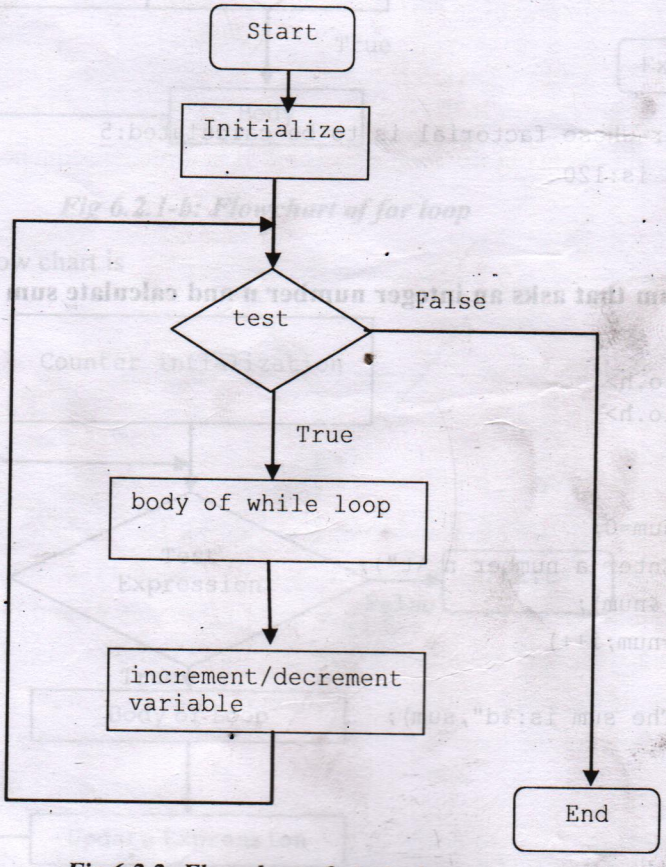


Fig 6.2.2: Flow chart of `while` *loop*

Comparison of `for` and `while` loop:

When frequency of repetition is known in advance, the `for` loop is used and when it is not known, the `while` loop is suitable. An equivalent while loop can be written using `for` loop like:

```
for(i=0;i<10;i++)

{
//for loop body
}
```

The equivalent `while` loop is

```
i=0;
while(i<10)

{
//while body
i++;
}
```

Example 6.9

Write a program to add two numbers and display their sum. The program must ask next two numbers and add till user wants. (i.e. program would terminate if user doesn't want to add another numbers).

```
#include<stdio.h>
#include<conio.h>
int main()
{
int a,b,sum;
char nextTime;
nextTime='y';
while(nextTime=='y')

{
printf("\nEnter two number to be added:\t");
scanf("%d%d",&a,&b);
sum=a+b;
printf("\nThe sum is:\t%d",sum);
printf("\n\nDo you want to add another two numbers?");
printf("\nPress y for yes and other characters for exit\t");
scanf(" %c",&nextTime);
 /* Be careful blank space before %c */

}
getch();
return 0;
}
```

Output

```
Enter two numbers to be added:   50 20
The sum is:      70
Do you want to add another two numbers?
Press y for yes and other characters for exit   y
Enter two number to be added:   80 70
The sum is:      150
Do you want to add another two numbers?
Press y for yes and other characters for exit   n
```

Example 6.10

Write a program to calculate factorial of a number using while loop.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int num,i=1;
long fact=1;
printf("Enter a number whose factorial is to be calculated:");
scanf("%d",&num);
while(i<=num)
  {
   fact*=i;
   i++;
  }
printf("The factorial is:%ld",fact);
getch();
return 0;
}
```

Point to remember:

All `for` loop can be converted into `while` loop. But it is not necessary to convert all `while` loop into `for` loop efficiently.

6.2.3 `do-while` loop

Syntax	Example

Do
{
//statements or body of loop;
}while(test_condition);
	char ch;
	do
	{
	ch=getch();
	printf("\nyou typed %c",ch);
	printf("do you want to exit?y/n");
	}while(ch!='y');

`do-while` loop executes a body (i.e. block of statements) first without checking any condition and then checks a `test condition` to determine whether the body of loop is to be executed for next time or not. Hence `do-while` loop executes its body at least once even if the condition is fail at all. If the condition is true for next time, the program continues to evaluate the body of the loop again and again. This process continues as long as the condition is true.

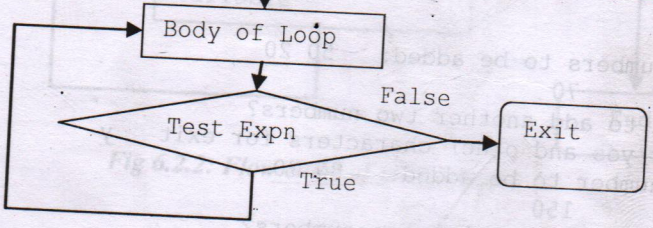


Fig 6.2.3: Flowchart of `do--while` loop

Difference between `while` loop and `do-while` loop

While	do—while
1) `while` loop is `entry-controlled` loop i.e. test condition is evaluated first and body of loop is executed only if the test is true. 2) The body of the loop may not be executed at all if the condition is not satisfied at the very first attempt.	1) `do-while` loop is `exit-controlled` loop i.e. the body of the loop is executed first without checking any condition and at the end of body of loop, the condition is evaluated for repeation of next time 2) The body of loop is always executed at least once.

Example 6.11

Write a program to read a number from keyboard until a zero or a negative number is keyed in. Finally, calculate sum and average of entered numbers.

```
#include<stdio.h>
#include<conio.h>
int main()

 int num,count=0;
 float sum=0,avg;
 do
  {
  printf("\nEnter number:\t");
  scanf("%d",&num);
  sum+=num;
  count++;
  }while(num>0);
 sum=sum-num;
 avg=(sum)/(count-1);
/* here the last number either 0 or negative number is not included in
average. Thus, the final number entered is excluded from the sum and hence
sum-num & count-1*/
printf("\nThe sum is:\t%d",sum);
printf("\nThe average is:\t%f",avg);
getch();
return 0;
```

Output

```
Enter number:   10
Enter number:   20
Enter number:   15
Enter number:   5
Enter number:   0
The sum is:     50.000000
The average is: 12.500000
```

Explanation: Here `do.. while` loop is used since we have to enter a number first and only then we can check whether it is zero or –ve.

6.2.4 Nested Loop

When body part of the loop contains another loop then the inner loop is said to be nested within the outer loop. There is no limit of the number of loops that can be nested within another loop. The outer loop may be one type (i.e. one of `for`, `while` and `do..while` loop) and inner loop may be same type or any other type. Thus, `for` loop may nest another `for` loop as well as `while` & `do..while` loop. In the case of nested loop, for each value or pass of outer loop, inner loop is completely executed. Thus, inner loop operates/moves fast and outer loop operates slowly. The nested loop is analogous to hour and minute hand of clock. The minute hand rotates quickly and hour hand rotates slowly. For each change in hour hand, minute hand rotates a complete round. Similarly, in nested loop, the inner loop executes for its total possible repeatitions for single pass of outer loop. For example, we can write syntax of nested for loop as

```
for(counter_initialization; test_condition;increment/
                decrement)
    {
   for(ctr initialization; test;incr/decrement)
        {
          statements;
        }
}
```

Inner loop

Outer loop

Fig. 6.2.4 nested `for` loop

Example 6.12

Write a program to display multiplication table of 5-10.

```
#include<stdio.h>
#include<conio.h>
int main()
 {
  int i,j;
  for(i=5;i<=10;i++)
    {
     for(j=1;j<=10;j++)
       {
         printf("%d*%d=%d\t",i,j,i*j);
       }
     printf("\n");
     }
getch();
return 0;
}
```

Output

```
5*1=5   5*2=10   5*3=15   5*4=20   5*5=25   5*6=30   5*7=35   5*8=40   5*9=45   5*10=50
6*1=6   6*2=12   6*3=18   6*4=24   6*5=30   6*6=36   6*7=42   6*8=48   6*9=54   6*10=60
7*1=7   7*2=14   7*3=21   7*4=28   7*5=35   7*6=42   7*7=49   7*8=56   7*9=63   7*10=70
8*1=8   8*2=16   8*3=24   8*4=32   8*5=40   8*6=48   8*7=56   8*8=64   8*9=72   8*10=80
9*1=9   9*2=18   9*3=27   9*4=36   9*5=45   9*6=54   9*7=63   9*8=72   9*9=81   9*10=90
10*1=10 10*2=20 10*3=30 10*4=40 10*5=50 10*6=60 10*7=70 10*8=80 10*9=90 10*10=100
```

6.3 break and continue STATEMENT

6.3.1 break statement

The break statement terminates the execution of the loop and the control is transferred to the statement immediately following the loop. Generally, the loop is terminated when its test condition becomes false. But if the loop is to be terminated instantly without testing termination condition, the break statement is useful.

The syntax for break statement is: **break;**

The break statement is also used in switch statement which causes a transfer of control out of the entire switch statement, to the first statement following the switch statement. The switch statement will be discussed later.

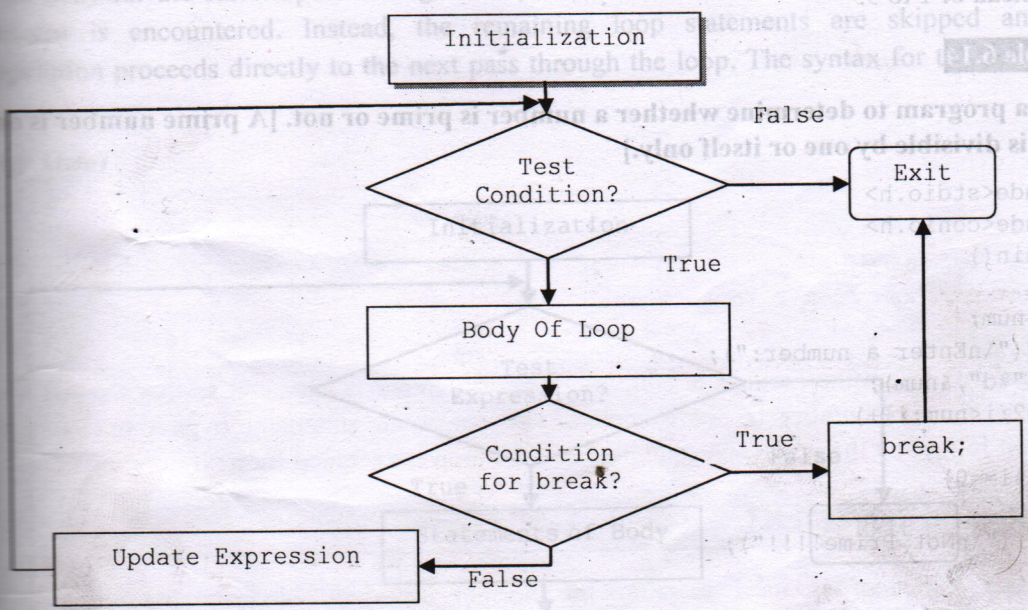


Fig. 6.3.1 Flow chart of break *statement with* for *loop*

Example 6.13

What is the output of following program?

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i;
for(i=1;i<10;i++)
  {
```

```
    printf("\t%d",i);
    if(i==5)
      break;
    }
  getch();
  }
```

Output

1 2 3 4 5

Explanation: The counter variable `i` is initialized to `1` and the test condition is `i<10`. Thus, the loop body is to be executed `9` times (i.e. `1 to 9`). But the use of `break` statement within body of loop has terminated the loop when the value of `i` becomes `5`. Thus, its output is only numbers `1` to `5` instead of `1` to `9`.

Example 6.14

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

```
#include<stdio.h>
#include<conio.h>
int main()
{
int i,num;
printf("\nEnter a number:");
scanf("%d",&num);
for(i=2;i<num;i++)
{
if(num%i==0)
 {
 printf("\nNot Prime!!!!");
 break;
 }
}
if(i==num)
 {
 printf("\nPrime Number!!!");
 }
getch();
return 0;
}
```

Output

a) Enter a number:23
   Prime Number!!!

b) Enter a number:40
   Not Prime!!!!

Note:

To determine a number for prime or not, we have to divide the number by 2, 3, 4,.....n-1. If the number is not divisible by 2, then we divide it by 3. Again, if it is not divisible by 3, then divide it by 4. This process is continued until the divisor is one less than the number (i.e.n-1). If the number is divisible by a one number, then it is not necessary to execute loop for next time (i.e. if it is divisible by 2, it is not necessary to divide the number by 3,4,5... It is known that the number is not prime) and in such situation, we use break to terminate from the loop.

6.3.2 continue statement

The continue statement is used when we want to continue running the loop, but stop processing the remainder of the code in its body for this particular iteration. It bypasses the remainder code of the body for the current pass through a loop. The loop does not terminate when a continue statement is encountered. Instead, the remaining loop statements are skipped and the computation proceeds directly to the next pass through the loop. The syntax for this statement is:

```
continue;
```

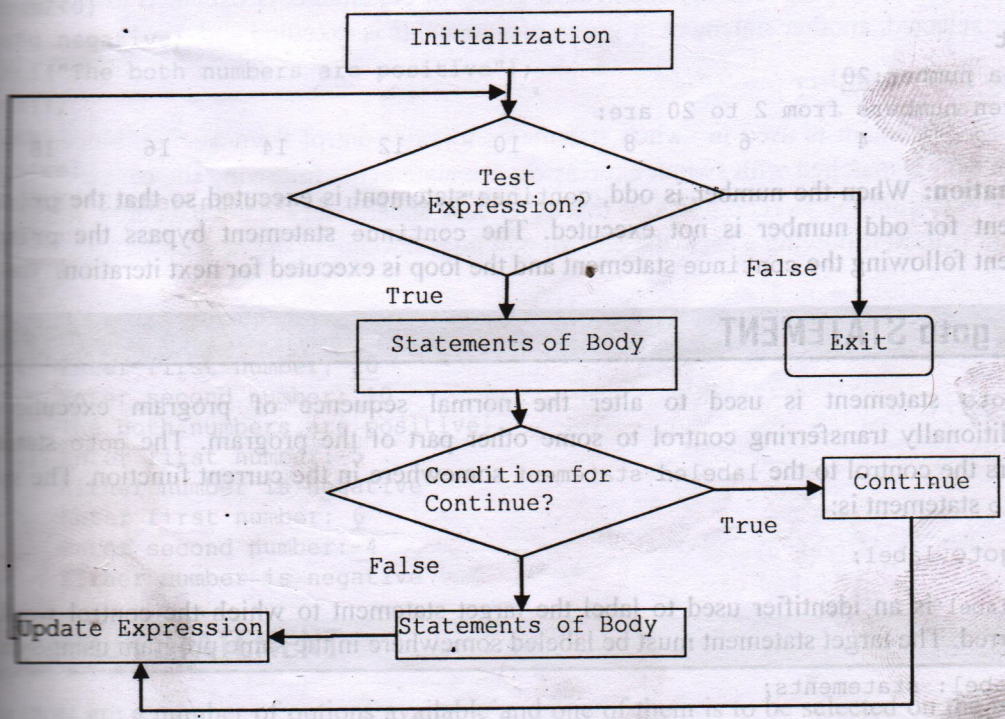


Fig. 6.3.2: Flowchart of continue *statement with* for *loop*

Example 6.15

Write a program that asks for a number n from user and then display only even numbers from 2 to n. [use continue statement].

```
#include<stdio.h>
#include<conio.h>
int main()
    {
    int i,num;
    printf("\nEnter a number:");
    scanf("%d",&num);
    printf("\nThe even numbers from 2 to %d are:\n",num);
    for(i=1;i<=num;i++)
        {
        if(i%2!=0)
            continue;
        printf("\t%d",i);
        }
    getch();
    return 0;
    }
```

Output

```
Enter a number:20
The even numbers from 2 to 20 are:
    2       4       6       8       10      12      14      16.     18      20
```

Explanation: When the number is odd, `continue` statement is executed so that the `printf()` statement for odd number is not executed. The `continue` statement bypass the `printf()` statement following the `continue` statement and the loop is executed for next iteration.

6.4 goto STATEMENT

The `goto` statement is used to alter the normal sequence of program execution by unconditionally transferring control to some other part of the program. The `goto` statement transfers the control to the `labeled statement` somewhere in the current function. The syntax for `goto` statement is:

```
goto label;
```

Here, `label` is an identifier used to label the target statement to which the control would be transferred. The target statement must be labeled somewhere in the same program using syntax:

```
label: statements;
```

Generally, the use of `goto` statement is avoided as it makes program illegible. The use of `goto` statement makes the code very hard to read. This can be an annoyance when trying to understand code written by other people using `goto` construct. However, the `goto` statement can be used in following situations:

i. To make branching around statements or group of statements under certain conditions.

ii. To Jump to the end of a loop under certain conditions, thus bypassing the remainder of the loop during current pass.

iii. To jump completely out of the loop under certain conditions, terminating the execution of a loop.

Example 6.16

Write a program to ask two numbers. Display message "Either number is negative" if either number is negative; otherwise display message "Both numbers are positive".

```
int main()

{
int i,num1,num2;
printf("Enter first number:");
scanf("%d",&num1);
if(num1<0)
  goto negative;
printf("Enter second number:");
scanf("%d",&num2);
if(num2<0)
  goto negative;
printf("The both numbers are positive");
getch();
return;
negative:
  printf("Either number is negative");
getch();
return 0;
}
```

Output

a) Enter first number: 20
   Enter second number: 10
   The both numbers are positive

b) Enter first number:-5
   Either number is negative

c) Enter first number: 6
   Enter second number:-4
   Either number is negative

6.5 switch STATEMENT

When there are a number of options available and one of them is to be selected on the basis of some criteria, `switch` statement is used. Thus, a `switch` statement allows user to choose a statement (or a group of statements) among several alternatives. The `switch` statement compares a variable or expression with different constants (i.e. cases). If it is equal to a case constant, a set of statements following the constant are to be executed. The constants in case statement may be either `char` or `int` type only.

```
switch(variable or expression)
    {
    case caseConstant1:
            statements;
            break;
    case caseConstant2:
            statements;
            break;
    .
    .
    default:
            statements;
}
```

The `switch` statement can be implemented using `if-else if---else` construct also. This construct is generally used to make menu where there are many options in the list. When an option is selected, a particular statement or a group of statements is executed and when next option is selected, another statement or group of statements is executed and so on.

Point to be remember

The `break` statement is used in switch statements to exit control from `switch` block. When a case constant is matched with switch expression, the statements following the case are executed and then exit from the `switch` block so that other case statements are not executed. If no `break` statement is used following a case, the control will goes through to the next cases also.

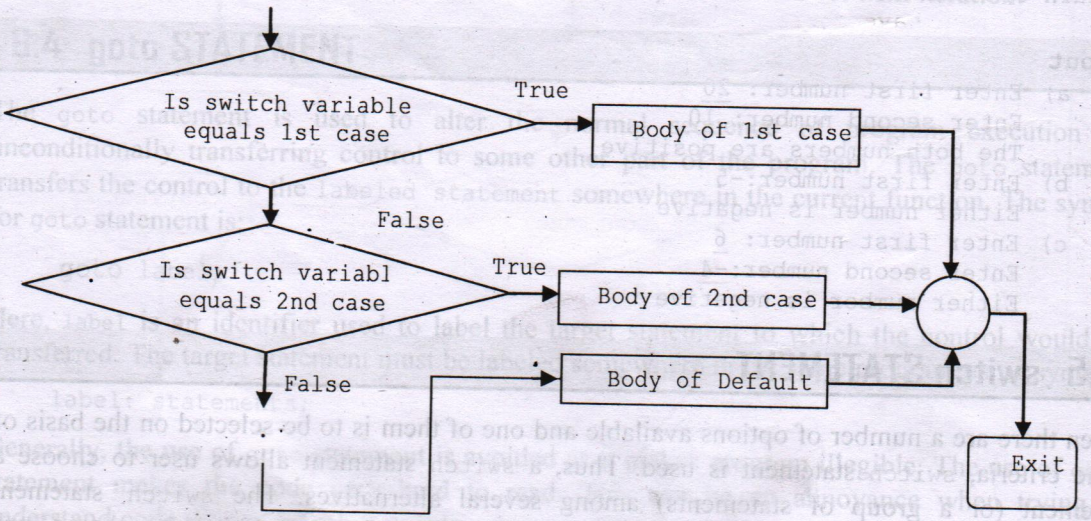


Fig. 6.5: Flowchart of `switch` statement

Example 6.17

Write a program to make a menu with following options:

1 ➔ File

2 ➔ Edit

3 ➔ Save

```
#include<stdio.h>
#include<conio.h>
int main()
{
int choice;
printf("\nselect 1 for File, 2 for Edit and 3 for Save\n");
printf("\n1==>File\n2==>Edit\n3==>Save\n");
scanf("%d",&choice);
switch(choice) //match the case with choice
{
case 1:
  printf("\nYou have chosen File menu item");
  break;
case 2:
   printf("\nYou have chosen Edit menu item");
   break;
case 3:
  printf("\nYou have chosen Save menu item");
  break;
default:
   printf("\nInvalid option!!!");
}
getch();
return 0;
}
```

Output

```
Select 1 for File, 2 for Edit and 3 for Save
1==>File
2==>Edit
3==>Save

You have chosen Edit menu item
```

Example 6.18

Write a program that asks two operands and an operator among +, -, * and /. Depending upon entered operator, evaluate two operands and display the result. Also draw a flowchart.[use switch statement]

```
#include<stdio.h>
#include<conio.h>
int main()
      {
      float op1,op2,result;
      char opr;
      printf("Enter two operands:\t");
      scanf("%f%f",&op1,&op2);
      printf("\nEnter one operator among +,-,* and /:\t");
      scanf(" %c",&opr);
      switch(opr)
      {
      case '+':
        result=op1+op2;
        printf("\nThe sum is %f",result);
        break;
      case '-':
         result=op1-op2;
         printf("\nThe difference is %f",result);
         break;
      case '*':
        result=op1*op2;
        printf("\nThe product is %f",result);
        break;
      case '/':
        result=op1/op2;
        printf("\nThe quotient is %f",result);
        break;
      default:
         printf("\nInvalid operator!!!");
      }
     getch();
     return 0;
     }
```

Output

a) Enter two operands: 40 5
   Enter one operator among +,-,* and /: /
   The quotient is 8.000000

b) Enter two operands: 89 44
   Enter one operator among +,-,* and /: %
   Invalid operator!!!