UNIT 5 DECISION AND LOOP CONTROL STATEMENTS

Structure

5.0 Introduction
5.1 Objectives
5.2 Decision Control Statements
  5.2.1 The *if* Statement
  5.2.2 The *switch* Statement
5.3 Loop Control Statements
  5.3.1 The *while* Loop
  5.3.2 The *do-while* Statement
  5.3.3 The *for* Loop
  5.3.4 The Nested Loop
5.4 The *Goto* Statement
5.5 The *Break* Statement
5.6 The *Continue* Statement
5.7 Summary
5.8 Solutions / Answers
5.9 Further Readings

5.0 INTRODUCTION

A *program* consists of a number of statements to be executed by the computer. Not many of the programs execute all their statements in sequential order from beginning to end as they appear within the program. A *C program* may require that a logical test be carried out at some particular point within the program. One of the several possible actions will be carried out, depending on the outcome of the *logical test*. This is called ***Branching***. In the ***Selection*** process, a set of statements will be selected for execution, among the several sets available. Suppose, if there is a need of a group of statements to be executed repeatedly until some logical condition is satisfied, then ***looping*** is required in the program. These can be carried out using various control statements.

These ***Control statements*** determine the "*flow of control*" in a program and enable us to specify the order in which the various instructions in a program are to be executed by the computer. Normally, high level procedural programming languages require three basic control statements:

- Sequence instruction
- Selection/decision instruction
- Repetition or Loop instruction

Sequence instruction means executing one instruction after another, in the order in which they occur in the source file. This is usually built into the language as a default action, as it is with C. If an instruction is not a control statement, then the next instruction to be executed will simply be the next one in sequence.

Selection means executing different sections of code depending on a specific condition or the value of a variable. This allows a program to take different courses of action depending on different conditions. C provides three selection structures.

- *if*
- *if...else*
- *switch*

Repetition/Looping means executing the same section of code more than once. A section of code may either be executed a fixed number of times, or while some condition is true. C provides three looping statements:

- *while*
- *do…while*
- *for*

This unit introduces you the decision and loop control statements that are available in C programming language along with some of the example programs.

5.1 OBJECTIVES

After going through this unit you will be able to:

- work with different control statements;
- know the appropriate use of the various control statements in programming;
- transfer the control from within the loops;
- use the ***goto***, ***break*** and ***continue*** statements in the programs; and
- write programs using branching, looping statements.

5.2 DECISION CONTROL STATEMENTS

In a C program, a decision causes a one-time jump to a different part of the program, depending on the value of an expression. Decisions in C can be made in several ways. The most important is with the ***if...else*** statement, which chooses between two alternatives. This statement can be used without the ***else***, as a simple ***if*** statement. Another decision control statement, ***switch***, creates branches for multiple alternative sections of code, depending on the value of a single variable.

5.2.1 The *if* Statement

It is used to execute an *instruction* or sequence/*block of instructions* only if a *condition* is fulfilled. In ***if*** statements, expression is evaluated first and then, depending on whether the value of the expression (relation or condition) is "*true"* or "*false"*, it transfers the control to a particular statement or a group of statements.

Different forms of implementation *if*-statement are:

- Simple *if* statement
- *If-else* statement
- *Nested if-else* statement
- *Else if* statement

Simple *if* statement

It is used to execute an instruction or block of instructions only if a condition is fulfilled.

The syntax is as follows:

```
if (condition)
    statement;
```

where c*ondition* is the expression that is to be evaluated. If this *condition* is ***true***, *statement* is executed. If it is ***false***, *statement* is ignored (not executed) and the program continues on the next instruction after the conditional statement.

This is shown in the Figure 5.1 given below:

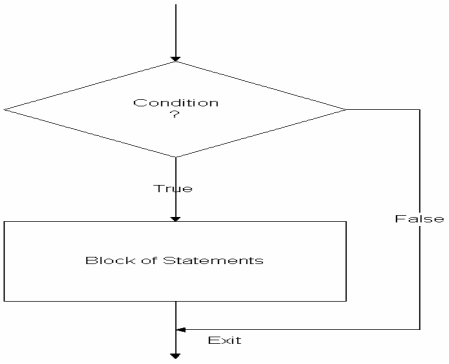


Figure 5.1: Simple *if* statement

If we want more than one statement to be executed, then we can specify a block of statements within the curly bracets { }. The syntax is as follows:

```
if (condition)
   {
     block of statements;
  }
```

Example 5.1

Write a program to calculate the net salary of an employee, if a tax of 15% is levied on his gross-salary if it exceeds Rs. 10,000/- per month.

```
/*Program to calculate the net salary of an employee */

#include <stdio.h>
main( )
{
float gross_salary, net_salary;

printf("Enter gross salary of an employee\n");
scanf("%f ",&gross_salary );

if (gross_salary <10000)
      net_salary= gross_salary;
if (gross_salary >= 10000)
      net_salary = gross_salary- 0.15*gross_salary;

printf("\nNet salary is Rs.%.2f\n", net_salary);
}
```

OUTPUT

```
Enter gross salary of an employee
9000
Net salary is Rs.9000.00

Enter gross salary of any employee
10000
Net salary is Rs. 8500.00
```

If … else statement

If...else statement is used when a different sequence of instructions is to be executed depending on the logical value *(True / False)* of the condition evaluated.

Its form used in conjunction with *if* and the syntax is as follows:

```
if (condition)
        Statement _1;
  else
        Statement_ 2;
statement_3;
```

Or

```
if (condition)
     {
     Statements_1_Block;
     }
else
    {
    Statements_2_Block;
    }
Statements _3_Block;
```

If the *condition* is ***true***, then the sequence of statements (S*tatements_1_Block)* executes; otherwise the *Statements_2_Block* following the *else* part of *if-else* statement will get executed. In both the cases, the control is then transferred to *Statements_3* to follow sequential execution of the program.
This is shown in figure 5.2 given below:

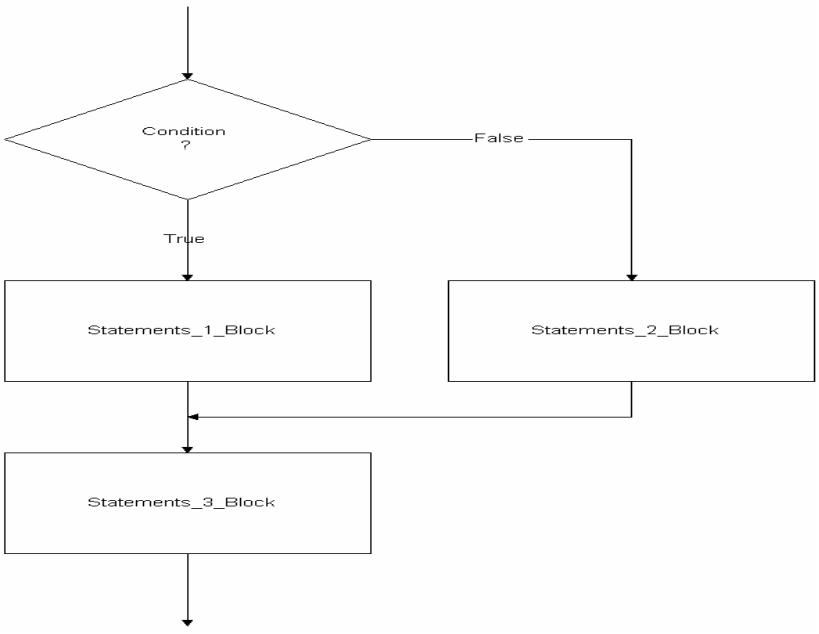


Figure 5.2: *If…else* statement

Let us consider a program to illustrate *if...else* statement,

Example 5.2

Write a program to print whether the given number is even or odd.

```
/* Program to print whether the given number is even or odd*/
#include <stdio.h>
main ( )
{
int x;
printf(“Enter a number:\n”);
scanf("%d",&x);
if (x % 2 == 0)
        printf(“\nGiven number is even\n”);
else
        printf(“\nGiven number is odd\n”);
}
```

OUTPUT

```
Enter a number:
6
Given number is even


Enter a number
7
Given number is odd
```

Nested *if…else* statement

In *nested if… else statement*, an entire *if…else* construct is written within either the body of the ***if*** statement or the body of an *else* statement. The syntax is as follows:

```
if (condition_1)
   {
     if (condition_2)
        {
          Statements_1_Block;
        }

    else
       {
          Statements_2_Block;
       }
   }

else
     {
       Statements_3_Block;
     }
Statement_4_Block;
```

Here, *condition_1* is evaluated. If it is ***false*** then *Statements_3_Block* is executed and is followed by the execution of *Statements_4_Block*, otherwise if *condition_1* is ***true,*** then *condition_2* is evaluated. *Statements_1_Block* is executed when *condition_2* is ***true*** otherwise *Statements_2_Block* is executed and then the control is transferred to *Statements_4_Block*.

This is shown in the figure 5.3 given in the next page:

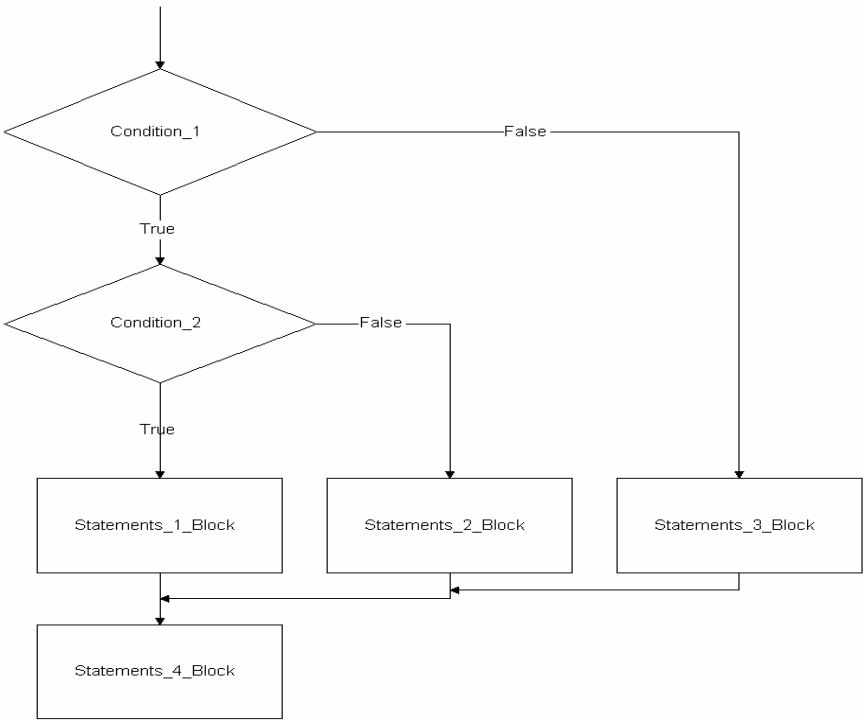


Figure 5.3: Nested *if…else* statement

Let us consider a program to illustrate Nested if…else statement,

Example 5.3

Write a program to calculate an Air ticket fare after discount, given the following conditions:

- If passenger is below 14 years then there is 50% discount on fare
- If passenger is above 50 years then there is 20% discount on fare
- If passenger is above 14 and below 50 then there is 10% discount on fare.

```
/* Program to calculate an Air ticket fare after discount */

#include <stdio.h>
main( )
{
int age;
float fare;
printf("\n Enter the age of passenger:\n");
scanf("%d",&age);
printf("\n Enter the Air ticket fare\n");
scanf("%f",&fare);
if (age < 14)
     fare = fare - 0.5 * fare;
else
     if (age <= 50)
       {
          fare = fare - 0.1 * fare;
       }
          else
       {
          fare = fare - 0.2 * fare;
       }
printf("\n Air ticket fare to be charged after discount is %.2f",fare);
}
```

OUTPUT

```
Enter the age of passenger
12
Enter the Air ticket fare
2000.00
Air ticket fare to be charged after discount is 1000.00
```

Else if statement

To show a multi-way decision based on several conditions, we use the ***else if*** statement. This works by cascading of several comparisons. As soon as one of the conditions is true, the statement or block of statements following them is executed and no further comparisons are performed. The syntax is as follows:

```
if (condition_1)
    {
    Statements_1_Block;
    }
    else if (condition_2)
        {
            Statements_2_Block;
        }
                ------------
                        else if (condition_n)
                           {
                               Statements_n_Block;
                           }
else
    Statements_x;
```

Here, the *conditions* are evaluated in order from top to bottom. As soon as any condition evaluates to *true*, then the statement associated with the given condition is executed and control is transferred to *Statements_x* skipping the rest of the conditions following it. But if all conditions evaluate *false*, then the statement following final ***else*** is executed followed by the execution of *Statements_x*. This is shown in the figure 5.4 given below:

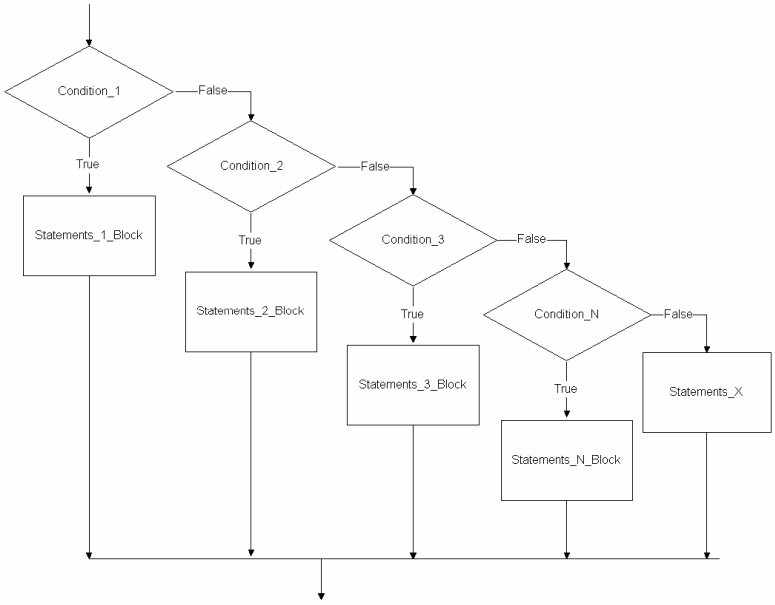


Figure 5.4: *Else if* statement

Let us consider a program to illustrate *Else if* statement,

Example 5.4

Write a program to award grades to students depending upon the criteria mentioned below:

- Marks less than or equal to 50 are given "D" grade
- Marks above 50 but below 60 are given "C" grade
- Marks between 60 to 75 are given "B" grade
- Marks greater than 75 are given "A" grade.

```
/* Program to award grades */
#include <stdio.h>
main()
{
int result;
printf("Enter the total marks of a student:\n");
scanf("%d",&result);
if (result <= 50)
        printf("Grade D\n");
    else if (result <= 60)
            printf("Grade C\n");
        else if (result <= 75)
                printf("Grade B\n");
            else
                printf("Grade A\n");
}
```

OUTPUT

```
Enter the total marks of a student:
80
Grade A
```

Check Your Progress 1

1. Find the output for the following program:

```
#include <stdio.h>
main()
{
        int a=1, b=1;
        if(a==0)
            if(b==0)
                printf("HI");
            else
        printf("Bye");
}
```

……

……

……

2. Find the output for the following program:

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

```
                int a,b=0;
                if (a=b=1)
                        printf("hello");
                else
                        printf("world");
        }
```

……

……

……

5.2.2 The *Switch* Statement

Its objective is to check several possible constant values for an expression, something similar to what we had studied in the earlier sections, with the linking of several ***if*** and ***else if*** statements. When the actions to be taken depending on the value of control variable, are large in number, then the use of control structure ***Nested if…else*** makes the program complex. There ***switch*** statement can be used. Its form is the following:

```
switch (expression){
        case expression 1:
                block of instructions 1
                break;
        case expression 2:
                block of instructions 2
                break;
        .
        .
        default:
                default block of instructions
        }
```

It works in the following way: **switch** evaluates expression and checks if it is equivalent to *expression1*. If it is, it executes *block of instructions 1* until it finds the **break** keyword, moment at finds the control will go to the end of the *switch*. If *expression* was not equal to *expression 1* it will check whether *expression* is equivalent to *expression 2*. If it is, it will execute *block of instructions 2* until it finds the **break** keyword.

Finally, if the value of *expression* has not matched any of the previously specified constants (you may specify as many **case** statements as values you want to check), the program will execute the instructions included in the **default:** section, if it exists, as it is an optional statement.

Let us consider a program to illustrate *Switch* statement,

Example 5.5

Write a program that performs the following, depending upon the choice selected by the user.

i). calculate the square of number if choice is 1
ii). calculate the cube of number if choice is 2 and 4
iii). calculate the cube of the given number if choice is 3
iv). otherwise print the number as it is

```
main()
{
int choice,n;
```

```
printf("\n Enter any number:\n ");
scanf("%d",&n);
printf("Choice is as follows:\n\n");
printf("1. To find square of the number\n");
printf("2. To find square-root of the number\n");
printf("3. To find cube of a number\n");
printf("4. To find the square-root of the number\n\n");
printf("Enter your choice:\n");
scanf("%d",&choice);
switch (choice)
{
        case 1 : printf("The square of the number is %d\n",n*n);
                 break;
        case 2 :
        case 4 : printf("The square-root of the given number is %f",sqrt(n));
                 break;
        case 3:  printf(" The cube of the given number is %d",n*n*n);
        default : printf("The number you had given is %d",n);
                  break;
}
}
```

OUTPUT

```
Enter any number:
 4

Choice is as follows:
1. To find square of the number
2. To find square-root of the number\n");
3. To find cube of a number
4. To find the square-root of the number

Enter your choice:
2
The square-root of the given number is 2
```

In this section we had discussed and understood various decision control statements. Next section explains you the various loop control statements in C.

5.3 LOOP CONTROL STATEMENTS

Loop control statements are used when a section of code may either be executed a fixed number of times, or while some condition is true. C gives you a choice of three types of loop statements, *while*, *do- while* and *for*.

- The *while* loop keeps repeating an action until an associated *condition* returns ***false***. This is useful where the programmer does not know in advance how many times the loop will be traversed.
- The *do while* loop is similar, but the c*ondition* is checked after the loop body is executed. This ensures that the loop body is run at least once.
- The *for* loop is frequently used, usually where the loop will be traversed a fixed number of times.

5.3.1 The *While* Loop

When in a program a single statement or a certain group of statements are to be executed repeatedly depending upon certain test condition, then *while statement* is used.

The syntax is as follows:

while (test condition*)*
{
  body_of_the_loop;
}

Here, *test condition* is an expression that controls how long the loop keeps running. Body of the loop is a statement or group of statements enclosed in braces and are repeatedly executed till the value of *test condition* evaluates to *true*. As soon as the *condition* evaluates to ***false***, the control jumps to the first statement following the *while* statement. If condition initially itself is ***false***, the body of the loop will never be executed. *While* loop is sometimes called as *entry-control loop,* as it controls the execution of the body of the loop depending upon the value of the *test condition*. This is shown in the figure 5.5 given below:

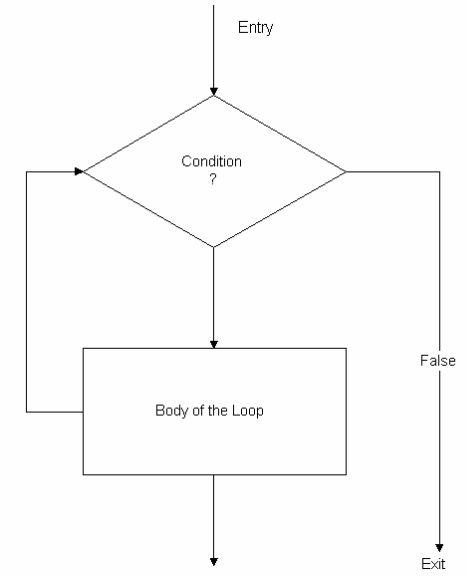


Figure 5.5: The *while* loop statement

Let us consider a program to illustrate *while loop*,

Example 5.6

Write a program to calculate the factorial of a given input natural number.

/* Program to calculate factorial of given number */

```
#include <stdio.h>
#include <math.h>
#include <stdio.h>
main( )
{
int x;
long int fact = 1;
printf("Enter any number to find factorial:\n");                    /*read the number*/
scanf("%d",&x);
while (x > 0)
    {
        fact = fact * x;          /* factorial calculation*/
        x=x-1;
       }
printf("Factorial is %ld",fact);
```

```
}
```

OUTPUT

```
Enter any number to find factorial:
4
Factorial is 24
```

Here, *condition* in *while* loop is evaluated and body of loop is repeated until *condition* evaluates to ***false*** i.e., when x becomes zero. Then the control is jumped to first statement following *while* loop and print the value of factorial.

5.3.2 The *do...while* Loop

There is another loop control structure which is very similar to the *while* statement – called as the ***do.. while*** statement. The only difference is that the expression which determines whether to carry on looping is evaluated at the end of each loop. The syntax is as follows:

```
do
{
  statement(s);
} while(test condition);
```

In *do-while* loop, the body of loop is executed at least once before the *condition* is evaluated. Then the loop repeats body as long as *condition* is ***true***. However, in *while* loop, the statement doesn't execute the body of the loop even once, if *condition* is ***false***. That is why *do-while* loop is also called *exit-control loop*. This is shown in the figure 5.6 given below.

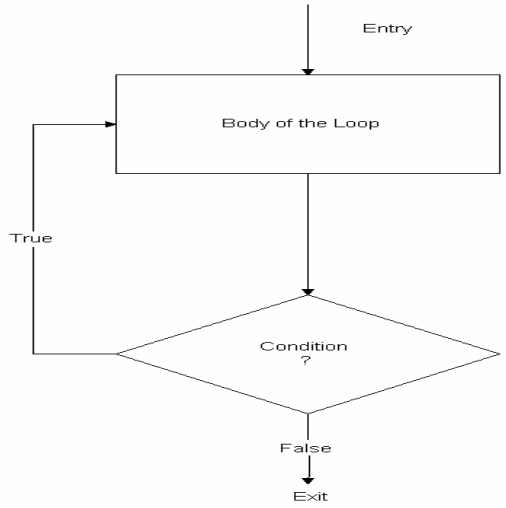


Figure 5.6: The *do…while* statement

Let us consider a program to illustrate do..*while loop*,

Example 5.7

Write a program to print first ten even natural numbers.

```
/* Program to print first ten even natural numbers */
#include <stdio.h>
main()
{
```

```
int i=0;
int j=2;
do   {
          printf("%d",j);
           j =j+2;
           i=i+1;   } while (i<10);      }
```

OUTPUT
2 4 6 8 10 12 14 16 18 20

5.3.3 The *for* Loop

for statement makes it more convenient to count iterations of a loop and works well where the number of iterations of the loop is known before the loop is entered. The syntax is as follows:

for (*initialization*; *test condition*; *increment or decrement*)
{
    Statement(s);
}

The main purpose is to repeat *statement* while *condition* remains true, like the *while* loop. But in addition, ***for*** provides places to specify an *initialization* instruction and an *increment or decrement of the control variable* instruction. So this loop is specially designed to perform a repetitive action with a counter.

The *for* loop as shown in figure 5.7, works in the following manner:

1. *initialization* is executed. Generally it is an initial value setting for a counter variable. This is executed only once.
2. *condition* is checked, if it is *true* the loop continues, otherwise the loop finishes and *statement* is skipped.
3. S*tatement(s)* is/are executed. As usual, it can be either a single instruction or a block of instructions enclosed within curly brackets { }.
4. Finally, whatever is specified in the *increment or decrement of the control variable* field is executed and the loop gets back to step 2.

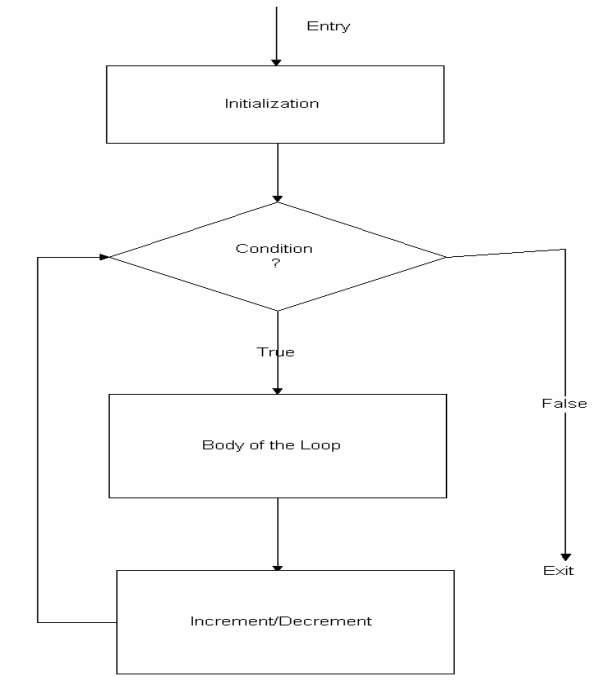


Figure 5.7: The *for* statement

Let us consider a program to illustrate *for loop*,

Example 5.8

Write a program to print first n natural numbers.

```
/* Program to print first n natural numbers */

#include <stdio.h>
main( )
{
int i,n;
printf("Enter  value of n \n");
scanf("%d",&n);
printf("\nThe first %d natural numbers are :\n", n);
for (i=1;i<=n;++i)
   {
     printf("%d",i);
    }
}
```

OUTPUT

```
Enter value of n
6
The first 6 natural numbers are:
1 2 3 4 5 6
```

The three statements inside the braces of a *for* loop usually meant for one activity each, however any of them can be left blank also. More than one control variables can be initialized but should be separated by comma.

Various forms of loop statements can be:

(a) for(;condition;increment/decrement)
body;
A blank first statement will mean no initialization.

(b) for (initialization;condition;)
body;
A blank last statement will mean no running increment/decrement.

(c) for (initialization;;increment/decrement)
body;

A blank second conditional statement means no test condition to control the exit from the loop. So, in the absence of second statement, it is required to test the condition inside the loop otherwise it results in an infinite loop where the control never exits from the loop.

(d) for (;;increment/decrement)
body;
Initialization is required to be done before the loop and test condition is checked inside the loop.

(e) for (initialization;;)
body;

Test condition and *control variable* increment/decrement is to be done inside the body of the loop.

(f) for (;condition;)
body;
Initialization is required to be done before the loop and control variable increment/decrement is to be done inside the body of the loop.

(g) for (;;;)
body;
Initialization is required to be done before the loop, *test condition* and *control variable* increment/decrement is to be done inside the body of the loop.

5.3.4 The Nested Loops

C allows loops to be *nested*, that is, one loop may be inside another. The program given below illustrates the *nesting* of loops.

Let us consider a program to illustrate *nested loops*,

Example 5.9

Write a program to generate the following pattern given below:

```
1
1       2
1       2       3
1       2       3       4
```

```
/* Program to print the pattern */

#include <stdio.h>
main( )
{
int i,j;
for (i=1;i<=4;++i)
  {
    printf("%d\n",i);
        for(j=1;j<=i;++j)
                printf("%d\t",j);
        }
}
```

Here, an *inner for loop* is written inside the *outer for loop*. For every value of ***i, j*** takes the value from 1 to ***i*** and then value of ***i*** is incremented and next iteration of outer loop starts ranging ***j*** value from 1 to ***i***.

Check Your Progress 2

1. Predict the output :

```
#include <stdio.h>
main()
{
 int i;
        for (i=0;i<=10;i++,printf("%d ",i));
}
```

……

……

2. What is the output?

```
#include <stdio.h>
main( )
{
        int i;
        for(i=0;i<3;i++)
         printf("%d ",i);
}
```

……

……

……

3. What is the output for the following program?

```
#include <stdio.h>
main( )
{
  int i=1;
  do
  {
     printf("%d",i);
  }while(i=i-1);
}
```

……

……

……

4. Give the output of the following:

```
#include <stdio.h>
main( )
{
   int i=3;
   while(i)
   {
       int x=100;
       printf("\n%d..%d",i,x);
       x=x+1;
       i=i+1;
    }
}
```

……

……

……

5.4 THE *goto* STATEMENT

The ***goto*** statement is used to alter the normal sequence of program instructions by transferring the control to some other portion of the program. The syntax is as follows:

goto label;

Here, ***label*** is an identifier that is used to label the statement to which control will be transferred. The targeted statement must be preceded by the unique label followed by colon.

label : statement;

Although *goto* statement is used to alter the normal sequence of program execution but its usage in the program should be avoided. The most common applications are:

i). To branch around statements under certain conditions in place of use of *if-else* statement,
ii). To jump to the end of the loop under certain conditions bypassing the rest of statements inside the loop in place of *continue* statement,
iii). To jump out of the loop avoiding the use of *break* statement.

goto can never be used to jump into the loop from outside and it should be preferably used for forward jump.

Situations may arise, however, in which the ***goto*** statement can be useful. To the possible extent, the use of the ***goto*** statement should generally be avoided.

Let us consider a program to illustrate *goto* and *label* statements.

Example 5.10

Write a program to print first 10 even numbers

```
/* Program to print 10 even numbers */

#include <stdio.h>
main()
{
 int i=2;
 while(1)
       {
          printf("%d ",i);
          i=i+2;
             if (i>=20)
                goto outside;
       }
  outside : printf("over");
  }
```

OUTPUT

2 4 6 8 10 12 14 16 18 20 over

5.5 THE *break* STATEMENT

Sometimes, it is required to jump out of a loop irrespective of the *conditional test value*. ***Break*** statement is used inside any loop to allow the control jump to the immediate statement following the loop. The syntax is as follows:

break;

When nested loops are used, then ***break*** jumps the control from the loop where it has been used. *Break* statement can be used inside any loop i.e., *while*, *do-while*, *for* and also in *switch* statement.

Let us consider a program to illustrate *break* statement.

Example 5.11

Write a program to calculate the first smallest divisor of a number.

```
/*Program to calculate smallest divisor of a number */

#include <stdio.h>
main( )
{
int div,num,i;
printf("Enter any number:\n");
scanf("%d",&num);
for (i=2;i<=num;++i)
   {
  if ((num % i) == 0)
          {
          printf("Smallest divisor for number %d is %d",num,i);
          break;
          }
   }
}
```

OUTPUT

```
Enter any number:
9
Smallest divisor for number 9 is 3
```

In the above program, we divide the input number with the integer starting from 2 onwards, and print the smallest divisor as soon as remainder comes out to be zero. Since we are only interested in first smallest divisor and not all divisors of a given number, so jump out of the *for* loop using *break* statement without further going for the next iteration of *for* loop.

Break is different from *exit*. Former jumps the control out of the loop while exit stops the execution of the entire program.

5.6 THE *continue* STATEMENT

Unlike *break* statement, which is used to jump the control out of the loop, it is sometimes required to skip some part of the loop and to continue the execution with next loop iteration. ***Continue*** statement used inside the loop helps to bypass the section of a loop and passes the control to the beginning of the loop to continue the execution with the next loop iteration. The syntax is as follows:

continue;

Let us see the program given below to know the working of the ***continue*** statement.

Example 5.12

Write a program to print first 20 natural numbers skipping the numbers divisible by 5.

```
/*  Program to print first 20 natural numbers skipping the numbers divisible by 5 */

#include <stdio.h>
main( )
{
          int i;
          for (i=1;i<=20;++i)
          {
```

```
                if ((i % 5) == 0)
                        continue;
                printf("%d ",i);
        }
}
```

OUTPUT

1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19

Here, the printf statement is bypassed each time when value stored in ***i*** is divisible by 5.

Check Your Progress 3

1. How many times will hello be printed by the following program?

```
#include <stdio.h>
main( )
{
        int i = 5;
        while(i)
          {
        i=i-1;
        if (i==3)
        continue;
        printf("\nhello");
        }
 }
```

……

……

……

2. Give the output of the following program segment:

```
#include <stdio.h>
main( )
{
int num,sum;
for (num=2,sum=0;;)
   {
   sum = sum + num;
    if (num > 10)
       break;
    num=num+1;
   }
    printf("%d",sum);
}
```

……

……

……

3. What is the output for the following program?

```
#include <stdio.h>
main( )
{
    int i, n = 3;
```

```
        for (i=3;n<=20;++n)
         {
          if (n%i == 0)
          break;
          if (i == n)
           printf("%d\n",i);
         }
    }
```

……

……

……

5.7 SUMMARY

A *program* is usually not limited to a linear sequence of instructions. During its process it may require to repeat execution of a part of code more than once depending upon the requirements or take decisions. For that purpose, C provides *control* and looping statements. In this unit, we had seen the different looping statements provided by C language namely ***while, do…while and for.***

Using ***break*** statement, we can leave a loop even if the condition for its end is not fulfilled. It can be used to end an infinite loop, or to force it to end before its natural end. The ***continue*** statement causes the program to skip the rest of the loop in the present iteration as if the end of the *statement* block would have reached, causing it to jump to the following iteration.

Using the ***goto*** statement, we can make an absolute jump to another point in the program. You should use this feature carefully since its execution ignores any type of nesting limitation. The destination point is identified by a label, which is then used as argument for the *goto* instruction. A *label* is made of a valid identifier followed by a colon (**:**).

5.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1 Nothing

2 hello

Check Your Progress 2

1 1 2 3 4 5 6 7 8 9 10 11

2 0 1 2

3 1 0 2

4 3..100
2..100
1..100
…..
…..
…...
till infinity

Check Your Progress 3

1 4 times

2 65

3 3

5.9 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. Programming with C, Second Edition, *Byron Gottfried*, Tata McGraw Hill, 2003.
3. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill,
4. 2002.
5. Computer Science: A Structured Programming Approach Using C, Second Edition, *Behrouz A. Forouzan, Richard F. Gilberg,* Brooks/Cole Thomas Learning, 2001.
6. The C Primer, *Leslie Hancock*, *Morris Krieger*, Mc Graw Hill, 1983.

UNIT 6 ARRAYS

Structure

6.0 Introduction
6.1 Objectives
6.2 Array Declaration
  6.2.1 Syntax of Array Declaration
  6.2.2 Size Specification
6.3 Array Initialization
  6.3.1 Initialization of Array Elements in the Declaration
  6.3.2 Character Array Initialization
6.4 Subscript
6.5 Processing the Arrays
6.6 Multi-Dimensional Arrays
  6.6.1 Multi-Dimensional Array Declaration
  6.6.2 Initialization of Two-Dimensional Arrays
6.7 Summary
6.8 Solutions / Answers
6.9 Further Readings

6.0 INTRODUCTION

C language provides four basic data types - *int, char, float and double.* We have learnt about them in Unit 3. These basic data types are very useful; but they can handle only a limited amount of data. As programs become larger and more complicated, it becomes increasingly difficult to manage the data. Variable names typically become longer to ensure their uniqueness. And, the number of variable names makes it difficult for the programmer to concentrate on the more important task of correct coding. Arrays provide a mechanism for declaring and accessing several data items with only one identifier, thereby simplifying the task of data management.

Many programs require the processing of multiple, related data items that have common characteristics like *list* of numbers, marks in a course, or enrolment numbers. This could be done by creating several individual variables. But this is a hard and tedious process. For example, suppose you want to read in five numbers and print them out in reverse order. You could do it the hard way as:

```
main()
{
 int al,a2,a3,a4,a5;
 scanf("%d %d %d %d %d",&a1,&a2,&a3,&a4,&a5);
 printf("%d %d %d %d %d",a5,a4,a3,a2,a1);
}
```

Does it look good if the problem is to read in 100 or more related data items and print them in reverse order? Of course, the solution is the use of the regular variable names **a1**, **a2** and so on. But to remember each and every variable and perform the operations on the variables is not only tedious a job and disadvantageous too. One common organizing technique is to use arrays in such situations. An array is a collection of similar kind of data elements stored in adjacent memory locations and are referred to by a single array-name. In the case of C, you have to declare and define **array** before it can be used. Declaration and definition tell the compiler the name of the array, the type of each element, and the size or number of elements.To explain it, let us consider to store marks of five students. They can be stored using five variables as follows:

```
int ar1, ar2, ar3, ar4, ar5;
```

Now, if we want to do the same thing for 100 students in a class then one will find it difficult to handle 100 variables. This can be obtained by using an array. An array declaration uses its size in [ ] brackets. For above example, we can define an array as:

int ar [100];

where *ar* is defined as an array of size 100 to store marks of integer data-type. Each element of this collection is called an *array-element* and an integer value called the *subscript* is used to denote individual elements of the array. An *ar* array is the collection of 200 consecutive memory locations referred as below:

Figure 6.1: Representation of an array

In the above figure, as each integer value occupies 2 bytes, 200 bytes were allocated in the memory.

This unit explains the use of arrays, types of arrays, declaration and initialization with the help of examples.

6.1 OBJECTIVES

After going through this unit you will be able to:

- declare and use arrays of one dimension;
- initialize arrays;
- use subscripts to access individual array elements;
- write programs involving arrays;
- do searching and sorting; and
- handle multi-dimensional arrays.

6.2 ARRAY DECLARATION

Before discussing how to declare an array, first of all let us look at the characteristic features of an array.

- Array is a data structure storing a group of elements, all of which are of the same data type.
- All the elements of an array share the same name, and they are distinguished from one another with the help of an index.
- Random access to every element using a numeric index (subscript).
- A simple data structure, used for decades, which is extremely useful.
- Abstract Data type (ADT) *list* is frequently associated with the array data structure.

The declaration of an array is just like any variable declaration with additional *size* part, indicating the number of elements of the array. Like other variables, arrays must be declared at the beginning of a function.

The declaration specifies the base type of the array, its name, and its size or dimension. In the following section we will see how an array is declared:

6.2.1 Syntax of Array Declaration

Syntax of array declaration is as follows:

data-type array_name [constant-size];

Data-type refers to the type of elements you want to store
Constant-size is the number of elements

The following are some of declarations for arrays:

```
int      char [80];
float    farr [500];
static   int iarr [80];
char     charray [40];
```

There are two restrictions for using arrays in C:

- The amount of storage for a declared array has to be specified at **compile time** before execution. This means that an array has a fixed size.
- The data type of an array applies uniformly to all the elements; for this reason, an array is called a **homogeneous** data structure.

6.2.2 Size Specification

The size of an array should be declared using symbolic constant rather a fixed integer quantity (The subscript used for the individual element is of are integer quantity). The use of a symbolic constant makes it easier to modify a program that uses an array. All reference to maximize the array size can be altered simply by changing the value of the symbolic constant. (Please refer to Unit – 3 for details regarding symbolic constants).

To declare size as 50 use the following symbolic constant, SIZE, defined:

#define SIZE 50

The following example shows how to declare and read values in an array to store marks of the students of a class.

Example 6.1

Write a program to declare and read values in an array and display them.

```
/* Program to read values in an array*/

#  include < stdio.h >
#  define  SIZE   5                          /* SIZE is a symbolic constant */

main ( )
{
int  i = 0;                                  /* Loop variable */
int   stud_marks[SIZE]; /* array declaration */

/* enter the values of the elements */
for( i = 0;i<SIZE;i++)
    {
       printf  (“Element no. =%d”,i+1);
       printf(“ Enter the value of the element:”);
```

```
        scanf("%d",&stud_marks[i]);
    }
printf("\nFollowing are the values stored in the corresponding array elements: \n\n");
for( i = 0; i<SIZE;i++)
        {
            printf("Value stored in a[%d] is %d\n"i, stud_marks[i]);
        }
}
```

OUTPUT:

Element no. = 1 Enter the value of the element = 11
Element no. = 2 Enter the value of the element = 12
Element no. = 3 Enter the value of the element = 13
Element no. = 4 Enter the value of the element = 14
Element no. = 5 Enter the value of the element = 15

Following are the values stored in the corresponding array elements:

Value stored in a[0] is 11
Value stored in a[1] is 12
Value stored in a[2] is 13
Value stored in a[3] is 14
Value stored in a[4] is 15

6.3 ARRAY INITIALIZATION

Arrays can be initialized at the time of declaration. The initial values must appear in the order in which they will be assigned to the individual array elements, enclosed within the braces and separated by commas. In the following section, we see how this can be done.

6.3.1 Initialization of Array Elements in the Declaration

The values are assigned to individual array elements enclosed within the braces and separated by comma. Syntax of array initialization is as follows:

data type array-name [size] = {val 1, val 2, ...val n};

val 1 is the value for the first array element, *val 2* is the value for the second element, and *val n* is the value for the *n* array element. Note that when you are initializing the values at the time of declaration, then there is no need to specify the size. Let us see some of the examples given below:

```
int digits  [10] = {1,2,3,4,5,6,7,8,9,10};

int digits[ ] = {1,2,3,4,5,6,7,8,9,10};

int vector[5] = {12,-2,33,21,13};

float temperature[10] ={ 31.2, 22.3, 41.4, 33.2, 23.3, 32.3, 41.1, 10.8, 11.3, 42.3};

double width[ ] = { 17.33333456, -1.212121213, 222.191345 };

int height[ 10 ] = { 60, 70, 68, 72, 68 };
```

6.3.2 Character Array Initialisation

The array of characters is implemented as strings in C. Strings are handled differently as far as initialization is concerned. A special character called null character ***' \0 '***, implicitly suffixes every string. When the external or static string character array is assigned a string constant, the size specification is usually omitted and is automatically assigned; it will include the '\0'character, added at end. For example, consider the following two assignment statements:

char thing [3] = "TIN";
char thing [ ] = "TIN";

In the above two statements the assignments are done differently. The first statement is not a string but simply an array storing three characters 'T', 'I' and 'N' and is same as writing:

char thing [3] = {'T', 'I', 'N'};

whereas, the second one is a four character string TIN\0. The change in the first assignment, as given below, can make it a string.

char thing [4] = "TIN";

Check Your Progress 1

1. What happens if I use a subscript on an array that is larger than the number of elements in the array?
   ……
   ……
2. Give sizes of following arrays.
   a. char carray [ ]= "HELLO";
   b. char carray [5]= "HELLO";
   c. char carray [ ]={ 'H', 'E', 'L', 'L', 'O' };
   ……
   ……
3. What happens if an array is used without initializing it?
   ……
   ……
4. Is there an easy way to initialize an entire array at once?
   ……
   ……
5. Use a *for* loop to total the contents of an integer array called numbers with five elements. Store the result in an integer called TOTAL.
   ……
   ……

6.4 SUBSCRIPT

To refer to the individual element in an array, a subscript is used. Refer to the statement we used in the Example 6.1,

scanf (" % d", &stud_marks[i]);

Subscript is an integer type constant or variable name whose value ranges from 0 to SIZE - 1 where SIZE is the total number of elements in the array. Let us now see how we can refer to individual elements of an array of size 5:

Consider the following declarations:

```
char country[ ] = "India";
int stud[ ] = {1, 2, 3, 4, 5};
```

Here both arrays are of size 5. This is because the country is a char array and initialized by a string constant "India" and every string constant is terminated by a null character '\0'. And stud is an integer array. country array occupies 5 bytes of memory space whereas stud occupies size of 10 bytes of memory space. The following table: 6.1 shows how individual array elements of *country* and *stud* arrays can be referred:

Table 6.1: Reference of individual elements

Element no.	Subscript	country array		stud array	
		Reference	Value	Reference	Value
1	0	country [0]	'I'	stud [0]	1
2	1	country [1]	'n'	stud [1]	2
3	2	country [2]	'd'	stud [2]	3
4	3	country [3]	'i'	stud [3]	4
5	4	country [4]	'a'	stud [4]	5

Example 6.2

Write a program to illustrate how the marks of 10 students are read in an array and then used to find the maximum marks obtained by a student in the class.

```
/* Program to find the maximum marks among the marks of 10 students*/

#  include < stdio.h >
#  define  SIZE   10                          /* SIZE is a symbolic constant */

main ( )
{

int  i = 0;
int max = 0;
int   stud_marks[SIZE]; /* array declaration */

/* enter the values of the elements */
for( i = 0;i<SIZE;i++)
    {
      printf  ("Student no. =%d",i+1);
      printf(" Enter the marks out of 50:");
      scanf("%d",&stud_marks[i]);
    }

/* find maximum */
for (i=0;i<SIZE;i ++)
    {
     if (stud_marks[i]>max)
      max = stud_marks[ i ];
     }
```

```
printf("\n\nThe maximum of the marks obtained  among all the 10 students is: %d
      ",max);
}
```

OUTPUT

```
Student no. = 1   Enter the marks out of 50: 10
Student no. = 2   Enter the marks out of 50: 17
Student no. = 3   Enter the marks out of 50: 23
Student no. = 4   Enter the marks out of 50: 40
Student no. = 5   Enter the marks out of 50: 49
Student no. = 6   Enter the marks out of 50: 34
Student no. = 7   Enter the marks out of 50: 37
Student no. = 8   Enter the marks out of 50: 16
Student no. = 9   Enter the marks out of 50: 08
Student no. = 10 Enter the marks out of 50: 37

The maximum of the marks obtained among all the 10 students is: 49
```

6.5 PROCESSING THE ARRAYS

For certain applications the assignment of initial values to elements of an array is required. This means that the array be defined globally (extern) or locally as a static array.

Let us now see in the following example how the marks in two subjects, stored in two different arrays, can be added to give another array and display the average marks in the below example.

Example 6.3:

Write a program to display the average marks of each student, given the marks in 2 subjects for 3 students.

```
/* Program to display the average marks of 3 students */

# include < stdio.h >
# define SIZE 3
main()
{
int  i  = 0;
float   stud_marks1[SIZE];        /* subject 1array declaration */
float   stud_marks2[SIZE];        /*subject 2 array declaration */
float   total_marks[SIZE];
float   avg[SIZE];

printf("\n Enter the marks in subject-1 out of 50 marks: \n");
for( i = 0;i<SIZE;i++)
              {
                printf("Student no. =%d",i+1);
                printf(" Enter the marks= ");
                scanf("%f",&stud_marks1[i]);
              }
printf("\n Enter the marks in subject-2 out of 50 marks \n");
   for(i=0;i<SIZE;i++)
              {
```

```
            printf("Student no. =%d",i+1);
            printf(" Please enter the marks= ");
           scanf("%f",&stud_marks2[i]);
          }

    for(i=0;i<SIZE;i++)
    {
          total_marks[i]=stud_marks1[i]+ stud_marks2[i];
                avg[i]=total_marks[i]/2;
                printf("Student no.=%d, Average= %f\n",i+1, avg[i]);
        }
        }
```

OUTPUT

```
Enter the marks in subject-1out of 50 marks:
Student no. = 1  Enter the marks= 23
Student no. = 2  Enter the marks= 35
Student no. = 3  Enter the marks= 42

Enter the marks in subject-2 out of 50 marks:
Student no. = 1  Enter the marks= 31
Student no. = 2  Enter the marks= 35
Student no. = 3  Enter the marks= 40

Student no. = 1 Average= 27.000000
Student no. = 2 Average= 35.000000
Student no. = 3 Average= 41.000000
```

Let us now write another program to search an element using the linear search.

Example 6.4

Write a program to search an element in a given list of elements using Linear Search.

```
/* Linear Search.*/

# include<stdio.h>
# define SIZE   05
main()
{
int  i  = 0;
int  j;
int   num_list[SIZE];      /* array declaration */

/* enter elements in the following loop */

printf("Enter any 5 numbers: \n");
for(i = 0;i<SIZE;i ++)
    {
        printf("Element no.=%d Value of the element=",i+1);
        scanf("%d",&num_list[i]);
    }
printf ("Enter the element to be searched:");
scanf ("%d",&j);

/* search using linear search */
for(i=0;i<SIZE;i++)
```

```
    {
    if(j == num_list[i])
          {
       printf("The number exists in the list at position: %d\n",i+1);
            break;
          }
    }
}
```

OUTPUT

```
Enter any 5 numbers:
Element no.=1 Value of the element=23
Element no.=2 Value of the element=43
Element no.=3 Value of the element=12
Element no.=4 Value of the element=8
Element no.=5 Value of the element=5
Enter the element to be searched: 8
The number exists in the list at position: 4
```

Example 6.5

Write a program to sort a list of elements using the selection sort method

```
/* Sorting list of numbers using selection sort method*/

#include <stdio.h>
#define SIZE 5

main()
{

int j,min_pos,tmp;
int i;                          /* Loop variable */
int a[SIZE];      /* array declaration */

/* enter the elements  */

for(i=0;i<SIZE;i++)
   {
    printf("Element no.=%d",i+1);
    printf("Value of the element: ");
    scanf("%d",&a[i]);
    }

/* Sorting by descending order*/

for (i=0;i<SIZE;i++)
  {
   min_pos = i;
   for (j=i+1;j<SIZE;j++)
     if (a[j] < a[min_pos])
            min_pos = j;
   tmp = a[i];
   a[i] = a[min_pos];
   a[min_pos] = tmp;
 }
```

```
/* print the result */

printf("The array after sorting:\n");
      for(i=0;i<SIZE;i++)
        printf("% d\n",a[i]);
}
```

OUTPUT

```
Element no. = 1 Value of the element: 23
Element no. =2  Value of the element: 11
Element no. = 3 Value of the element: 100
Element no. = 4 Value of the element: 42
Element no. = 5 Value of the element: 50

The array after sorting:
11
23
42
50
100
```

Check Your Progress 2

1. Name the technique used to pass an array to a function.
   ……

   ……
2. Is it possible to pass the whole array to a function?
   ……

   ……
3. List any two applications of arrays.
   ……

   ……

6.6 MULTI-DIMENSIONAL ARRAYS

Suppose that you are writing a chess-playing program. A chessboard is an 8-by-8 grid. What data structure would you use to represent it? You could use an array that has a chessboard-like structure, i.e. a *two-dimensional array*, to store the positions of the chess pieces. Two-dimensional arrays use two indices to pinpoint an individual element of the array. This is very similar to what is called "algebraic notation", commonly used in chess circles to record games and chess problems.

In principle, there is no limit to the number of subscripts (or dimensions) an array can have. Arrays with more than one dimension are called *multi- dimensional arrays*. While humans cannot easily visualize objects with more than three dimensions, representing multi-dimensional arrays presents no problem to computers. In practice, however, the amount of memory in a computer tends to place limits on the size of an array . A simple four-dimensional array of double-precision numbers, merely twenty elements wide in each dimension, takes up 20^4 * 8, or 1,280,000 bytes of memory - about a megabyte.

For exmaple, you have ten rows and ten columns, for a total of 100 elements. It's really no big deal. The first number in brackets is the number of rows, the second number in brackets is the number of columns. So, the upper left corner of any grid

would be element [0][0]. The element to its right would be [0][1], and so on. Here is a little illustration to help.

[0][0]	[0][1]	[0][2]
[1][0]	[1][1]	[1][2]
[2][0]	[2][1]	[2][2]

Three-dimensional arrays (and higher) are stored in the same way as the two-dimensional ones. They are kept in computer memory as a linear sequence of variables, and the last index is always the one that varies fastest (then the next-to-last, and so on).

6.6.1 Multi - Dimensional Array Declaration

You can declare an array of two dimensions as follows:

datatype array_name[*size1*][*size2*];

In the above example, *variable_type* is the name of some type of variable, such as int. Also, *size1* and *size2* are the sizes of the array's first and second dimensions, respectively. Here is an example of defining an 8-by-8 array of integers, similar to a chessboard. Remember, because C arrays are zero-based, the indices on each side of the chessboard array run 0 through 7, rather than 1 through 8. The effect is the same: a two-dimensional array of 64 elements.

int chessboard [8][8];

To pinpoint an element in this grid, simply supply the indices in both dimensions.

6.6.2 Initialisation of Two - Dimensional Arrays

If you have an *m* x *n* array, it will have *m * n* elements and will require *m*n*element-size* bytes of storage. To allocate storage for an array you must reserve this amount of memory. The elements of a two-dimensional array are stored row wise. If table is declared as:

int table [2] [3] = { 1,2,3,4,5,6 };

It means that element
table [0][0] = 1;
table [0][1] = 2;
table [0][2] = 3;
table [1][0] = 4;
table [1][1] = 5;
table [1][2] = 6;

The neutral order in which the initial values are assigned can be altered by including the groups in { } inside main enclosing brackets, like the following initialization as above:

```
int     table  [ 2 ] [ 3 ] = { {1,2,3},
                               {4,5,6}    };
```

The value within innermost braces will be assigned to those array elements whose last subscript changes most rapidly. If there are few remaining values in the row, they will be assigned zeros. The number of values cannot exceed the defined row size.

```
int     table [ 2 ] [ 3 ] = { { 1, 2, 3},{ 4}};
```

It assigns values as
table [0][0] = 1;
table [0][1] = 2;
table [0][2] = 3;
table [1][0] = 4;
table [1][1] = 0;
table [1][2] = 0

Remember that, C language performs no error checking on array bounds. If you define an array with 50 elements and you attempt to access element 50 (the 51st element), or any out of bounds index, the compiler issues no warnings. It is the programmer's task to check that all attempts to access or write to arrays are done only at valid array indexes. Writing or reading past the end of arrays is a common programming bug and is hard to isolate.

Check Your Progress 3

1. Declare a multi-dimensioned array of floats called balances having three rows and five columns.
   ……
   ……
2. Write a *for* loop to total the contents of the multi-dimensioned float array balances.
   ……
   ……
3. Write a for loop which will read five characters (use scanf) and deposit them into the character based array words, beginning at element 0.
   ……
   ……

6.7 SUMMARY

Like other languages, C uses arrays as a way of describing a collection of variables with identical properties. The group has a single name for all its members, with the individual member being selected by an *index*. We have learnt in this unit, the basic purpose of using an array in the program, declaration of array and assigning values to the arrays. All elements of the arrays are stored in the consecutive memory locations**.** Without exception, all arrays in C are indexed from 0 up to one less than the bound given in the declaration. This is very puzzling for a beginner. Watch out for it in the examples provided in this unit. One important point about array declarations is that they don't permit the use of varying subscripts. The numbers given must be constant expressions which can be evaluated at compile time, not run time. As with other variables, global and static array elements are initialized to 0 by default, and automatic array elements are filled with garbage values. In C, an array of type char is used to represent a character string, the end of which is marked by a byte set to 0 (also known as a NULL character).

Whenever the arrays are passed to function their starting address is used to access rest of the elements. This is called – Call by reference. Whatever changes are made to the

elements of an array in the function, they are also made available in the calling part. The formal argument contains no size specification except for the rightmost dimension. Arrays and pointers are closely linked in C. Multi-dimensional arrays are simply arrays of arrays. To use arrays effectively it is a good idea to know how to use pointers with them. More about the pointers can be learnt from Unit -10 (Block -3).

6.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. If you use a subscript that is out of bounds of the array declaration, the program will probably compile and even run. However, the results of such a mistake can be unpredictable. This can be a difficult error to find once it starts causing problems. So, make sure you're careful when initializing and accessing the array elements.

2.
   a) 6
   b) 5
   c) 5

3. This mistake doesn't produce a compiler error. If you don't initialize an array, there can be any value in the array elements. You might get unpredictable results. You should always initialize the variables and the arrays so that you know their content.

4. Each element of an array must be initialized. The safest way for a beginner is to initialize an array, either with a declaration, as shown in this chapter, or with a ***for*** statement. There are other ways to initialize an array, but they are beyond the scope of this Unit.

5. Use a ***for*** loop to total the contents of an integer array which has five elements. Store the result in an integer called total.

```
for ( loop = 0, total = 0; loop < 5; loop++ )
total = total + numbers[loop];
```

Check Your Progress 2

1. Call by reference.

2. It is possible to pass the whole array to a function. In this case, only the address of the array will be passed. When this happens, the function can change the value of the elements in the array.

3. Two common statistical applications that use arrays are:

   - **Frequency distributions**: A frequency array show the number of elements with an identical value found in a series of numbers. For example, suppose we have taken a sample of 50 values ranging from 0 to 10. We want to know how many of the values are 0, how many are 1, how many are 2 and so forth up to 10. Using the arrays we can solve the problem easily . Histogram is a pictorial representation of the frequency array. Instead of printing the values of the elements to show the frequency of each number, we print a histogram in the form of a bar chart.
   - **Random Number Permutations**: It is a set of random numbers in which no numbers are repeated. For example, given a random number permutation of 5 numbers, the values of 0 to 5 would all be included with no duplicates.

Check Your Progress 3

1. float balances[3][5];

2. for(row = 0, total = 0; row < 3; row++)
   for(column = 0; column < 5; column++)
   total = total + balances[row][column];

3. for(loop = 0; loop < 5; loop++)
   scanf ("%c", &words[loop]);

6.9 FURTHER READINGS

1. The C Programming Language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. C, The Complete Reference, Fourth Edition, *Herbert Schildt*, TMGH, 2002.
3. Computer Science – A Structured Programming Approach Using C, *Behrouz A. Forouzan, Richard F. Gilberg*, Thomas Learning, Second edition, 2001.
4. Programming with ANSI and TURBO C, *Ashok N. Kamthane*, Pearson Education, 2002.

UNIT 7 STRINGS

Structure

7.0 Introduction
7.1 Objectives
7.2 Declaration and Initialization of Strings
7.3 Display of Strings Using Different Formatting Techniques
7.4 Array of Strings
7.5 Built-in String Functions and Applications
7.5.1 Strlen Function
7.5.2 Strcpy Function
7.5.3 Strcmp Function
7.5.4 Strcat Function
7.5.5 Strlwr Function
7.5.6 Strrev Function
7.5.7 Strspn Function
7.6 Other String Functions
7.7 Summary
7.8 Solutions / Answers
7.9 Further Readings

7.0 INTRODUCTION

In the previous unit, we have discussed numeric arrays, a powerful data storage method that lets you group a number of same-type data items under the same group name. Individual items, or elements, in an array are identified using a subscript after the array name. Computer programming tasks that involve repetitive data processing lend themselves to array storage. Like non-array variables, arrays must be declared before they can be used. Optionally, array elements can be initialized when the array is declared. In the earlier unit, we had just known the concept of *character arrays* which are also called *strings*.

String can be represented as a single-dimensional character type array. C language does not provide the intrinsic string types. Some problems require that the characters within a string be processed individually. However, there are many problems which require that strings be processed as complete entities. Such problems can be manipulated considerably through the use of special string oriented library functions. Most of the C compilers include string library functions that allow string comparison, string copy, concatenation of strings etc. The string functions operate on null-terminated arrays of characters and require the header <string.h>.The use of the some of the string library functions are given as examples in this unit.

7.1 OBJECTIVES

After going through this unit, you will be able to:

- define, declare and initialize a string;
- discuss various formatting techniques to display the strings; and
- discuss various built-in string functions and their use in manipulation of strings.

7.2 DECLARATION AND INITIALIZATION OF STRINGS

Strings in C are group of characters, digits, and symbols enclosed in quotation marks or simply we can say the string is declared as a “character array”. The end of the string is marked with a special character, the ‘\0’ (*Null character)*, which has the decimal value 0. There is a difference between a *character* stored in memory and a

single character string stored in a memory. The character requires only one byte whereas the single character string requires two bytes (one byte for the character and other byte for the delimiter).

Declaration of strings

A string in C is simply a sequence of characters. To declare a string, specify the data type as char and place the number of characters in the array in square brackets after the string name. The syntax is shown as below:

char string-name[size];

For example,

char name[20];
char address[25];
char city[15];

Initialization of strings

The string can be initialized as follows:

char name[8] = {'P', 'R', 'O', 'G', 'R', 'A', 'M', '\0'};

Each character of string occupies 1 byte of memory (on 16 bit computing). The size of character is machine dependent, and varies from 16 bit computers to 64 bit computers. The characters of strings are stored in the contiguous (adjacent) memory locations.

1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte	1 byte
P	R	O	G	R	A	M	\0
1001	1002	1003	1004	1005	1006	1007	1008

The C compiler inserts the NULL (\0) character automatically at the end of the string. So initialization of the NULL character is not essential.

You can set the initial value of a character array when you declare it by specifying a string literal. If the array is too small for the literal, the literal will be truncated. If the literal (including its null terminator) is smaller than the array, then the final characters in the array will be undefined. If you don't specify the size of the array, but do specify a literal, then C will set the array to the size of the literal, including the null terminator.

```
char str[4] = {'u', 'n', 'i', 'x'};
char str[5] = {'u', 'n', 'i', 'x', '\0'};
char str[3];
char str[ ] =  "UNIX";
char str[4] = "unix";
char str[9] = "unix";
```

All of the above declarations are legal. But which ones don't work? The first one is a valid declaration, but will cause major problems because it is not *null-terminated.* The second example shows a correct null-terminated string. The special escape character **\0** denotes string termination. The fifth example suffers the size problem, the character array ***'str'*** is of size 4 bytes, but it requires an additional space to store ***'\0'***. The fourth example however does not. This is because the compiler will determine the length of the string and automatically initialize the last character to a null-terminator. The strings not terminated by a ***'\0'*** are merely a collection of characters and are called as *character arrays*.

String Constants

String constants have double quote marks around them, and can be assigned to char pointers. Alternatively, you can assign a string constant to a char array - either with no size specified, or you can specify a size, but don't forget to leave a space for the null character! Suppose you create the following two code fragments and run them:

```
/*   Fragment 1   */
{
   char *s;
   s=hello";
   printf("%s\n",s);
}

/*  Fragment  2 */

{
   char s[100];
   strcpy(s, " hello");
   printf("%s\n",s);
}
```

These two fragments produce the same output, but their internal behaviour is quite different. In fragment 2, you cannot say **s = "hello";**. To understand the differences, you have to understand how the *string constant table* works in C. When your program is compiled, the compiler forms the object code file, which contains your machine code and a table of all the string constants declared in the program. In fragment 1, the statement **s = "hello";** causes ***s*** to point to the address of the string **hello** in the string constant table. Since this string is in the string constant table, and therefore technically a part of the executable code, you cannot modify it. You can only point to it and use it in a read-only manner. In fragment 2, the string **hello** also exists in the constant table, so you can copy it into the array of characters named ***s***. Since ***s*** is not an address, the statement **s="hello";** will not work in fragment 2. It will not even compile.

Example 7.1

Write a program to read a name from the keyboard and display message **Hello** onto the monitor

Program 7.1

```
/*Program that reads the name and display the hello along with your name*/
#include <stdio.h>
main()
{
char name[10];
printf("\nEnter Your Name : ");
scanf("%s", name);
printf("Hello %s\n", name);
}
```

OUTPUT

```
Enter Your Name :  Alex
Hello Alex
```

In the above example declaration char name [10] allocates 10 bytes of memory space (on 16 bit computing) to array name [ ]. We are passing the base address to scanf function and scanf() function fills the characters typed at the keyboard into array until enter is pressed. The scanf() places '\0' into array at the end of the input. The printf()

function prints the characters from the array on to monitor, leaving the end of the string '\0'. The *%s* used in the scanf() and printf() functions is a format specification for strings.

7.3 DISPLAY OF STRINGS USING DIFFERENT FORMATTING TECHNIQUES

The ***printf*** function with ***%s*** format is used to display the strings on the screen. For example, the below statement displays entire string:

printf("%s", name);

We can also specify the accuracy with which character array (string) is displayed. For example, if you want to display first 5 characters from a field width of 15 characters, you have to write as:

printf("%15.5s", name);

If you include minus sign in the format (e.g. % –10.5s), the string will be printed left justified.

printf("% -10.5s", name);

Example 7.2

Write a program to display the string "UNIX" in the following format.

```
U
UN
UNI
UNIX
UNIX
UNI
UN
U
```

```
/* Program to display the string in the above shown format*/

# include <stdio.h>
main()
{
int  x, y;
static char string[ ] = "UNIX";
printf("\n");
for( x=0; x<4; x++)
{
        y = x + 1;
  /* reserves 4 character of space on to the monitor and minus sign is for left
justified*/
        printf("%-4.*s \n", y, string);

 /* and for every loop the * is replaced by value of y */
/* y value starts with 1 and for every time it is incremented by 1 until it reaches to 4*/
}

for( x=3; x>=0; x- -)
        {
         y = x + 1;
         printf("%-4.*s \n", y, string);
```

```
/*  y value starts with 4 and for every time it is decrements by 1 until it reaches to 1*/
        }
}
```

OUTPUT

```
U
UN
UNI
UNIX
UNIX
UNI
UN
U
```

7.4 ARRAY OF STRINGS

Array of strings are multiple strings, stored in the form of table. Declaring array of strings is same as strings, except it will have additional dimension to store the number of strings. Syntax is as follows:

char array-name[size][size];

For example,

char names[5][10];

where names is the name of the character array and the constant in first square brackets will gives number of string we are going to store, and the value in second square bracket will gives the maximum length of the string.

Example 7.3

char names [3][10] = {“martin”, “phil”, “collins”};

It can be represented by a two-dimensional array of size[3][10] as shown below:

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
m	a	r	t	i	n	\0			
p	h	i	l	\0					
c	o	l	l	i	n	s	\0		

Example 7.4

Write a program to initializes 3 names in an array of strings and display them on to monitor

```
/* Program that initializes 3 names in an array of strings and display them on to
monitor.*/

#include <stdio.h>
main()
{
        int   n;
        char   names[3][10] = {“Alex”, “Phillip”, “Collins” };
        for(n=0; n<3; n++)
        printf(“%s \n”,names[n] );    }
```

OUTPUT

Alex
Phillip
Collins

Check Your Progress 1

1. Which of the following is a static string?

   A. Static String;
   B. “Static String”;
   C. ‘Static String’;
   D. char string[100];

……

……

……

2. Which character ends all strings?

   A. ‘.’
   B. ‘ ‘
   C. ‘0’
   D. ‘n’

……

……

……

3. What is the Output of the following programs?

(a)
```
main()
{
    char name[10] = “IGNOU”;
    printf(“\n %c”, name[0]);
    printf(“\n %s”, name);
}
```

(b)
```
main()
{
  char  s[ ] = “hello”;
  int j = 0;
  while ( s[j] != ‘\0’ )
      printf(“ %c”,s[j++]);
}
```

(c)
```
main()
{
  char  str[ ] = “hello”;
  printf(“%10.2s”, str);
  printf(“%-10.2s”, str);
}
```

……

……

……

4 Write a program to read 'n' number of lines from the keyboard using a two-dimensional character array (ie., strings).

……

……

……

7.5 BUILT IN STRING FUNCTIONS AND APPLICATIONS

The header file <string.h> contains some string manipulation functions. The following is a list of the common string managing functions in C.

7.5.1 Strlen Function

The **strlen** function returns the length of a string. It takes the string name as argument. The syntax is as follows:

n = strlen (str);

where **str** is name of the string and **n** is the length of the string, returned by **strlen** function.

Example 7. 5

Write a program to read a string from the keyboard and to display the length of the string on to the monitor by using strlen( ) function.

```
/*  Program to illustrate the strlen function to determine the length of a string */

#include <stdio.h>
#include <string.h>
main()
{
char name[80];
int length;
printf("Enter your name: ");
gets(name);
length = strlen(name);
printf("Your name has %d characters\n", length);
}
```

OUTPUT

```
Enter your name: TYRAN
Your name has 5 characters
```

7.5.2 Strcpy Function

In C, you cannot simply assign one character array to another. You have to copy element by element. The string library <string.h> contains a function called **strcpy** for this purpose. The **strcpy** function is used to copy one string to another. The syntax is as follows:

strcpy(str1, str2);

where str1, str2 are two strings. The content of string str2 is copied on to string str1.

Example 7.6

Write a program to read a string from the keyboard and copy the string onto the second string and display the strings on to the monitor by using strcpy( ) function.

```
/* Program to illustrate strcpy function*/

#include <stdio.h>
#include <string.h>
main()
{
char first[80], second[80];
printf("Enter a string: ");
gets(first);
strcpy(second, first);
printf("\n First string is : %s, and second string is: %s\n", first, second);
}
```

OUTPUT

```
Enter a string: ADAMS
First string is: ADAMS, and second string is: ADAMS
```

7.5.3 Strcmp Function

The **strcmp** function in the string library function which compares two strings, character by character and stops comparison when there is a difference in the ASCII value or the end of any one string and returns ASCII difference of the characters that is integer. If the return value ***zero*** means the two strings are equal, a negative value means that first is less than second, and a positive value means first is greater than second**.** The syntax is as follows:

n = strcmp(str1, str2);
where **str1** and **str2** are two strings to be compared and **n** is returned value of differed characters.

Example 7.7

Write a program to compare two strings using string compare function.

```
/* The following program uses the strcmp function to compare two strings. */

#include <stdio.h>
#include <string.h>
main()
{
 char first[80], second[80];
 int value;
printf("Enter a string: ");
 gets(first);
 printf("Enter another string: ");
 gets(second);
 value = strcmp(first,second);
  if(value == 0)
       puts("The two strings are equal");
    else if(value < 0)
         puts("The first string is smaller ");
        else if(value > 0)
```

```
        puts("the first string is bigger");
}
```

OUTPUT

```
Enter a string: MOND
Enter another string: MOHANT
The first string is smaller
```

7.5.4 Strcat Function

The **strcat** function is used to join one string to another. It takes two strings as arguments; the characters of the second string will be appended to the first string. The syntax is as follows:

strcat(str1, str2);
where str1 and str2 are two string arguments, string *str2* is appended to string *str1*.

Example 7.8

Write a program to read two strings and append the second string to the first string.

```
/* Program for string concatenation*/

#include <stdio.h>
#include <string.h>
main()
{
char first[80], second[80];
printf("Enter a string:");
gets(first);
printf("Enter another string: ");
gets(second);
strcat(first, second);
printf("\nThe two strings joined together: %s\n", first);
}
```

OUTPUT

```
Enter a string: BOREX
Enter another string: BANKS
The two strings joined together: BOREX BANKS
```

7.5.5 Strlwr Function

The **strlwr** function converts upper case characters of string to lower case characters. The syntax is as follows:

strlwr(str1);
where str1 is string to be converted into lower case characters.

Example 7.9

Write a program to convert the string into lower case characters using in-built function.

```
/* Program that converts input string to lower case characters */

#include <stdio.h>
#include <string.h>
```

```
main()
{
char first[80];
printf("Enter a string: ");
gets(first);
printf("Lower case of the string is %s", strlwr(first));
}
```

OUTPUT

```
Enter a string: BROOKES
Lower case of the string is brookes
```

7.5.6 Strrev Function

The **strrev** funtion reverses the given string. The syntax is as follows:

strrev(str);
where string **str** will be reversed.

Example 7.9

Write a program to reverse a given string.

```
/* Program to reverse a given string */

#include <stdio.h>
#include <string.h>
main()
{
char first[80];
printf("Enter a string:");
gets(first);
printf("\n Reverse of the given string is :  %s ", strrev(first));
}
```

OUTPUT

```
Enter a string: ADANY
Reverse of the given string is:  YNADA
```

7.5.7 Strspn Function

The **strspn** function returns the position of the string, where first string mismatches with second string. The syntax is as follows:

n = strspn (first, second);
where **first** and **second** are two strings to be compared, **n** is the number of character from which first string does not match with second string.

Example 7.10

Write a program, which returns the position of the string from where first string does not match with second string.

```
/*Program which returns the position of the string from where first string does not
match with second string*/

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

```
{
char first[80], second[80];
printf("Enter  first string: “);
gets(first);
printf(“\n Enter  second string: “);
gets(second);
printf(“\n After %d characters there is no match”,strspn(first, second));
}
```

OUTPUT

```
Enter first string: ALEXANDER
Enter second string: ALEXSMITH
After 4 characters there is no match
```

7.6 OTHER STRING FUNCTIONS

strncpy function

The **strncpy** function same as *strcpy*. It copies characters of one string to another string up to the specified length. The syntax is as follows:

strncpy(str1, str2, 10);
where str1 and str2 are two strings. The **10** characters of string **str2** are copied onto string **str1.**

stricmp function

The **stricmp** function is same as *strcmp*, except it compares two strings ignoring the case (lower and upper case). The syntax is as follows:

n = stricmp(str1, str2);

strncmp function

The **strncmp** function is same as *strcmp*, except it compares two strings up to a specified length. The syntax is as follows:

n = strncmp(str1, str2, 10);
where **10** characters of **str1** and **str2** are compared and **n** is returned value of differed characters.

strchr function

The **strchr** funtion takes two arguments (the string and the character whose address is to be specified) and returns the address of first occurrence of the character in the given string. The syntax is as follows:

cp = strchr (str, c);
where **str** is string and **c** is character and **cp** is character pointer.

strset function

The **strset** funtion replaces the string with the given character**.** It takes two arguments the string and the character. The syntax is as follows:

strset (first, ch);
where string **first** will be replaced by character **ch**.

strchr function

The **strchr** function takes two arguments (the string and the character whose address is to be specified) and returns the address of first occurrence of the character in the given string. The syntax is as follows:

cp = strchr (str, c);
where **str** is string and **c** is character and **cp** is character pointer.

strncat function

The **strncat** function is the same as *strcat*, except that it appends upto specified length. The syntax is as follows:

strncat(str1, str2,10);
where 10 character of the str2 string is added into str1 string.

strupr function

The strupr function converts lower case characters of the string to upper case characters. The syntax is as follows:

strupr(str1);
where str1 is string to be converted into upper case characters.

strstr function

The **strstr** function takes two arguments address of the string and second string as inputs. And returns the address from where the second string starts in the first string. The syntax is as follows:

cp = strstr (first, second);
where **first** and s**econd** are two strings, **cp** is character pointer.

Check Your Progress 2

1. Which of the following functions compares two strings?
   A. compare();
   B. stringcompare();
   C. cmp();
   D. strcmp();

……

……

……

2. Which of the following appends one string to the end of another?
   A. append();
   B. stringadd();
   C. strcat();
   D. stradd();

……

……

……

3. Write a program to concatenate two strings without using the *strcat()* function.

……

……

……

4. Write a program to find string length without using the *strlen()* function.

……

……

……

5. Write a program to convert lower case letters to upper case letters in a given string without using strupp().

……

……

……

7.7 SUMMARY

Strings are sequence of characters. Strings are to be null-terminated if you want to use them properly. Remember to take into account null-terminators when using dynamic memory allocation. The string.h library has many useful functions. Losing the ' **\0'** character can lead to some very considerable bugs. Make sure you copy \0 when you copy strings. If you create a new string, make sure you put \0 in it. And if you copy one string to another, make sure the receiving string is big enough to hold the source string, including \0. Finally, if you point a character pointer to some characters, make sure they end with \0.

String Functions	Its Use
strlen	Returns number of characters in string.
strlwr	Converts all the characters in the string into lower case characters
strcat	Adds one string at the end of another string
strcpy	Copies a string into another
strcmp	Compares two strings and returns zero if both are equal.
strdup	Duplicates a string
strchr	Finds the first occurrence of given character in a string
strstr	Finds the first occurrence of given string in another string
strset	Sets all the characters of string to given character or symbol
strrev	Reverse a string

7.8 SOLUTIONS / ANSWERS

Check Your Progress 1

1. B

2. C

3. (a) I
IGNOU

(b) hello

(c) hehe

Check Your Progress 2

1. D

2. C

3. /* Program to concatenate two strings without using the strcat() function*/

```
# include<string.h>
# include <stdio.h>
 main()
 {
  char str1[10];
  char str2[10];
  char output_str[20];
  int i=0, j=0, k=0;
 printf(" Input the first string: ");
           gets(str1);
           printf("\nInput the second string: ");
           gets(str2);
          while(str1[i] != '\0')
  output_str[k++] = str1[i++];
 while(str2[j] != '\0')
  output_str[k++] = str2[j++];
 output_str[k] = '\0';
          puts(output_str);
 }
```

4. /* Program to find the string length without using the strlen() funtion */

```
# include<stdio.h>
# include<string.h>
main()
{
char  string[60];
int len=0, i=0;
printf(" Input the string : ");
gets(string);
while(string[i++] != '\0')
              len ++;
printf("Length of Input String = %d", len);
getchar();
}
```

5. /* Program to convert the lower case letters to upper case in a given string without using strupp() function*/

```
#include<stdio.h>
main()
{
int i= 0;   char source[10], destination[10];
gets(source);
while( source[i] != '\0')
{
  if((source[i]>=97) && (source[i]<=122))
```

```
            destination[i]=source[i]-32;
        else
            destination[i]=source[i];
    i++;
    }
    destination[i]= ' \0 ';
    puts(destination);
}
```

7.9 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI.
2. Programming with ANSI and Turbo C, *Ashok N. Kamthane*, Pearson Education, 2002.
3. Computer Programming in C, *Raja Raman. V*, 2002, PHI.
4. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill, 2002.
5. Computer Science A structured Programming Approach Using C, *Behrouz A. Forouzan, Richard F. Gilberg*, Brooks/Cole Thomas Learning, Second Edition, 2001.

UNIT 8 FUNCTIONS

Structure

8.0 Introduction
8.1 Objectives
8.2 Definition of a Function
8.3 Declaration of a Function
8.4 Function Prototypes
8.5 The Return Statement
8.6 Types of Variables and Storage Classes
  8.6.1 Automatic Variables
  8.6.2 External Variables
  8.6.3 Static Variables
  8.6.4 Register Variables
8.7 Types of Function Invoking
8.8 Call by Value
8.9 Recursion
8.10 Summary
8.11 Solutions / Answers
8.12 Further Readings

8.0 INTRODUCTION

To make programming simple and easy to debug, we break a larger program into smaller *subprograms* which perform '*well defined tasks*'. These subprograms are called *functions.* So far we have defined a single function *main ( ).*

After reading this unit you will be able to define many other functions and the *main( )* function can call up these functions from several different places within the program, to carry out the required processing.

Functions are very important tools for ***Modular Programming***, where we break large programs into small subprograms or modules (functions in case of C). The use of functions reduces complexity and makes programming simple and easy to understand.

In this unit, we will discuss how functions are defined and how are they accessed from the main program? We will also discuss various types of functions and how to invoke them. And finally you will learn an interesting and important programming technique known as *Recursion*, in which a function calls within itself.

8.1 OBJECTIVES

After going through this unit, you will learn:

- the need of functions in the programming;
- how to define and declare functions in 'C' Language;
- different types of functions and their purpose;
- how the functions are called from other functions;
- how data is transferred through parameter passing, to functions and the Return statement;
- recursive functions; and
- the concept of 'C*all by Value*' and its drawbacks.

8.2 DEFINTION OF A FUNCTION

A ***function*** is a self- contained block of executable code that can be called from any other function .In many programs, a set of statements are to be executed repeatedly at various places in the program and may with different sets of data, the idea of functions comes in mind. You keep those repeating statements in a function and call them as and when required. When a function is called, the control transfers to the called function, which will be executed, and then transfers the control back to the calling function (to the statement following the function call). Let us see an example as shown below:

Example 8.1

```
/* Program to illustrate a function*/

#include <stdio.h>
main ()
{
void sample( );
printf("\n You are in main");
}

void sample( )
 {
  printf("\n You are in sample");
  }
```

OUTPUT

```
You are in sample
You are in main
```

Here we are calling a function ***sample ( )*** through ***main( )*** i.e. control of execution transfers from ***main( )*** to ***sample( )*** , which means ***main( )*** is suspended for some time and ***sample( )*** is executed. After its execution the control returns back to ***main( )***, at the statement following function call and the execution of ***main( )*** is resumed.

The syntax of a function is:

```
return  data type  function_name (list of arguments)
{
        datatype declaration of the arguments;
        executable statements;
        return (expression);
}
```

where,

- return data type is the same as the data type of the variable that is returned by the function using return statement.
- a function_name is formed in the same way as variable names / identifiers are formed.
- the list of arguments or parameters are valid variable names as shown below, separated by commas: (data type1 var1,data type2 var2,…….. data type n var n) for example *(int x, float y, char z)*.
- arguments give the values which are passed from the calling function.

- the body of function contains executable statements.
- the return statement returns a ***single*** value to the calling function.

Example 8.2

Let us write a simple function that calculates the square of an integer.

```
/*Program to calculate the square of a given integer*/

/* square( ) function */
{
  int square (int no)                  /*passing of argument */
  int result ;                 /* local variable to function square */
  result = no*no;
  return (result);                     /* returns an integer value */
          }

  /*It will be called from main()as follows */
main( )
{
int n ,sq;                     /* local variable to function main */
printf (“Enter a number to calculate square value”);
scanf(“%d”,&n);
sq=square(n);                /* function call with parameter passing */
printf (”\nSquare of  the number is : %d”, sq);
 }  /* program  ends */
```

OUTPUT

```
Enter a number to calculate square value : 5
Square of the number is : 25
```

8.3 DECLARATION OF A FUNCTION

As we have mentioned in the previous section, every function has its declaration and function definition. When we talk of declaration only, it means only the function name, its argument list and return type are specified and the function body or definition is not attached to it. The *syntax* of a function declaration is:

return data type function_name(list of arguments);

For example,

```
int square(int no);
float temperature(float c, float f);
```

We will discuss the use of function declaration in the next section.

8.4 FUNCTION PROTOTYPES

In Example 8.1 for calculating square of a given number, we have declared function ***square( )*** before ***main( )*** function; this means before coming to ***main( )***, the compiler knows about ***square( )***, as the compilation process starts with the first statement of

any program. Now suppose, we reverse the sequence of functions in this program i.e., writing the ***main( )*** *function* and later on writing the ***square( )*** function, *what happens* ? The "C" compiler will give an error. Here the introduction of concept of *"function prototypes"* solves the above problem.

Function Prototypes require that every function which is to be accessed should be declared in the calling function. The function declaration, that will be discussed earlier, will be included for every function in its calling function . Example 8.2 may be modified using the function prototype as follows:

Example 8.3

```
/*Program to calculate the square of a given integer using the function prototype*/
#include <stdio.h>
main ( )
{
int n , sq ;
int square (int ) ;                    /* function prototype */
printf ("Enter a number to calculate square value");
scanf("%d",&n);
sq = square(n);              /* function call with parameter passing */
printf ("\nSsquare of  the number is : %d", sq);
}

/* square function */
int square (int no)                    /*passing of argument */
{
  int result ;               /* local variable to function square */
  result = no*no;
  return (result);                     /* returns an integer value */
}
```

OUTPUT

```
Enter a number to calculate square value :  5
Square of the number is:  25
```

Points to remember:

- *Function prototype* requires that the function declaration must include the return type of function as well as the type and number of arguments or parameters passed.
- The variable names of arguments need not be declared in prototype.
- The major reason to use this concept is that they enable the compiler to check if there is any mismatch between function declaration and function call.

Check Your Progress 1

(1) Write a function to multiply two integers and display the product.

……

……

(2) Modify the above program, by introducing function prototype in the main function.

……

……

8.5 THE *return* STATEMENT

If a function has to return a value to the calling function, it is done through the ***return*** statement. It may be possible that a function does not return any value; only the control is transferred to the calling function. The syntax for the *return* statement is:

return (expression);

We have seen in the *square( )* function, the *return* statement, which returns an integer value.

Points to remember:

- You can pass any number of arguments to a function but can return only one value at a time.

  For example, the following are the valid *return* statements

  (a) return (5);
  (b) return (x*y);

  For example, the following are the invalid *return* statements
  (c) return (2, 3);
  (d) return (x, y);

- If a function does not return anything, ***void*** specifier is used in the function declaration.

  For example:

```
void square (int no)
 {
 int sq;
 sq = no*no;
 printf ("square is %d", sq);
 }
```

- All the function's return type is by default is ***"int"***, i.e. a function returns an integer value, if no type specifier is used in the function declaration.

  Some examples are:

  (i) square (int no); /* will return an integer value */

  (ii) int square (int no); /* will return an integer value */

  (iii) void square (int no); /* will not return anything */

- What happens if a function has to return some value other than integer? The answer is very simple: use the particular type specifier in the function declaration.

For example consider the code fragments of function definitions below:

1) **Code Fragment - 1**

```
char func_char( …….. )
 {
      char c;
```

```
…….
…….
……
}
```

2) **Code Fragment - 2**

```
float func_float (……..)
{
  float f;
       ……..
       ……..
       ……..
       return(f);
}
```

Thus from the above examples, we see that you can return all the data types from a function, the only condition being that the value returned using return statement and the type specifier used in function declaration should match.

- A function can have many *return* statements. This thing happens when some condition based returns are required.

  For example,

```
/*Function to find greater of two numbers*/
int greater (int x, int y)
 {
     if (x>y)
         return (x);
      else
         return (y);
  }
```

- And finally, with the execution of return statement, the control is transferred to the calling function with the value associated with it.

  In the above example if we take x = 5 and y = 3, then the control will be transferred to the calling function when the first return statement will be encountered, as the condition (x > y) will be satisfied. All the remaining executable statements in the function will not be executed after this returning.

Check Your Progress 2

1. Which of the following are valid return statements?

   a) return (a);
   b) return (z,13);
   c) return (22.44);
   d) return;
   e) return (x*x, y*y);

……

……

……

8.6 TYPES OF VARIABLES AND STORAGE CLASSES

In a program consisting of a number of functions a number of different types of variables can be found.

Global vs. Static variables: Global variables are recognized through out the program whereas local valuables are recognized only within the function where they are defined.

Static vs. Dynamic variables: Retention of value by a local variable means, that in static, retention of the variable value is lost once the function is completely executed whereas in certain conditions the value of the variable has to be retained from the earlier execution and the execution retained.

The variables can be characterized by their ***data type*** and by their ***storage class***. One way to classify a variable is according to its data type and the other can be through its storage class. ***Data type*** refers to the type of value represented by a variable whereas ***storage*** class refers to the ***permanence*** of a variable and its scope within the program i.e. portion of the program over which variable is recognized.

Storage Classes

There are four different storage classes specified in C:

1.	Auto (matic)	2.	Extern (al)
3.	Static	4.	Register

The storage class associated with a variable can sometimes be established by the location of the variable declaration within the program or by prefixing keywords to variables declarations.

For example:
```
auto    int     a, b;
static  int     a, b;
extern  float   f;
```

8.6.1 Automatic Variables

The variables local to a function are automatic i.e., declared within the function. The scope of lies within the function itself. The automatic defined in different functions, even if they have same name, are treated as different. It is the default storage class for variables declared in a function.

Points to remember:

- The auto is optional therefore there is no need to write it.
- All the formal arguments also have the auto storage class.
- The initialization of the auto-variables can be done:
    - in declarations
    - using assignment expression in a function
- If not initialized the unpredictable value is defined.
- The value is not retained after exit from the program.

Let us study these variables by a sample program given below:

Example 8.4

/* To print the value of automatic variables */

```
# include <stdio.h>
main ( int argc, char * argv[ ])
{
int    a, b;
double  d;
printf("%d",   argc);
a = 10;
b = 5;
d = (b * b) – (a/2);
printf("%d, %d, %f", a, b, d);
}
```

All the variables a, b, d, argc and argv [ ] have automatic storage class.

8.6.2 External (Global) Variables

These are not confined to a single function. Their scope ranges from the point of declaration to the entire remaining program. Therefore, their scope may be the entire program or two or more functions depending upon where they are declared.

Points to remember:

- These are global and can be accessed by any function within its scope. Therefore value may be assigned in one and can be written in another.
- There is difference in external variable definition and declaration.
- External Definition is the same as any variable declaration:
    - Usually lies outside or before the function accessing it.
- It allocates storage space required.
- Initial values can be assigned.
- The external specifier is not required in external variable definition.
- A declaration is required if the external variable definition comes after the function definition.
- A declaration begins with an external specifier.
- Only when external variable is defined is the storage space allocated.
- External variables can be assigned initial values as a part of variable definitions, but the values must be constants rather than expressions.
- If initial value is not included then it is automatically assigned a value of zero.

Let us study these variables by a sample program given below:

Example 8.5

/* Program to illustrate the use of global variables*/

```
#  include <stdio.h>
int gv;                                    /*global variable*/
main  (  )
{
void function1();                    /*function declaration*/
gv = 10;
printf  ("%d is the value of gv before function call\n", gv);
function1( );
printf  ("%d is the value of gv after function call\n", gv);
}
```

```
void function1 ( )
{
gv = 15: }
```

OUTPUT

10 is the value of gv before function call
15 is the value of gv after function call

8.6.3 Static Variables

In case of single file programs static variables are defined within functions and individually have the same scope as automatic variables. But static variables retain their values throughout the execution of program within their previous values.

Points to remember:

- The specifier precedes the declaration. Static and the value cannot be accessed outside of their defining function.
- The static variables may have same name as that of external variables but the local variables take precedence in the function. Therefore external variables maintain their independence with locally defined auto and static variables.
- Initial value is expressed as the constant and not expression.
- Zeros are assigned to all variables whose declarations do not include explicit initial values. Hence they always have assigned values.
- Initialization is done only is the first execution.

Let us study this sample program to print value of a static variable:

Example 8.6

```
/* Program to illustrate the use of static variable*/

#include <stdio.h>

main()
{
int call_static();
int  i,j;
i=j=0;
j = call_static();
printf("%d\n",j);
j = call_static ();
printf("%d\n",j);
j = call_static();
printf("%d\n",j);
}

int    call_static()
{
static  int  i=1;
int   j;
j  =  i;
i++;
return(j);
}
```

OUTPUT

1
2
3

This is because ***i*** is a static variable and retains its previous value in next execution of function call_static( ). To remind you ***j*** is having auto storage class. Both functions main and call_static have the same local variable ***i*** and ***j*** but their values never get mixed.

8.6.4 Register Variables

Besides three storage class specifications namely, Automatic, External and Static, there is a *register* storage class. *Registers* are special storage areas within a computer's CPU. All the arithmetic and logical operations are carried out with these registers.

For the same program, the execution time can be reduced if certain values can be stored in registers rather than memory. These programs are smaller in size (as few instructions are required) and few data transfers are required. The reduction is there in machine code and not in source code. They are declared by the proceeding declaration by register reserved word as follows:

register int m;

Points to remember:

- These variables are stored in registers of computers. If the registers are not available they are put in memory.
- Usually 2 or 3 register variables are there in the program.
- Scope is same as automatic variable, local to a function in which they are declared.
- Address operator '&' cannot be applied to a register variable.
- If the register is not available the variable is though to be like the automatic variable.
- Usually associated integer variable but with other types it is allowed having same size (short or unsigned).
- Can be formal arguments in functions.
- Pointers to register variables are not allowed.
- These variables can be used for loop indices also to increase efficiency.

8.7 TYPES OF FUNCTION INVOKING

We categorize a function's invoking (calling) depending on arguments or parameters and their returning a value. In simple words we can divide a function's invoking into four types depending on whether parameters are passed to a function or not and whether a function returns some value or not.

The various types of invoking functions are:

- With no arguments and with no return value.
- With no arguments and with return value
- With arguments and with no return value
- With arguments and with return value.

Let us discuss each category with some examples:

TYPE 1: With no arguments and have no return value

As the name suggests, any function which ***has no arguments and does not return any values to the calling function***, falls in this category. These type of functions are confined to themselves i.e. neither do they receive any data from the calling function nor do they transfer any data to the calling function. So there is no data communication between the calling and the called function are only program control will be transferred.

Example 8.7

/* Program for illustration of the function with no arguments and no return value*/

/* Function with no arguments and no return value*/

```
#include <stdio.h>
main()
 {
void message();
printf("Control is in main\n");
message();                    /* Type 1 Function */
printf("Control is again in main\n");
 }

void message()
{
  printf("Control is in message function\n");
 }                            /* does not return anything */
```

OUTPUT

```
Control is in main
Control is in message function
Control is again in main
```

TYPE 2: With no arguments and with return value

Suppose if a function does not receive any data from calling function but does send some value to the calling function, then it falls in this category.

Example 8.8

Write a program to find the sum of the first ten natural numbers.

/* Program to find sum of first ten natural numbers */

```
#include <stdio.h>

int cal_sum()
 {
int i, s=0;
for (i=0; i<=10; i++)
s=s + i;
return(s);                    /* function returning sum of first ten natural numbers */
}

main()
 {
int sum;
```

```
sum = cal_sum();
printf("Sum of first ten natural numbers is % d\n", sum);
}
```

OUTPUT

Sum of first ten natural numbers is 55

TYPE 3: With Arguments and have no return value

If a function ***includes arguments but does not return anything***, it falls in this category. One way communication takes place between the calling and the called function.

Before proceeding further, first we discuss the *type of arguments or parameters* here. There are two types of arguments:

- Actual arguments
- Formal arguments

Let us take an example to make this concept clear:

Example 8.9

Write a program to calculate sum of any three given numbers.

```
#include <stdio.h>

main()
{
int a1, a2, a3;
void sum(int, int, int);
printf("Enter three numbers: ");
scanf ("%d%d%d",&a1,&a2,&a3);
sum (a1,a2,a3);    /*  Type 3 function */
}


/* function to calculate sum of three numbers */
void sum (int f1, int f2, int f3)
{
int s;
 s = f1+ f2+ f3;
printf ("\nThe sum of the three numbers is %d\n",s);
}
```

OUTPUT

Enter three numbers: 23 34 45
The sum of the three numbers is 102

Here f1, f2, f3 are ***formal arguments*** and a1, a2, a3 are ***actual arguments***.
Thus we see in the function declaration, the arguments are formal arguments, but when values are passed to the function during function call, they are actual arguments.

Note: The actual and formal arguments should match in type, order and number

TYPE 4: With arguments function and with return value

In this category two-way communication takes place between the calling and called function i.e. a function returns a value and also arguments are passed to it. We modify above Example according to this category.

Example 8.10

Write a program to calculate sum of three numbers.

```
/*Program to calculate the sum of three numbers*/

#include <stdio.h>
main ( )
 {
int a1, a2, a3, result;
int sum(int, int, int);
printf("Please enter any 3 numbers:\n");
scanf ("%d %d %d", & a1, &a2, &a3);
result = sum (a1,a2,a3);   /* function call */
printf ("Sum of the given numbers is : %d\n", result);
}

/* Function to calculate the sum of three numbers */
int sum (int f1, int f2, int f3)
 {
  return(f1+ f2 + f3);  /* function returns a value */
 }
```

OUTPUT

```
Please enter any 3 numbers:
3 4 5
Sum of the given numbers is: 12
```

8.8 CALL BY VALUE

So far we have seen many functions and also passed arguments to them, but if we observe carefully, we will see that we have always created new variables for arguments in the function and then passed the values of actual arguments to them. Such function calls are called ***"call by value".***

Let us illustrate the above concept in more detail by taking a simple function of multiplying two numbers:

Example 8.11

Write a program to multiply the two given numbers

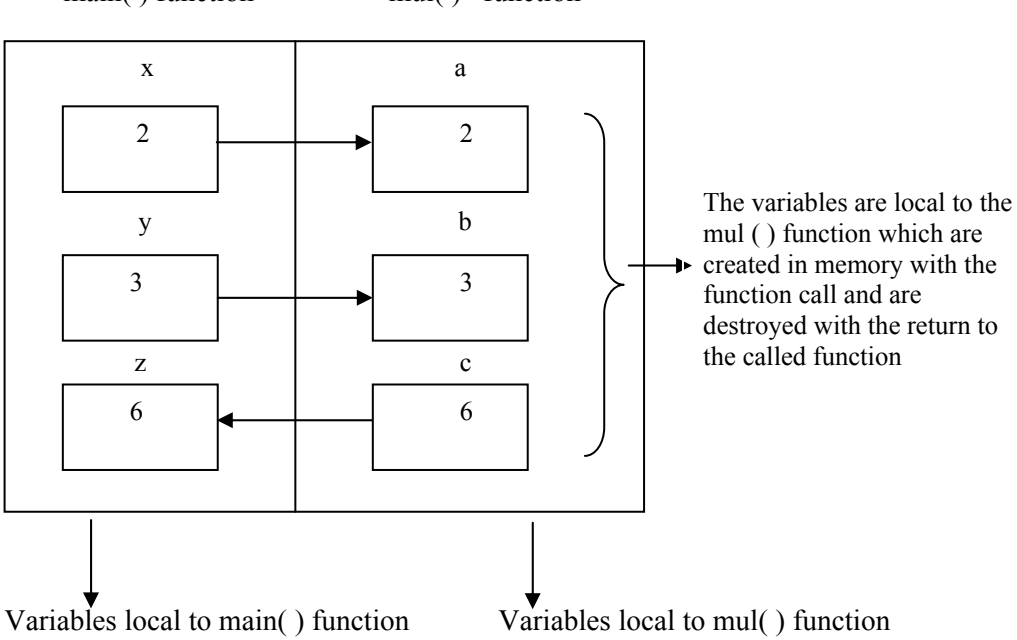
```
#include <stdio.h>
main()
{
int x, y, z;
int mul(int, int);
printf ("Enter two numbers: \n");
scanf ("%d %d",&x,&y);
z= mul(x, y);                /* function call by value */
printf ("\n The product of the two numbers is : %d", z);
}
```

```
/* Function to multiply two numbers */
int mul(int a, int b)
{
int c;
c =a*b;
return(c);    }
```

OUTPUT

```
Enter two numbers:
23  2
The product of two numbers is: 46
```

Now let us see what happens to the actual and formal arguments in memory.



What are meant by local variables? The answer is *local variables are those which can be used only by that function.*

Advantages of Call by value:

The only advantage is that this mechanism is simple and it reduces confusion and complexity.

Disadvantages of Call by value:

As you have seen in the above example, there is separate memory allocation for each of the variable, so unnecessary utilization of memory takes place.

The second disadvantage, which is very important from programming point of view, is that any changes made in the arguments are not reflected to the calling function, as these arguments are local to the called function and are destroyed with function return.

Let us discuss the second disadvantage more clearly using one example:

Example 8.12

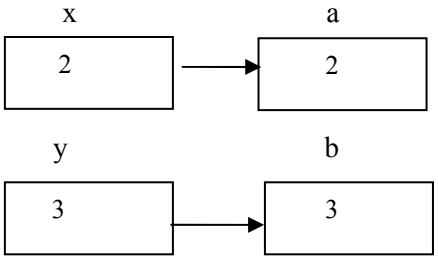
Write a program to swap two values.

```
/*Program to swap two values*/

#include <stdio.h>
main ( )
{
int x = 2, y = 3;
void swap(int, int);

printf ("\n Values before swapping are %d %d", x, y);
swap (x, y);
printf ("\n Values after swapping are %d %d", x, y);
}

/* Function to swap(interchange) two values */
void swap( int a, int b )
{
int t;
t = a;
a = b;
b = t;
}
```

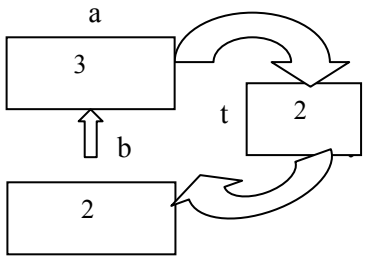
OUTPUT

Values before swap are 2 3
Values after swap are 2 3

But the output should have been 3 2. So what happened?



Values passing from main ( ) to swap() function



Variables in swap ( ) function

Here we observe that the changes which takes place in argument variables are not reflected in the main() function; as these variables namely a, b and t will be destroyed with function return.

- All these disadvantages will be removed by using *"call by reference"*, which will be discussed with the introduction of pointers in UNIT 11.

Check Your Progress 3

1. Write a function to print Fibonacci series upto 'n' terms 1,1,2,3,…..n
   ……
   ……
2. Write a function power (a, b) to calculate a^b
   ……
   ……
   ……

8.9 RECURSION

Within a function body, if the function calls itself, the mechanism is known as **'Recursion'** and the function is known as **'Recursive function'**. Now let us study this mechanism in detail and understand how it works.

- As we see in this mechanism, a chaining of function calls occurs, so it is necessary for a recursive function to stop somewhere or it will result into infinite callings. So the most important thing to remember in this mechanism is that every "recursive function" should have a terminating condition.

- Let us take a very simple example of calculating factorial of a number, which we all know is computed using this formula 5! = 5*4*3*2*1

- First we will write non – recursive or iterative function for this.

Example 8.13

Write a program to find factorial of a number

```
#include <stdio.h>
main ()
{
int n, factorial;
int fact(int);
printf ("Enter any number:\n" );
scanf ("%d", &n);
factorial = fact ( n);   /* function call  */
 printf ("Factorial is %d\n", factorial);
}

/* Non recursive function of factorial */

int fact (int n)
{
int res = 1, i;
for (i = n; i >= 1; i--)
res = res * i;
return (res);
}
```

OUTPUT

```
Enter any number: 5
Factorial is 120
```

How it works?

Suppose we call this function with n = 5

Iterations:

1. i= 5 res = 1*5 = 5
2. i= 4 res = 5*4 = 20
3. i= 3 res = 20*4 = 60
4. i= 2 res = 60*2 = 120
5. i= 1 res = 120*1 = 120

Now let us write this function **recursively**. Before writing any function recursively, we first have to examine the problem, that it can be implemented through recursion.

For instance, we know n! = n* (n – 1)! (Mathematical formula)

Or fact (n) = n*fact (n-1)
Or fact (5) = 5*fact (4)

That means this function calls itself but with value of argument *decreased by '1'*.

Example 8.14

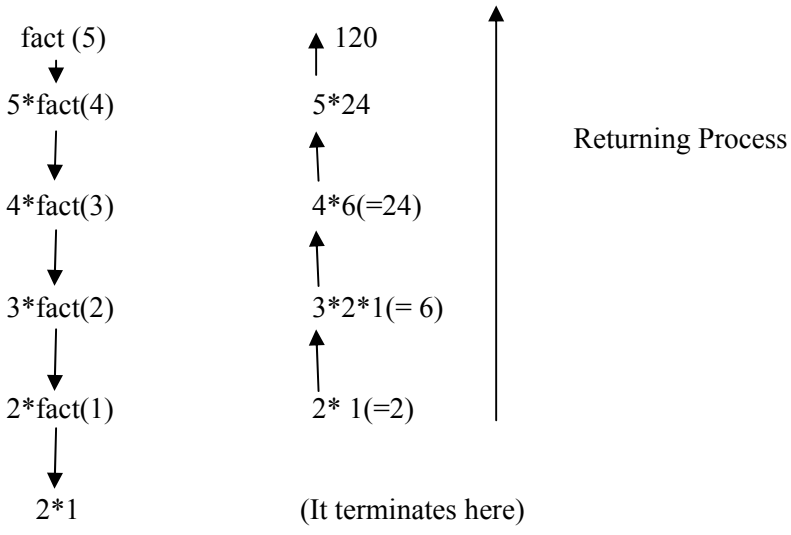
Modify the program 8 using recursion.

```
/*Program to find factorial using recursion*/
#include<stdio.h>
main()
{
int n, factorial;
int fact(int);
printf("Enter any number:  \n" );
scanf("%d",&n);
factorial = fact(n);          /*Function call  */
printf ("Factorial is %d\n", factorial);        }

/* Recursive function of factorial */
int fact(int n)
{
int res;
      if(n == 1)              /* Terminating condition  */
            return(1);
      else
             res = n*fact(n-1);       /* Recursive call */
            return(res);  }
```

OUTPUT
Enter any number: 5
Factorial is 120

How it works?

Suppose we will call this function with n = 5



Thus a recursive function first proceeds towards the innermost condition, which is the termination condition, and then returns with the value to the outermost call and produces result with the values from the previous return.

Note: This mechanism applies only to those problems, which repeats itself. These types of problems can be implemented either through loops or recursive functions, which one is better understood to you.

Check Your Progress 4

1. Write recursive functions for calculating power of a number 'a' raised by another number 'b' i.e. a^b

   ……

   ……

   ……

   …….....

8.10 SUMMARY

In this unit, we learnt about "Functions": definition, declaration, prototypes, types, function calls datatypes and storage classes, types function invoking and lastly Recursion. All these subtopics must have given you a clear idea of how to create and call functions from other functions, how to send values through arguments, and how to return values to the called function. We have seen that the functions, which do not return any value, must be declared as *"void"*, return type. A function can return only one value at a time, although it can have many return statements. A function can return any of the data type specified in 'C'.

Any variable declared in functions are local to it and are created with function call and destroyed with function return. The actual and formal arguments should match in type, order and number. A recursive function should have a terminating condition i.e. function should return a value instead of a repetitive function call.

8.11 SOLUTIONS / ANSWERS

Check Your Progress 1

1.
```
/* Function to multiply two integers */
int mul( int a, int b)
{
      int c;
       c = a*b;
      return( c );
}
```

2.
```
#include <stdio.h>
main ( )
{
 int x, y, z;
int mul (int, int);     /* function prototype */
printf ("Enter two numbers");
scanf ("%d %d", &x, &y);
z = mul (x, y);                  /* function call  */
printf ("result is %d", z);      }
```

Check Your Progress 2

1. (a) Valid
   (b) In valid
   (c) Valid
   (d) Valid
   (e) Invalid

Check Your Progress 3

1. /* Function to print Fibonacci Series */

```
void fib(int n)
   {
      int curr_term, int count = 0;
      int first = 1;
      int second = 1;
      print ("%d %d", curr_term);
      count = 2;
      while(count < = n)
        {  curr_term = first + second;
           printf ("%d", curr_term);
           first = second;
           second = curr_term;
           count++;
        }
    }
```

2. /* Non Recursive Power function i.e. pow(a, b) */

```
int pow( int a, int b)
    {
 int i, p = 1;
 for (i = 1; i < = b; i++)
 p = p*a;
 return (p);
        }
```

Check Your Progress 4

1. /* Recursive Power Function */

```
int pow ( int a, int b )
  {       if ( b = = 0 )
             return (1);
        else
             return (a* pow (a, b-1 ));     /*  Recursive call  */
   }

/*  Main Function  */
main ( )
      {
           int a, b, p;
           printf (" Enter two numbers");
           scanf ( "%d %d", &a, &b );
           p = pow (a, b);      /* Function call */
printf ( " The result is %d", p);
        }
```

8.12 FURTHER READINGS

1. The C programming language, *Brain W. Kernighan, Dennis M. Ritchie*, PHI
2. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, Tata McGraw Hill, 2002.
3. Computer Programming in C, *Raja Raman. V*, 2002, PHI.
5. C,The Complete Reference, Fourth Edition, *Herbert Schildt*, TMGH,2002.