Chapter 7

Decision Making and Looping

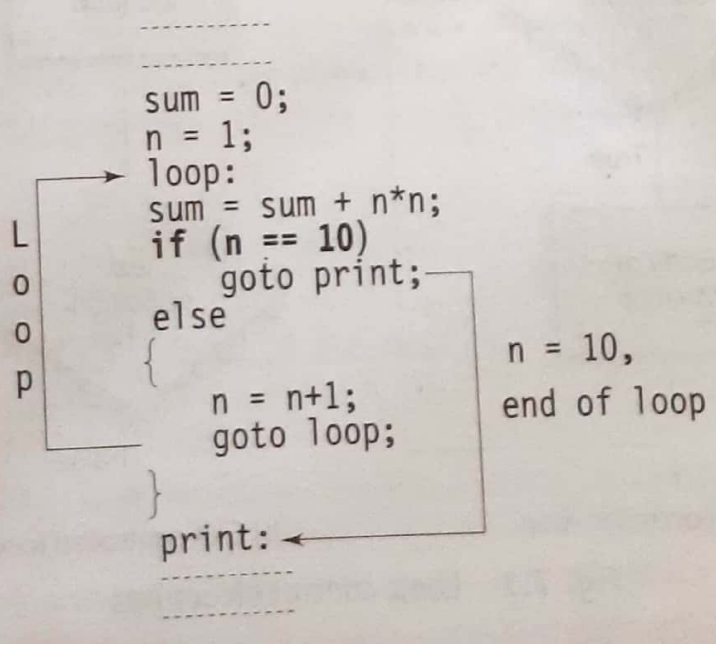
Learning Objectives

Upon completing this chapter, you will be able to:

- **LO 7.1** Discuss while statement
- **LO 7.2** Explain do statement
- **LO 7.3** Describe for statement
- **LO 7.4** Illustrate how jumps are applied in loops

Introduction

We have seen in the previous chapter that it is possible to execute a segment of a program repeatedly introducing a counter and later testing it using the **if** statement. While this method is quite satisfactory for practical purposes, we need to initialize and increment a counter and test its value at an appropriate place the program for the completion of the loop. For example, suppose we want to calculate the sum of squa of all integers between 1 and 10, we can write a program using the **if** statement as follows:



A looping process, in general, would include the following four steps:

1. Setting and 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.

The test may be either to determine whether the loop has been repeated the specifie . of times o to determine whether a particular condition has been met.

The C language provides for three *constructs* for performing *loop* operations. They are:

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

We shall discuss the features and applications of each of these statements in this chapter.

Sentinel Loops

Based on the nature of control variable and the kind of value assigned to it for testing the control expressi the loops may be classified into following two general categories:

1. Counter-controlled loops
2. Sentinel-controlled loops

When we know in advance exactly how many times the loop will be executed, we use a *counter-contro loop*. We use a control variable known *as counter*. The counter must be initialized, tested and updated prop for the desired loop operations. The number of times we want to execute the loop may be a constant variable that is assigned a value. A counter-controlled loop is sometimes called *definite repetition loop*.

In a *sentinel-controlled loop*, a special value called a *sentinel* value is used to change the loop co expression from true to false. For example, when reading data we may indicate the "end of data" by a sp value, like –1 and 999. The control variable is called **sentinel** variable. A sentinel-controlled loop is often c *indefinite repetition loop* because the number of repetitions is not known before the loop begins executin

The while Statement

LO 7.1
Discuss while stateme

The simplest of all the looping structures in C is the **while** statement. We have used **while** in many of our earlier programs. The basic format of the **while** statement is

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

The **while** is an *entry-controlled* loop statement. The *test-condition* is evaluated and if the condition then the body of the loop is executed. After execution of the body, the test-condition is once again ev and if it is true, the body is executed once again. This process of repeated execution of the body co until the test-condition finally becomes *false* and the control is transferred out of the loop. On exit, the p continues with the statement immediately after the body of the loop.

The body of the loop may have one or more statements. The braces are needed only if the body two or more statements. However, it is a good practice to use braces even if the body has only one sta

We can rewrite the program loop discussed in Section 7.1 as follows:

```
========
sum = 0;                          /* Initialization */
n = 1;
while(n <= 10)                    /* Testing */
{
    sum = sum + n * n;
    n = n+1;                      /* Incrementing */
}
printf("sum = %d\n", sum);
========
```

loop

The body of the loop is executed 10 times for n = 1, 2, ..., 10, each time adding the square of n, which is incremented inside the loop. The test condition may also be written as **n < 11**; the result is the same. This is a typical example of counter-controlled loops. The variable n is called a ***counter variable***.

Another example of **while** statement, which uses the keyboard input is shown below:

```
=========
character = ' ' ;
while (character != 'Y')
        character = getchar();
xxxxxxx;
=========
```

First the **character** is initialized to ' '. The **while** statement then begins by testing whether character is equal to Y. Since the **character** was initialized to ' ', the test is true and the loop statement

```
character = getchar();
```

is executed. Each time a letter is keyed in, the test is carried out and the loop statement is executed until the letter Y is pressed. When Y is pressed, the condition becomes false because **character** equals Y and the loop terminates, thus transferring the control to the statement xxxxxxx;. This is a typical example of counter-controlled loops. The character constant 'y' is called *sentinel* value and the variable **character** is called the sentinel variable, which often referred to as the *sentinel variable*.

WORKED-OUT PROBLEM 7.1

A program to evaluate the equation

$$y = x^n$$

when n is a non-negative integer, is given in Fig. 7.2

The variable **y** is initialized to 1 and then multiplied by **x**, n times using the **while** loop. The control variable **count** is initialized outside the loop and incremented inside the loop. When the value of **count** becomes greater than **n**, the control

Program

```
main()
{
   int count, n;
   float x, y;
   printf("Enter the values of x and n : ");
   scanf("%f %d", &x, &n);
   y = 1.0;
   count = 1;                    /* Initialisation */
   /* LOOP BEGINs */
   while ( count <= n)           /* Testing */
   {
      y = y*x;
      count++;                   /* Incrementing */
   }
   /* END OF LOOP */
   printf("\nx = %f; n = %d; x to power n = %f\n",x,n,y);
   }
```

Output

```
Enter the values of x and n : 2.5 4
x = 2.500000; n = 4; x to power n = 39.062500
Enter the values of x and n : 0.5 4
x = 0.500000; n = 4; x to power n = 0.062500
```

Fig. 7.2 Program to compute **x** to the power n using **while** loop

Test Your Skills

The factorial of an integer m is the product of consecutive integers from 1 to m. That is,
factorial m = m! = m x (m–1) x ..... x 1.
Write a program that computes and prints a table of factorials for any given m. **[M]**

The do Statement

LO 7.2
Explain do statement

The **while** loop construct that we have discussed in the previous section, makes a test of 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. On some occasions it might be necessary to execute the body of the loop before the test performed. Such situations can be handled with the help of the **do** statement. This takes the form:

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

On reaching the **do** statement, the program proceeds to evaluate the body of the loop. The *test-condition* in the **while** statement is evaluated. If the condition is true, it evaluates the body of the loop once again. This process continues as long as the condition becomes false, the loop will be terminated and the control goes to the statement immediately after the **while** statement.

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

A simple example of a **do...while** loop is:

```
        do
        {
            printf ("Input a number\n");
loop        number = getnum ( );
        }
        while (number > 0);
```

This segment of a program reads a number from the keyboard until a zero or a negative number is entered, and assigned to the *sentinel* variable **number**.

The test conditions may have compound relations as well. For instance, the statement

```
while (number > 0 && number < 100);
```

in the above example would cause the loop to be executed as long as the number lies between 0 and 100.

Consider another example:

```
I = 1;                                  /* Initializing */
sum = 0;
do
{
    sum = sum + I;
loop
    I = I+2;                            /* Incrementing */
}
while(sum < 40 || I < 10);              /* Testing */
printf("%d %d\n", I, sum);
```

The loop will be executed as long as one of the two relations is true.

WORKED-OUT PROBLEM 7.2

A program to print the multiplication table from 1 to 12 is given in Fig.

value in the table.

, the inner loop is executed a total of 120 times, each time printing

Program:

```
#define COLMAX 10
#define ROWMAX 12
main()
{
    int row,column, y;
    row = 1;
    printf("     MULTIPLICATION TABLE     \n");
    printf("------------------------------\n");
    do /*......OUTER LOOP BEGINS.....*/
    {
        column = 1;
        do /*.....INNER LOOP BEGINS.....*/
        {
            y = row * column;
            printf("%4d", y);
            column = column + 1;
        }
            while (column <= COLMAX); /*...INNER LOOP ENDS...*/
            printf("\n");
            row = row + 1;
    }
    while (row <= ROWMAX);/*..... OUTER LOOP ENDS .....*/
    printf("------------------------------------\n");
}
```

Output

MULTIPLICATION TABLE

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	18	21	24	27	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	30	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	90
10	20	30	40	50	60	70	80	90	100
11	22	33	44	55	66	77	88	99	110
12	24	36	48	60	72	84	96	108	120

Fig. 7.3 Printing of a multiplication table using **do...while loop**

Simple 'for' Loops

The **for** loop is another entry-controlled loop that provides a more concise loop control structure. The general form of the **for** loop is

```
for ( initialization ; test-condition ; increment or decrement)
{
    body of the loop
}
```

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

1. *Initialization* of the control variables is done first, using assignment statements such as i = 0. The variables i and **count** are known as loop-control variables.
2. The value of the control variable is tested using the test-condition. The test-condition is a relational expression, such as i < 10 or i > 0, that determines when the loop will exit. If the condition is true, the body of the loop is executed; otherwise the loop is terminated and the execution will resume at the statement that immediately follows the loop.
3. When the body of the loop is executed, the control is transferred back to the for statement after evaluating the last statement in the loop. Now, the control variable is incremented or decremented using an assignment statement, such as i = i+1 or i = i – 1, and the new value of the control variable is again tested to see whether it satisfies the loop condition. If the condition is satisfied, the body of the loop is again executed. This process continues till the value of the control variable fails to satisfy the test-condition.

Note C99 enhances the **for** loop by allowing declaration of variables in the initialization permits portion. See the Appendix "C99 Features".

Consider the following segment of a program:

```
                for ( x = 0 ; x <= 9 ; x = x+1)
loop  ------>   {
                    printf("%d", x);
                }
                printf("\n");
```

for loop is executed 10 times and prints the digits 0 to 9 in one line. The three sections enses must be separated by semicolons. Note that there is no semicolon at the end of the x = x+1.

or statement allows for *negative increments*. For example, the loop discussed above can

The program evaluates the value

$$p = 2^n$$

successively by multiplying 2 by itself n times.

$$q = 2^{-n} = \frac{1}{p}$$

Note that we have declared **p** as a ***long int*** and **q** as a **double**.

WORKED-OUT PROBLEM 7.4

E

The program in Fig. 7.5 shows how to write a C program to print nth Fibonacci number.

Program

```
# include <stdio.h>

void main()
{
        int num 1=0, num2=1, n, i, fib;

        printf("\n\nEnter the value of n: ");
        scanf ("%d", &n);
        for (i = 1; i <= n-2; i++)
        {
                fib=num1 + num2;
               num1=num2;
               num2=fib;
      }
      printf("\nnth fibonacci number (for n = %d) = %d, n,fib);
}
```

Fig. 7.5 Program to print nth fibonacci number

WORKED-OUT PROBLEM 7.5

The program in Fig. 7.6 shows how to write a C program to print all the prime numbers between where 'n' is the value supplied by the user.

Program

```
# include <stdio.h>

void main()
```

12 TUESDAY MARCH

Array:-

An array is a fixed size sequenced collection of elements of the same data type. It is simply a grouping of like type data.

In its simplest form, an array can be used to represent a list of numbers, or a list of names.

Some examples in that the concept of array can be used.

* List of temperature recorded every hour in a day or month or a year
* List of employees in an organization
* List of products and their cost sold by a store.
* Test scores of a class of students
* List of customers and their telephone numbers

Types of Arrays:-
One dimensional Arrays ✓
Two dimensional ,,
Multidimensional ,,

Data Types

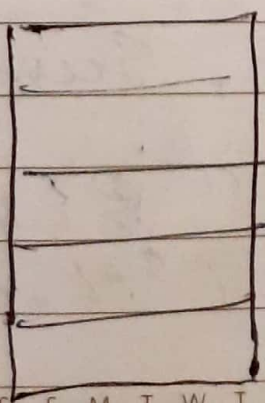
- Derived Types
  - Arrays
  - pointers
- Fundamental Types
  - Integral Types
  - Float ,,
  - character ,,
- User defined types
  - Structs
  - Unions
  - Enumeration

One dimensional Arrays:

To represent a set of five numbers say (35, 40, 20, 57, 19) by an array variable number;

```
int number[5];
```

Five storage locations as



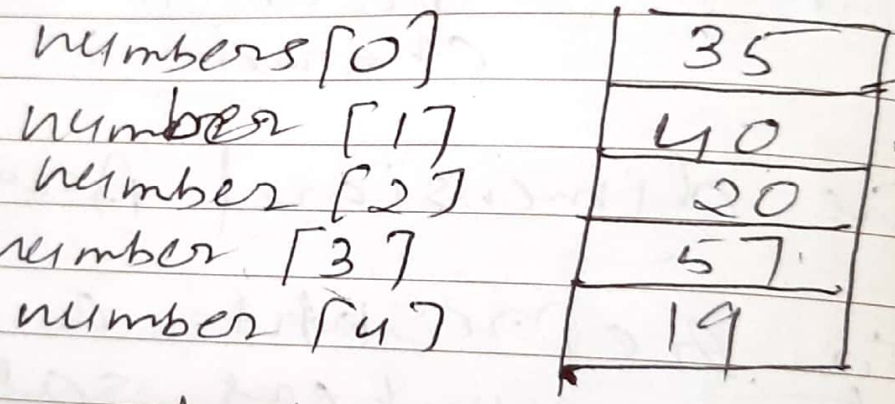
number[0]
number[1]
number[2]
number[3]
number[4]

14 THURSDAY MARCH

The values to the array elements can be assigned as follows:

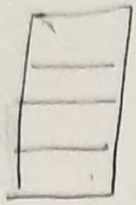
```
number[0] = 35;
number[1] = 40;
number[2] = 20;
number[3] = 57;
number[4] = 19;
```

Array number to store the value as shown:



// T.O Calculate sum of n numbers using an array

```
#include <stdio.h>
void main()
{
  int n, sum=0, i, array
  Pag.
  printf("Enter an integ
  Scanf("%d", &n);
```

n = [i+1] 

FRIDAY 15
MARCH WK 11 (074-291)

```
for ( i=0; i<n ; i++)
  {
   scanf ("%d", &array[i]);
   Sum = Sum + array[i];
  }
printf ("Sum = %d\n", Sum);
}
```

Calculate Average

```
void main()
{ int marks[10], i, n,
        Sum=0, avg;
printf ("Enter no. of elements");
scanf ("%d", &n);
for ( i=0; i<n; i++)
  {
   printf ("Enter no. of %d.",
   scanf ("%d", &marks[i]);
   Sum = Sum + marks[i];
  }
avg = Sum/n;
printf ("Average = %d", avg);
}
```

APRIL MAY JUNE