DECISION MAKING AND LOOPING

11.1 INTRODUCTION

Loops provide a way to repeat commands and control how many times they are repeated. Looping is repeating a set of instructions until a specific condition is met. If the condition is never met due to some inherent characteristics of the loop, infinite loop occurs. A loop is defined as a block of statements which are repeatedly executed for a certain number of times.

A program loop consists of two segments, one known as the body of the loop and the other known as the control statement. The control statement tests certain conditions and then directs the repeated execution of the statements contained in the body of the loop.

Depending on the position of the control statement in the loop, a control structure may be classified either as the entry-controlled loop or as the exit-controlled loop. The entry-controlled and exit-controlled loops are also known as pre-test and post-test loops respectively. In the entry-controlled loop, the control conditions are tested before the start of the loop execution. If the conditions are not satisfied then the

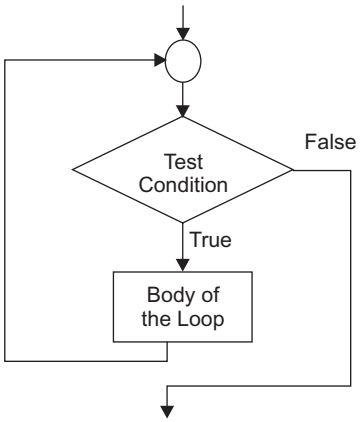

Fig. 11.1 *Entry-controlled loop*

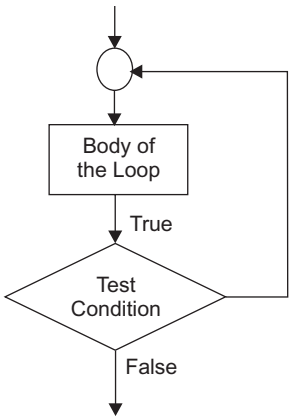

Fig. 11.2 *Exit-controlled loop*

body of the loop will not be executed. In the 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 includes the following steps:

1. Initialization of a condition variable.
2. Execution of the statements in the loop.
3. Test for a specified value of the condition variable for execution of the loop.
4. Incrementing or updating the condition variable.

C language gives us a choice of three types of loop:

1. The while..do statement
2. do…while statement
3. for statement

11.2 WHILE LOOP

The while loop is an entry controlled loop. The test condition is evaluated, and if it is true, the body of loop is executed. The process is repeated until the condition becomes false.

Syntax

```
while (test condition)
{
……..
body of the  loop
……
}
```

The body of the loop may have one or more statements. The braces are needed only if the body contains two or more statements.

```
int i=1;
while (i<=3)
{
printf("C Program");
printf("\n");
i++;
#
i++;
}
#
```

When the while statement is executed for the first time, the value of i is set to an initial value 1. Now the condition i <= 3 is tested. Since i is 1, the condition is satisfied and the body of the loop is executed for the first time.

After the statements are executed, the increment is done within the braces of the while and the value of i is incremented (i.e., i++ means I = i+1). Again the test is performed to check whether the new value of i exceeds 3. If the value of i is still within the range 1 to 3, the statements within the braces of while are executed again. The body of the while loop continues to get executed till i doesn't exceed the final value 3.

When i reaches the value 4, the control exits from the loop.

EXAMPLE PROGRAM

```
#include<stdio.h>
main()
{
int n=10,i=1,sum=0;
while(i<=n)
{
sum=sum+i;
i=i+1;
}
printf("Sum of ten numbers…..%d",sum);}
```

Output

```
Sum of ten numbers……55
```

11.3 DO…WHILE LOOP

The **while** loop construct tests the condition before the loop is executed. Therefore, the body of the loop may not be executed at all if the condition is not satisfied at the very first attempt. But during some situations, it might be necessary to execute the body of the loop before the test is performed. Such situations can be handled with the help of the **do** statement.

On reaching the **do** statement, the program proceeds to evaluate the body of the loop first. At the end of the loop, the test-condition in the **while** statement is evaluated. If the condition is true, the program continues to evaluate the body of the loop once again. This process continues as long as the test condition is true. Once the condition becomes false, the loop is terminated and the control goes to the statement that appears immediately after the **while** statement.

Since the test condition is evaluated at the bottom of the loop, the **do…while** construct provides an exit-controlled loop and therefore the body of the loop is always executed at least once.

Syntax

```
do
{
   ……
body of the loop
}while( test-condition);
```

EXAMPLE PROGRAM FOR DO..WHILE LOOP

```
#include<stdio.h>
main()
{
int n=10,i=1,sum=0;
do
{
sum=sum+i;
i=i+1;
}while(i<=n);

printf("Sum of ten numbers…..%d",sum);
}
```

Output

```
Sum of ten numbers……55
```

11.4 THE FOR LOOP

The **for** loop is the most commonly used loop statement. It is a pre-test loop and it is used when the number of iterations of the loop is known before the loop is entered. The head of the loop consists of three parts (initialization expression, test expression and increment/decrement expression) separated by semicolons.

Syntax

for(initialization expression, test expression , increment/decrement expression)
{
body of the loop;
}

For loop has three parts:

1. **Initialize:** It is used for initializing a counter variable.
2. **Test condition:** It is used for test the condition.
3. **Increment/decrement:** It is used for increment/decrement the counter variable.

```
for (i = 0; i < 10; i++)
{
     printf ("Hello\n");
     printf ("World\n");
}
```

Let us look at the **for loop** from the example. We first start by setting the variable i to 0. This is where we start to count. Then we say that the **for loop** must run if the counter i is smaller than ten. Last we say that every cycle i must be increased by one (i++).

In the example, we used i++ which is same as using i = i + 1. This is called incrementing. The instruction i++ adds 1 to i. If you want to subtract 1 from i you can use i--. It is also possible to use ++i or --i. The difference is that, with ++i (prefix incrementing) the one is added before the "for loop" tests if i < 10. With i++ (postfix incrementing) the one is added after the test i < 10. In case of **for loop**, this makes no difference, but in **while loop** test, it makes a difference. Before we look at a postfix and prefix increment while loop example, we first look at the while loop.

EXAMPLE PROGRAM

```
#include<stdio.h>
main()
{
int n=10,i,sum=0;
for(i=1;i<=n;i++)
sum=sum+i;
printf("Sum of ten numbers…..%d",sum);
}
```

Output

```
Sum of ten numbers……55
```

11.5 NESTED FOR LOOPS

A **for** statement within another **for** statement is called **nested for loops**. In nested for loops two or more **for** statements are included in the body of the loop.

EXAMPLE PROGRAM

```
#include<stdio.h>
main()
{
int i,j;
for ( i=1 ; i<=3 ;i++)
{
for( j=1; j<=3; j++)
{
printf("i=%d , j=%d\t",i,j);
}
printf("\n");
}
}
```

Output

```
i=1 , j=1        i=1 , j=2              i=1 , j=3
i=2 , j=1        i=2 , j=2              i=2 , j=3
i=3 , j=1        i=3 , j=2              i=3 , j=3
```

Now what we have done here is make an outer loop and an inner loop. For each outer loop, the inner loop will execute three times. Let us make some points to make it clear. For the first time, the value of i (i.e., the looping variable of the outer loop) is 1, the condition is tested if satisfied the test so the body of the outer is executed. When the inner **for** statement is executed for the first time, the value of j is set to an initial value 1.

Now the condition j <= 3 is tested. Since j is 1, the condition is satisfied and the body of the inner loop is executed for the first time and value of i and j is printed. It is used as tab between values. Upon reaching the closing braces of inner **for**, control is sent back to the inner **for** statement, where the value of j gets incremented by 1(i.e., j++ means j = j+1).

Again the test is performed to check whether the new value of j exceeds 3. If the value of j is still within the range 1 to 3, the statements within the braces of inner for are executed again and the value of i and j are printed. The body of the inner **for** loop continues to get executed till j doesn't exceed the final value 3. When j reaches the value 4, the control exits from the inner loop and move to next statement after the closing braces of the **for** inner loop and a new line is printed (due to printf("\n");).

Similarly upon reaching the closing brace of outer **for**, control is sent back to the outer for statement, where the value of i gets incremented by 1(i.e., i++ means I = i+1). The body of the outer **for** loop continues to get executed till j doesn't exceed the final value 3.When i reach the value 4, the control exits from the outer loop and the program ends.

11.6 JUMPING STATEMENTS

11.6.1 Break Statement

The break statement allows you to exit a loop from any point within its body, bypassing its normal termination expression. When the break statement is encountered inside a loop, the loop is immediately terminated, and program control resumes at the next statement following the loop. The break statement can be used with all three of C's loops. You can have as many statements within a loop as you desire. It is generally best to use the break for special purposes, not as normal loop exit. Break is also used in conjunction with functions and case statements which will be covered in later sections.

EXAMPLE PROGRAM

```
int main()
{
    float num,average,sum;
    int i,n;
    printf("Maximum no. of inputs\n");
    scanf("%d",&n);
    for(i=1;i<=n;++i)
    {
         printf("Enter n%d: ",i);
         scanf("%f",&num);
```

```
        if(num<0.0)
        break;                    //for loop breaks if num<0.0
        sum=sum+num;
    }
  average=sum/(i-1);
  printf("Average=%.2f",average);
  return 0;
}
```

Output

```
Maximum no. of inputs
4
Enter n1: 1.5
Enter n2: 12.5
Enter n3: 7.2
Enter n4: -1
Average = 7.07
```

11.6.2 Continue Statement

The continue statement is somewhat the opposite of the break statement. It forces the next iteration of the loop to take place, skipping any code in between itself and the test condition of the loop. In while and do-while loops, a continue statement will cause control to go directly to the test condition and then continue the looping process. In the case of the **for** loop, the increment part of the loop continues. One good use of continue is to restart a statement sequence when an error occurs.

EXAMPLE PROGRAM

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

    /* do loop execution */
    do
    {
        if( a == 15)
        {
            /* skip the iteration */
            a = a + 1;
            continue;
        }
        printf("value of a: %d\n", a);
        a++;

    }while( a < 20 );
```

```
    return 0;
}
```

Output

```
Value of a: 10
Value of a: 11
Value of a: 12
Value of a: 13
Value of a: 14
Value of a: 16
Value of a: 17
Value of a: 18
Value of a: 19
```