PGCA/Sc IT S1 03

Computer Programming using C

SEMESTER-I

MASTER OF COMPUTER APPLICATION /

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

BLOCK-2

KRISHNA KANTA HANDIQUI STATE OPEN UNIVERSITY

Subject Experts

Prof. Anjana Kakati Mahanta,	Gauhati University
Prof. (Retd.) Pranhari Talukdar,	Gauhati University
Dr. Jyotiprokash Goswami,	Assam Engineering College
Prof. Sikhar Kumar Sarma,	Cotton University

Course Coordinator

Dr. Sanjib Kr. Kalita,	KKHSOU
Dr. Tapashi Kashyap Das,	KKHSOU
Sruti Sruba Bharali,	KKHSOU

SLM Preparation Team

Units	Contributors
8,14,15	Binod Deka, KKHSOU
9,10	Sruti Sruba Bharali, KKHSOU
11,13	Dr. Tapashi Kashyap Das, KKHSOU
12	Dr. Sanjib Kr. Kalita, KKHSOU

Editorial Team

Content	Dr. Tapashi Kashyap Das (Unit 9,10,12,15)
	Sruti Sruba Bharali (Unit 8,11,13,14)
Language	Prof. (Retd.) Robin Goswami, Cotton College

Structure, Format & Graphics: Dr. Tapashi Kashyap Das, KKHSOU

May, 2018

This Self Learning Material (SLM) of the Krishna Kanta Handiqui State Open University is made available under a Creative Commons Attribution-Non Commercial-Share Alike 4.0 License (international): http://creativecommons.org/licenses/by-nc-sa/4.0/

Printed and published by Registrar on behalf of the Krishna Kanta Handiqui State Open University.

Headquarters : Patgaon, Rani Gate, Guwahati - 781017
Housefed Complex, Dispur, Guwahati-781006; Web: www.kkhsou.in

The University acknowledges with thanks the financial support provided by the **Distance Education Bureau**, **UGC** for the preparation of this study material.

MASTER OF COMPUTER APPLICATION /

MASTER OF SCIENCE IN INFORMATION TECHNOLOGY

Computer Programming using C

DETAILED SYLLABUS

Block-2

BLOCK INTRODUCTION

This is the *second block* of the course '***Computer Programming using C***'. This block aims at acquainting learners with writing complete programs in C to handle real-world problem.

This block comprises of the following eight units:

Unit - 8 describe the various storages classes used in C language.

Unit - 9 is on arrays. In this unit we will discuss how arrays are defined, declared, initialized, accessed, etc. This unit will also discusses the types of arrays.

Unit - 10 discusses strings and some important string handling functions in C language.

Unit - 11 discusses the most important concept function. Library functions or built-in functions are already discussed in *Unit 5 of Block 1.* This unit discusses user-defined function which enable programmers to break up a program into small segments, each of which can be written more or less independently of the other.

Unit - 12 discusses the concept of pointers. Pointers are used to directly access memory using memory addresses and form a very powerful concept in the C language. We will learn about the basics of pointers and the uses of pointer to dynamically allocate memory.

Unit - 13 is on structure and union. A structure as well as union extends the concept of arrays by storing related information of different data types together under a single unit. In this unit we will see how structure and union are defined, declared and accessed in the C language.

Unit - 14 discusses file handling. We will learn about files which are very important for storing information permanently.

Unit - 15 is on preprocessor directives. Macro and few important preprocessor directives are discussed in this unit.

The structure of Block 2 is as follows:

UNIT 8: STORAGE CLASS

UNIT STRUCTURE

8.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- learn about Storage Class
- describe Automatic, External, Static and Register Variable
- describe the Scope of Variables
- define lifetime of a Variable

8.2 INTRODUCTION

We already have some basic idea about variables. Variables can be defined as the memory location where we can store the values of a particular data type. The value stored in the variable may be changed during the program executions.

Every C variable has a storage class and a scope. This storage class determines the part of memory where storage is allocated for an object and how long the storage allocation continues to exist during the execution of program.The storage class also determines the scope which specifies the part of the program over which a variable name is visible, i.e. the variable is accessible by name. In this unit, we will discuss the various storage classes.

8.3 STORAGE CLASS

Storage class is related to the declaration of variable, function, and parameters that we use in C programs. The storage class in the function is used when returning a value of a particular data type from the function. The storage class specifier used within the declaration determines whether:

- the object is to be stored in memory or in a register.
- the object receives the default initial value or an indeterminate default initial value.
- the object can be referenced throughout a program or only within the function, block,or source file where the variable is defined.

Here, the term 'object' refers to variable, function and parameters in which storage class is going to be used. Depending upon the above, storage class can be classified into four categories:

i) Automatic
ii) Register
iii) Static
iv) External

Now, let us try to understand each storage class in the next section using examples.

8.4 AUTOMATIC VARIABLE

We already know how variables are used in C program. Now, we are going to use storage class in the variable declarations. Actually, we have already used automatic storage class in the programs in the previous units. Let us take an example as shown below:

Program 8.1: Program to illustrate the use of automatic variable.

```
void main()
{
   int a,b,s;   // or we can write here as auto
```
`int a,b,s;`

```
   scanf("%d %d", &a,&b);
   s=a+b;
   printf("Sum is %d",s);
}
```

By default all the variables declared are automatic.We can also explicitly define the variables using **auto** keyword.

Let us look at the special properties of automatic storage that make it different from other storage class. The automatic variable has the following characteristics:

a) **Storage:** The value of the variable is stored in the memory of the computer (not in register).

b) **Default intial value:** The default initial value for automatic variables is garbage value i.e. any unpredictable value. It means that if the variable is not initialized then the variable contains some useless value.

c) **Scope:** The scope means the availabilty of the variable. Automatic variable is local to the block in which the variable is declared. Outside this block the variable can not be accesed.

d) **Life time:** The life time of this variable is within the block it is declared.

Program 8.2: Program to illustrate the use of automatic variables.

```
void  main ()
{
   auto int i,j;
   i=10;
   printf("i= %d \n j= %d",i,j);
}
```

Output:
```
i=10
j=8214 (or some other garbage value)
```

Since here we initialized **i** to 10 explicitly, therefore the value of **i** is displayed as 10; But in case of **j,** as we do not assign any value so garbage value is displayed at output. This example describes about the default intial value of auto variables. Now, let us understand about the life time and scope of automatic variables using another example.

Program 8.3: Program to show lifetime and scope of automatic variables.

```
void main ()
{
   auto int i=1;
      {
         auto int i=2;
      {
         auto int i=3;
         printf(“%d”,i);
      }
   printf(“%d”,i);
   }
   printf(“%d”,i);
}
```

Output:
```
3
2
1
```

The program has three blocks and each block initializes the value of **i**. Note that variable **i** allocates extra memory for each declarations and each **i** is different from one another. The first inner block is executed and therefore 3 is displayed as the first output as in this block **i=3.** After that the second inner block is executed and 2 is displayed as in the second inner block the value of **i** is 2**(i=2)**. Next, we come to the outer block where the value of **i** is 1**(i=1)** so 1 is displayed as the last output. We have seen that the value of **i** is different in each block.Whatever value we have initialized **i** with, it remains valid only within the block where we have declared it. This is known as **scope** of the variable.

8.5 EXTERNAL VARIABLE

We have now learnt about automatic variable which is local to the block in which it is declared.But sometimes we need a variable which should be available to all the functions and blocks within the program. External storage class fits this need. The properties of the external variable are:

a) **Storage:** The external storage variables are stored in memory.

b) **Default initial value:** The default initial value for external variables is zero.

c) **Scope:** The scope for external variables is global i.e., the variable is available to all funcitons and blocks within the program.

d) **Life time:** The lifetime of the variables is until the program's execution stops.

The main difference between automatic and external variable is in the scope and life time of the variables. They have similarities in storage and default initial value. External variables are declared outside all functions. Let us understand the scope and life time of external variables using examples.

Program 8.4: Program to illustrate the concept of external variables.

```
int var;   // external variable
void main()
{
   printf("%d",var);  //  just print the value of
'var'
   var_add_two();     // add 2 to var and display
it
   var_sub_one();
   }
   void var_add_two()
      {
         var=var+2;
         printf("\n%d",var);
```

```
        }
        void  var_sub_one()
        {
    var=var-1;
    printf("\n%d",var);
}
```

Output: 0

2

1

In the above pogram, the first output is 0 since default initial value of external variable is 0.Next we increment 'var' by 2, so the next output is 2 and after that we decrement the value of 'var' by one so the output is 2-1 i.e. 1. Note that, here the value of 'var' is visible to the funcitons var_add_two() and var_sub_one() each of which modifies value of 'var'.

CHECK YOUR PROGRESS

Q.1: Write true or false:

a) The default initial value of external variable is same as with automatic variable.

b) By default the variables declared are automatic.

c) The storage for the automatic variable is in the registers.

Q.2: Write down one similarity between automatic and external variables.

8.6 STATIC VARIABLE

Static variable is similar to the automatic variable. Like the automatic variables static variables are local to the block in which they are declared.The difference between them is that for static variables the value does not disappear when the function is no longer active.The last updated value for static variable always persists. That is, when the control comes back to the same function again, the static variables have the same value as they left at the last time. Properties of static variables are:

a) **Storage:** The static variables are stored in memory.

b) **Default initial value:** The default initial value for static variables is zero.

c) **Scope:** The scope of static variables is global.

d) **Life time:** The life time of static variables is untill the program's execution does not stop.

Program 8.5: Program to illustrate the concept of static variable.

```
void  add_one();
void main()
{
   add_one();
   add_one();
}
void add_one()
{
   static int  var=3;
   var=var+1
   printf("\n %d",var);
}
```

Output: 4
5

It can be observed that the output of the above program is 4 and 5. Since we have initialized 'var' with 3 and then increment 'var' by 1 so the first output is 4 during the first call of the add_one() function. The variable 'var' retains its previous value 4 and thus in the second call of the add_one() function, increments 4 by 1; thus the output is 5. If we write the above function defination as:

```
void  add_one()
{
   int  var=3;  //or auto int var=3;
   var=var+1
   printf("\n %d",var)
}
```

The output of the program after the modifications should be:

```
4
4
```

The reason behind this is that the automatic storage class variable does not retain its previous value.Whenever the add_one() function is called 'var' has always initialized value 3 and then increment by 1; so the output is always 4.

8.7 REGISTER VARIABLE

We have discussed about the automatic, static, external storage classes in earlier sections. Each class stores the variables in the memory of the computer. We know that there are mainly two areas for storing data in the computer– Memory and CPU registers. Accessing data from the register is faster than from memory. This makes the program to run faster. Register storage class makes it possible for the program to run faster. We use the register class with variables which accessed frequently like loops (such as for, do-while etc.).

The characteristics of register variables in terms of scope, life time etc are:

- **a)** **Storage:** The register variables are stroed in CPU registers.
- **b)** **Default initial value:** The register variables take garbage values as the default values.
- **c)** **Scope:** The scope of register variables is local to the block in which it is declared.
- **d)** **Life time:** The life time of register variables is till the control remains within the particular block where it is declared.

Program 8.6: Program to illustrate the concept of register storage class variables.

```
void  main()
{
   register int var;
   for(var=1; var<=10; var++)
```

```
        printf("%d",var);
    }
```

A question that may arise in your mind here, is that if we declare most of the variables as register then every program should run faster. But then why do we not always use this concept. This is because the number of registers is limited in a computer system and so we can not use register class for all variables. If in a program the total number of register variables exceeds the system register quantity; then all the variables that exceed are by default declared as automatic.

For example, if we write a program with a loop that uses 20 register variables (assume) and the computer has only 16 CPU registers, then the rest of the 4 variables are automatically transformed to automatic storage class variables. A important point to be noted here, is that register storage can not be used for float and double data type since CPU registers usual capacity is 16 bit and both float and double data types require 4 byte (4 x 8 = 32 bit) and 8 byte (8 x 8 = 64 bit) storage respectively.

CHECK YOUR PROGRESS

Q.3: Identify from the following which statements are true:

a) Default initial value of register storage class variable is zero.

b) Scope of external and register variables are not same.

c) There is no limit of using register variable.

Q.4: Write one difference between register and auotmatic storage class variable.

8.10 LET US SUM UP

- Storage class of a variable determines the variable location, default intial value,scope and life time of the variable.
- By default all the variables declared are aotomatic in nature.
- Except for the register storage class, all other storage class variables are located in the memory.

- Static storage variables retain their previous value during program executions. Static class variable are usually used when the program needs to share a variable.
- Since locations of register storage class variables are in the CPU registers, so those programs with register variables run faster than the programs where other class variables are used.
- The number of CPU registers is limited in a computer system so we can not use register class for all variables.

8.11 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.

2) Gottfried Byron, S; *Programming with C*; Tata McGraw-Hill Education.

8.12 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) False, b) True, c) False

Ans. to Q. No. 2: One similarity between automatic and external variables is that the storage location for both class variables are in the memory.

Ans. to Q. No. 3: a) False, b) True, c) False.

Ans. to Q. No. 4: Register class variable stores value in CPU registers whereas automatic class variable use memory for storing data.

8.13 MODEL QUESTIONS

Q.1: Define storage class of a variable. What are the different types of storage class?

Q.2: Compare external and automatic storage class variables.

Q.3: Explain the static and automatic storage class variable with examples.

Q.4: Why are register storage class variables used with loop counter variables?

Q.5: Mention the common factor among automatic, static and external variables.

*** ***** ***

UNIT 9: ARRAY

UNIT STRUCTURE

9.1 LEARNING OBJECTIVES

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

- define an array
- declare and initialize array
- create and access one dimensional array
- create and access two dimensional array.

9.2 INTRODUCTION

In the previous unit, we have learnt about the different storage class. We have also learnt about conditional statements and loop control structures in the earlier units.

You must have come across the term ***array*** many times and wondered what it is and where it can be applied.

In this unit we will learn to define an array. We will also learn to declare an array and to initialize an array. In addition to these, different types of arrays like one dimensional and two dimensional arrays will also be covered in this unit.

9.3 ARRAY

Array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations. In this definition by 'finite', we mean that the array contains a fixed number of elements. By 'ordered' we mean that all the elements are stored in contiguous locations of the computer memory in a linear way. By 'homogeneous' we mean that all the elements in the array must belong to the same data type.

9.3.1 Terminology

Let us look at some of the terminologies associated with array.

- **Size:** The number of elements in the array.
- **Type:** The type indicates the data type of the array. It can be integer, floating point or character.
- **Base:** The base of the array is the address of the first element of the array.
- **Index:** Elements in an array are referred using a subscript or index value. The index is an integer value which gives the position of the element in an array. It is denoted by $\mathbf{A_i}$ or **A [i]** where **A** is the name of the array and **"i"** is the subscript or index. Since array elements are identified by using index or subscripts, the array is also called an indexed or subscripted variable.

9.4 ARRAY DECLARATION AND INITIALIZATION

Declaration of Array: An array can be declared just like we declare any other variable. We give the data type of the variable followed with the name of the variable. In case of an array also we give the data type followed with the name of the array but we also include an additional component that is the size of the array. The number of elements that the array can contain needs to be declared at the beginning of the array. This is because according to the definition of an array, it contains a fixed number of elements.

We can declare an array in the following way:

```
data_type    name_of_array [ size_of_array ];
```

In the above declaration syntax, data type can be any of the valid data types for variables. The data type can be integer, floating point or a character. It is followed by the name of the array variable and the size of the array that is given inside square brackets. For example:

```
int     array [ 5 ] ;
char    arr [ 10 ] ;
float   balance [ 3 ] ;
```

Need for arrays: To stress the need for arrays let us look at the following example.

A cricket match has been organized between two teams A and B. Suppose we do not have the knowledge of arrays and we need to keep the batting scores of all the players for the team batting first. If we want to keep the batting score for one player we can declare a variable like the one below:

```
int  bat_score1;
```

where we store the score of player 1. To store the score of rest of the 10 players we will need to declare 10 more variables of data type integer as given below:

```
int bat_score2, bat_score3,        ….       bat_score11;
```

Now if, instead of keeping the scores of just 11 players for one match suppose we are required to keep score of 3 teams. How many variables do we create using the above approach? Do we create 33 integer variables?

An easier way to keep the scores of the team would be to use an array. Instead of declaring 11 variables for keeping the score of 11 players in a team wouldn't it be easier to store all the batting values of one team for one match in one array variable.

```
int    team_a [ 11 ];
int    team_b [ 11 ];
int    team_c [ 11 ];
```

Definition of array: When we declare an array, a contiguous memory location is allocated to that array. Let us look at an example of the physical representation of array of size 10 in computer's memory.

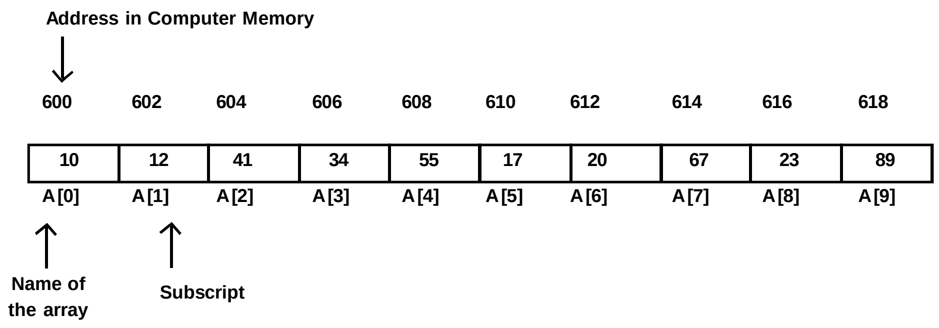


Fig. 9.1 Physical representation of array in memory

In the above figure we can see how array is actually stored in the memory. Let us consider an integer array of ten elements. Here, the name of the array is A. The smallest index of an array is called a lower bound and the highest index is called an upper bound. In case of C, the lower bound of an array is 0 and the number of elements can be calculated as the difference between upper and lower bound plus 1.

No of elements = Upper – Lower + 1

The number of elements in the above case will be (9-0+1=10) ten elements. The base address of the array in this case is 600, since that is the address of the first element in the array. If we assume that the compiler to store an integer value needs two bytes of storage then the address of the second element is 602. Similarly, the rest of the elements are also stored continuously taking two bytes of storage per integer number.

Array initialization: We can initialize the elements of the array when we declare the array. Initialization is generally done when we already know the values of the elements of an array. Just as in declaration, in initialization also we give the data type of the array, followed by the name and size of the array. But in addition to these we also provide the values of the data elements of the array within braces **{ }** separated by commas. Let us look at how initialization is done using the following example:

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

In the above example, the array A is declared and initialized. The array A has a size of 5 elements and the values of the five elements have been initialized as follows:

A [0] = 1

A [1] = 2

A [2] = 3

A [3] = 4

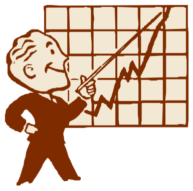
A [4] = 5

The first value that is given in braces is put into the first array location with subscript value 0. So 1 is kept at location A[0]. Similarly, the second value inside braces is put into the second array location with subscript value A[1] and so on till all the given values has been put in the array.

When initialization of values is done, a possibility of not mentioning the array size in the square brackets [ ] beforehand is also provided. That is, we can leave the square brackets empty. For this case the compiler assumes the array size equal to the number of values provided inside braces { }. For example, we can write the following statement:

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

In the above statement, the compiler will assume that the size of the array is 6 since six data values have been given inside the braces.



CHECK YOUR PROGRESS

Q.1: What is an array?

Q.2: Give the syntax for declaring an array.

Q.3: Arrays cannot store elements of __________ data types.

Q.4: Can we declare an array without mentioning the size of an array?

9.5 ONE DIMENSIONAL ARRAY (1-D ARRAY)

One dimensional array is a collection of homogeneous data elements with only one row. It is the simplest form of array. The declaration and definition that we have learnt so far has been for a single dimensional array.

Address calculation: The elements of an array are stored in contiguous memory locations. This means that if we know the base address then we can calculate the address of the other array elements.

Let ‘b’ be the memory location of the first element of the array and each element requires ‘w’ words of memory space. Then, the address or location of element A[i] will be the summation of the base address and the product value of ‘i’ and ‘w’.

Address of element A[i] = b + i X w

For example if we consider the figure 9.1, then the base address, b = 600. Now if we consider an integer array that requires word size 2, the address of the 10th element should be 618. If we calculate the address according to the formula given above we get the same answer.

Address of element A[9] = 600 + (9 X 2) = 600 + 18 = 618

9.5.1 Entering Data Values in 1-D Array

We can insert values into an array at the time of initialization. But apart from that, we can also insert values into array elements by accessing them individually. For example, if we have an integer array A of 5 elements, we can insert the values for each of the five array positions as follows:

```
A [0] = 11;
A [1] = 22;
A [2] = 33;
A [3] = 44;
A [4] = 55;
```

In the above statements, the value for each array position has been filled individually instead of giving the values at the time of initialization and declaration.

Let us now suppose that we have an array of 50 elements. Also we can write statements like above only when we have the knowledge of the element values in advance. Otherwise when we have to take the values from the user at run time these types of statements do not provide a suitable solution.

An easier solution to the above problem is to use loop control structure statements like ***while***, ***do_while*** and ***for*** loop. Let us look at a code where with the help of for loop we can easily enter any number of values at the run time. This can be implemented as:

```
for ( i = 0; i < 5;  i++ )
{
   scanf("%d", &A[ i ]);
}
```

In the code above a 'for loop' is used to traverse from the first element of the array to the last element. The above 'scanf' statement accepts an integer value and stores it in the address of the given array location. The array location is moved from the first element position to the last element position using for loop. "&A[i]" gives the address of the 'ith' element of the array where the current data value needs to be stored.

9.5.2 Accessing Values from 1-D Array

We have learnt in the previous part about entering values in a one dimensional array. Once the values have been inserted we may need to access the array for various purposes. The procedure for accessing the values in a one dimensional array is similar to entering the values in one. This can be implemented as:

```
for ( i = 0; i < size;  i++ )
{
printf("%d", A[ i ]);
}
```

In the above code, a 'for loop' is used to traverse from the first element of the array to its last element. The 'printf' statement prints the integer value which is stored in the address of the given array location. The array location is moved from the first element position to the last element position using for loop. "**A[i]"** gives the value of the element at the **ith position** of the array.

Program 9.2: **Program to enter and access data elements in a 1-D array**

```
# include<stdio.h>
# include<conio.h>
int main()
{
   // declaring integer array of size 10
   int array[10];
   int i;
   // Entering data values in array
   printf(“Enter any 10 integer values:\n”);
   for(i=0;i<10;i++)
   {
      scanf(“%d”,&array[i]);
   }
   // Traversing and printing data values from
array
   for(i=0;i<10;i++)
   {
      printf(“array[%d]    element    is
%d\n”,i,array[i]);
   }
getch();
return 0;
}
```

9.6 TWO DIMENSIONAL ARRAY (2-D ARRAY)

Two- dimensional arrays are collection of homogenous data elements where the elements are ordered in number of rows and columns. A two dimensional array can be declared just as we declare a one dimensional array variable. We give the data type of the array variable followed with the name of the array variable and the size of the array in square brackets. In case of a two dimensional array also we give the data type followed with name and size of the array but we add another value in brackets to give the second size of the array. The first size represents the row size and the second size represents the column size of the array. This is because according to the definition of a two dimensional array the array is organized in rows and columns.

Declaration of 2-D Array: We can declare a two dimensional array in the following way:

data_type name_of_array [row_size] [column_size];

In the above declaration syntax, data type can be any of the valid data types for 1-D arrays. The data type can be integer, floating point or a character. It is followed by the name of the array variable and the row size and column size of the array that are given inside different square brackets. For example:

```
int     array [ 5 ] [ 5 ];
char    arr [ 10 ] [ 3] ;
float   balance [ 2 ] [ 4 ] ;
```

When we declare an array, a contiguous memory location is allocated to that array. Let us look at an example of the physical representation of a two dimensional array of size [3][4] in computer's memory.

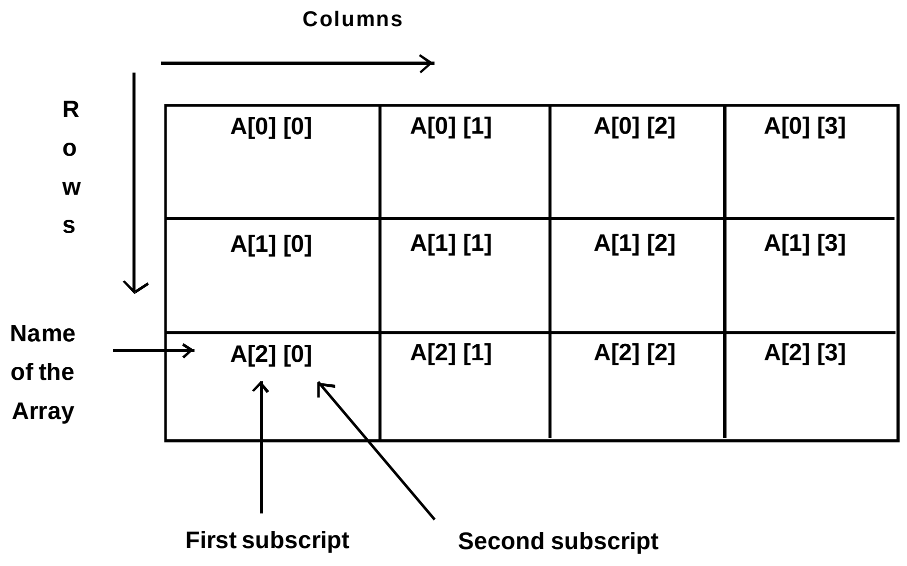


Figure 9.2: Representation of two-dimensional array A of size [3][4]

In the above figure, we can see that array A is a two dimensional matrix with three rows and four columns. Two subscripts are needed to represent an element in a two dimensional array. The first subscript represents the row value while the second subscript represents the column value of the array. In C language, similar to single dimensional arrays, the subscripts value for both rows and column values starts from 0. If we want to represent the first element in the array we denote it by **A[0][0],** where the first subscript value 0 means that the element is located in the first row and the second subscript value 0 means that the element is located in the first column. Since we need two subscripts to uniquely represent any element in this type of array, hence it is known as two dimensional arrays.

9.6.1 Storage Representation of 2-D Arrays

Two dimensional arrays can be stored in two different ways. They can be stored in either row-major order or in column-major order.

Row-major order: In row-major order, the elements are stored on a row-by-row basis. Here, the first row is filled first followed by second row and so on. This is the common way of storing elements generally for 2-D array. Let us try to understand this concept using an example. Suppose we need to fill the following 9 element values { 1, 2, 3 4, 5, 6, 7, 8, 9 } and in an array B[3][3].

Then, we will first fill up the first row of the array using the first three element values 1, 2, and 3. Once the first row is filled, we then proceed to fill up the rest of the rows one at a time using the rest of the data values. The filled array will have the following structure:

B =

1	**2**	**3**
4	**5**	**6**
7	**8**	**9**

Column-major order: In column-major order, the elements are stored in a column-by-column basis. Here, the first column is filled first followed by second column and so on. Let us use the earlier example itself to understand this concept. We need to fill the following 9 element values {1, 2, 3 4, 5, 6, 7, 8, 9} and in an array B[3][3].

We will first fill up the first column of the array using the first three element values 1, 2, and 3. Once the first column is filled, we will then proceed to fill up the rest of the columns one at a time using the rest of the data values. The filled array will have the following structure:

B =

1	**4**	**7**
2	**5**	**8**
3	**6**	**9**

Address calculation: The address calculation for elements in two-dimensional arrays will depend on whether the elements are stored in row-major order or in column-major order. Let '**b'** be the base address or the address of the first element in the array and **'w'** be the word size. Then the address of an element **B[i] [j]** when the array has a maximum row and column size **B [m] [n]** for both the cases can be calculated using the formulas below.

- For row-major, the address of element

  B [i] [j] = b + (i × n + j) × w

  where, i represents the row value,
  j represents the column value and
  n represents the column size.

- For column-major, the address of element

  B [i] [j] = b + (j × m + i) × w

  where, i represents the row value,
  j represents the column value and
  n represents the column size.

9.6.2 Entering Data Values in 2-D Array

We can insert values into a 2-D array at the time of initialization. Insertion of data elements in a 2-D array is similar to insertion of element in 1-D arrays. Let us look at the example below.

```
int  B [3] [4] =     {    { 1, 2, 3, 4 },
                     { 5, 6, 7, 8 },
                  { 9, 10, 11, 12 }
              };
```

In the above initialization, the data elements of the first row are put together inside braces, followed by data elements of second row inside second braces and so on till the last row. All these individual braces representing each row are then put inside one common brace to indicate that all rows belong to the same array.

We can also insert values into array elements by accessing them individually. For example, if we have an integer array B [2][3] of 6 elements, we can insert the values for each of the 6 data elements using their array positions as follows:

B [0] [0] = 11;
B [0] [1] = 22;
B [0] [2] = 33;
B [1] [0] = 44;
B [1] [1] = 55;
B [1] [2] = 66;

In the above statements, the value for each array position in the 2-D array has been filled individually instead of giving the values at the time of initialization.

Another easier way to insert data values into elements is to use loop control structure statements like *while*, *do while* and *for* loop. Let us look at a code where with the help of nested for loop we can easily enter the data values for the array B [2] [3] without initialization. This can be implemented as:

```
for ( i = 0; i < 2;  i++ )
{
   for ( j = 0; j < 3; j++ )
   {
      scanf("%d", &B[ i ] [ j ]);
   }
}
```

In the above code the first '*for loop*' is used to traverse through the rows and the second for loop is used to traverse through the columns. For each iteration of the first *for loop*, the second *for loop* traverses from the first to the last column in the array. "**B[i][j]"** gives the address of the element at the "**i**th" **row** and **"j**th" **column** of the array where the current data value needs to be stored.

9.6.3 Accessing Values from 2-D Array

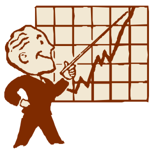
We have learnt the different ways of inserting values in a two dimensional array. Once the values have been inserted we may need to access the array for various purposes. The procedure for accessing the values in a two dimensional array is similar to entering the values. This can be implemented as:

```
for( i = 0; i < row_size;  i++ )
{
   for ( j=0; j< column_size; j++ )
   {
      printf("%d", B[ i ][ j ]);
   }
}
```

In the above code we have used a nested '*for loop*' to traverse the two dimensional array. The first '*for loop*' is used to traverse through the rows and the second *for loop* is used to traverse through the columns. For each iteration of the first *for loop*, the second for loop traverses from the first to the last column in the array. The 'printf' statement prints the integer value which is stored in the address of the given array location. "**B[i][j]"** gives the address of the element at the "**i**th" **row** and **"j**th" **column** of the array where the current data value needs is stored.

Program 9.2: **Program to enter and access data elements in a 2-D array**

```
# include<stdio.h>
# include<conio.h>
void main()
{
   int A[10][10], row_size, col_size, i, j;
   clrscr();
   printf("Give the number of rows you want : ");
   scanf("%d",&row_size);
   printf("Give the number of columns you want :
");
   scanf("%d",&col_size);
   // Entering elements in 2D array
   printf("\n Enter elemnts in an array\n");
   for(i=0;i<row_size;i++)
   {
      for(j=0;j<col_size;j++)
      {
         scanf("%d",&A[i][j]);
      }
   }
   // Accessing and printing elements in 2D array
   printf("\n The elements in the array A are:\n");
   for(i=0;i<row_size;i++)
   {
      for(j=0;j<col_size;j++)
      {
         printf("A[%d][%d] = %d\n",i,j,A[i][j]);
      }
      printf("\n");
   }
   getch();
}
```



CHECK YOUR PROGRESS

Q.5: The base address of a char array C[15] is 210. What is the memory address of the C[14]?

Q.6: Find the memory address of D[4] [8] for an array D[10][15] and when the word size is 4 and base address is 100.

9.7 LET US SUM UP

- Array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations. In this definition, by 'finite' we mean that the array contains a fixed number of elements.
- By 'ordered' we mean that all the elements are stored in contiguous locations of the computer memory in linear way. By 'homogeneous' we mean that all the elements in the array must belong to the same data type.
- Elements in an array are referred to as using a subscript or index value.
- An array is declared in the following way:

  data_type name_of_array [size_of_array];
- One dimensional array is a collection of homogeneous data elements with only one row.
- The address or location of element A[i] will be the summation of the base address and the product value of 'i' and 'w'.

  Address of element A[i] = b + i X w
- Two-dimensional arrays are collection of homogenous data elements where the elements are ordered in number of rows and columns.
- Two dimensional array is declared in the following way:

  data_type name_of_array[row_size][column_size];
- In row-major order, the elements are stored on a row-by-row basis.
- In column-major order, the elements are stored in a column-by-column basis.

9.8 FURTHER READING

1) Kanetkar, Y. P. (2008); *Let us C*; Jones and Bartlett Publishers, Inc.
2) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
3) Thareja, R. (2015); *Introduction to C Programming*.

9.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: An array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations.

Ans. to Q. No. 2: The syntax for declaring an array is:

data_type name_of_array [size_of_array];

Ans. to Q. No. 3: Arrays cannot store elements of <u>different</u> data types.

Ans. to Q. No. 4: Yes, arrays can be declared without mentioning the size provided we initialize the array with the value of the elements.

Ans. to Q. No. 5: In this case, b=210, i=14 and w=1(since character data types takes a byte size of 1 for storage in C).

So, the address of C [14] $= b + (i \times w)$

$= 210 + (14\times1)$

$= 224$

Ans. to Q. No. 6: Let us assume that row-major storage representation in used in this case. Then in this case, b =100, i= 4, j=8, n= 15, w=4

So the address for D [4] [8] $= b + (i \times n + j)$

$= 100 + (4\times15 + 8) \times 4$

$= 100 + 68 \times 4$

$= 100 + 272$

$= 372$

9.10 MODEL QUESTIONS

Q.1: Define array.

Q.2: What is row major order and column major order of representation?

Q.3: Write a program to input a one dimensional array of 10 elements. Also write a function to print the elements on computer screen.

Q.4: How is address translation done in case of 1-D arrays?

Q.5: Write a program to input a two dimensional array **A[5][3]**. Also write a function to print the elements on computer screen.

Q.6: How is address translation done in case of 2-D arrays?

Q.7: What do you mean by multi-dimensional arrays? Can there be arrays of more than two dimensions?

Q.8: Why do we need arrays?

*** ***** ***

UNIT 10: STRINGS

UNIT STRUCTURE

10.1 LEARNING OBJECTIVES

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

- define a string
- declare and initialize a string
- use string handling functions like *strlen()*, *strcmp()*, *strcpy()*
- use string handling functions like *strrev()* and *strcat()*

10.2 INTRODUCTION

In the previous unit, we have learnt about arrays and its types. In this unit we will learn about array of characters i.e., string. We will learn to define, declare and initialize string. In addition to this different string handling functions will also be discussed in this unit.

10.3 STRINGS

An array is a collection of homogeneous elements that are finite and ordered. Strings are also arrays but of character data type. Let us now discuss the concept of strings.

An array of characters is called a ***string***. In other words, strings are arrays where the data type is character. Arrays can be one dimensional or multidimensional. But strings are one-dimensional array of characters which are terminated by a **null character** represented by **'\0'**. Every string contains one or more characters that comprise the string followed by a null character '\0' that indicates the end of the string.

10.3.1 String Declaration and Initialization

Declaration of String: A string can be declared just as we declare any other array variable. We give the data type of the variable followed with the name and size of the variable. In case of a string, we give the data type as character followed with the name of the string and the size of the string. The size of the string should be the sum of the size of a number of elements that the string can contain plus the size for the null character that designates the end of the string. So, to hold the null character at the end of the string, the size of the string should be one more than the number of characters intended to enter in the string.

We can declare a string in the following way:

```
char  name_of_string [ size_of_string ];
```

In the above declaration syntax, data type has to be a character. It is followed by the name of the string variable and the size of the string that is given inside square brackets. For example:

```
char   array [ 5 ] ;
char   name [ 10 ] ;
char   book [ 30 ] ;
```

Need for Strings: To stress the need for strings let us look at the following example:

A cricket match has been organized between two teams A and B. Suppose we do not have the knowledge of string arrays and we need to keep the name of the best player. If we use a character variable then we will be able to store only one character of the player's name.

char name1;

Let us assume that name of the player is "Rahul". To store this name do we declare five char variables to store the five characters in the name? This as shown below does not solve our problem.

char name1, name2, name3, name4, name5;

An easier way to keep the scores of the team would be to use a character array or a string. Instead of declaring many variables for keeping the name of one individual we can store the name in one string variable.

char best_player [6];

Here, we declare a character array of size six since the first five locations will store the name "Rahul" and the 6th location of the array will store the null character.

Definition of Strings: When we declare a string, a contiguous memory location is allocated to that string. Let us look at an example of the physical representation of string of size 6 which contains the string "Hello" in computers memory.

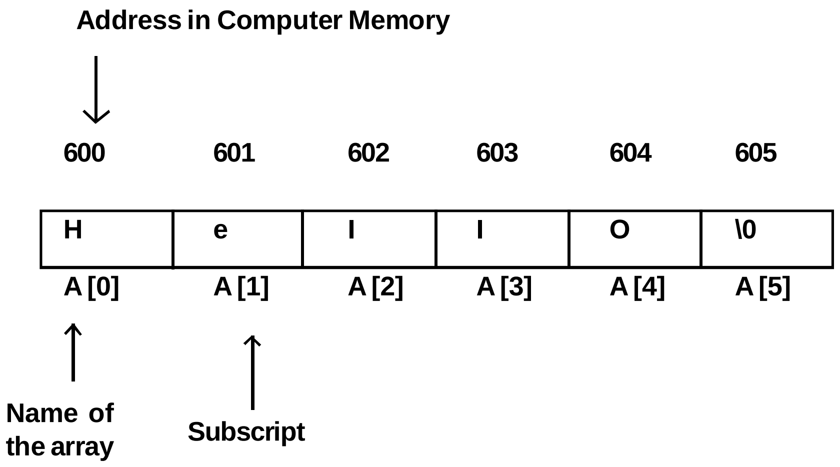


Fig. 10.1: Physical representation of string in memory

In Figure 10.1, we can see how array is actually stored in the memory. Each string element is identified with the help of the index or subscript value. The starting index value for string in C is always 0. In the above example, the first element of the string is stored in the memory address 600. For most C compilers char variable requires one byte of storage. Since strings are stored in contagious memory, so the second element will be stored next to the first element in location 601.

String Initialization: We can initialize the elements of the string when we declare the string. Initialization is generally done when we already know the character values of the string. Just like in declaration, in initialization also we give the data type of the string, followed by the name and size of the array. But in addition to these, we also provide the values of the character elements of the string within braces **{}** separated by commas where the values of the elements need to be put inside single quotes. Let us look at how initialization is done using the following example.

char A [6] = { ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘\0’ };

In the above example, the array A is declared and initialized. The character array A has a size of 6 and the values of the elements in the string have been initialized as follows:

A [0] = H

A [1] = e

A [2] = l

A [3] = l

A [4] = o

A [5] = ‘\0’

The first value that is given in braces is put into the first string location with subscript value 0. So ‘H’ is kept at location A [0]. Similarly the second value ‘e’ inside braces is put into the second string location with subscript value A [1] and so on till all the given values have been put in the string.

We can also initialize the values of string without mentioning the string size in the square brackets. That is, we can leave the square brackets empty. For this case the compiler assumes the string size to be equal to the number of char values provided inside braces { }. For example, we can write the following statement:

char A [ ] = { 'H', 'e', 'l', 'l', 'o', '\0' };

In the above statement, the compiler will assume that the size of the array is 6 since six data values have been given inside the braces. There is another much simpler way to initialize the string. We can also give the values of the character elements together in double quotes **""** instead to providing them individually. For example, we can write the following statement:

char A [ ] = "Hello";

For this case the compiler assumes the string size to be equal to the number of char values provided inside braces {} plus one more for the null character. In this case, we do not need to explicitly provide the null character at the end of the string. The compiler itself will add the null character at the end of the string.

10.3.2 Entering Values in String

We can insert values into a string at the time of initialization. But apart from that, we can also insert values into strings by accessing them individually. For example, if we have a string A of 6 elements, we can insert the values for each of the six string positions as follows:

A [0] = 'H';

A [1] = 'e';

A [2] = 'l';

A [3] = 'l';

A [4] = 'o';

A [5] = '\0';

In the above statements, the value for each string position has been filled individually instead of giving the values at the time of initialization and declaration. Let us now suppose that we have a string of 50 elements which consists in the name of a book.

A solution to the above problem is to use loop control structure statements like *while*, *do while* and *for* loop. Let us look at a code where with the help of *for* loop we can easily enter any number of character values. This can be implemented as:

```
for ( i = 0; i < 50;  i++ )
{
    scanf("%c", &A[ i ]);
}
```

In the code above, a 'for loop' is used to traverse from the first element of the string to the last element. The above 'scanf' statement accepts a character value and stores it in the address of the given string location. The location in the string is moved from the first element position to the last element position using for loop. "&A[i]" gives the address of the 'ith' element of the string where the current character data value needs to be stored.

An easier solution to this is that instead of providing the data values individually we can provide them as together as a string. Let us look at the code below which has implemented this solution.

```
scanf("%s", &A);
```

The above 'scanf' statement accepts a string instead of a single character and stores it in the address of the given string location. For our case, the above statement accepts the string and stores it the string variable A.

10.3.3 Accessing Values from Strings

We have learnt in the previous part about entering values in a string. Once the values have been inserted we may need to access the string for various purposes. The procedure for accessing the values of a string is similar to entering the values. This can be implemented in two ways. In the first way we access the values of the string individually as follows:

```
for ( i = 0; i < 6;  i ++ )
{
    printf("c", A[ i ]);
}
```

In the above code a 'for loop' is used to traverse from the first element of the string to its last element. The 'printf' statement prints the character value which is stored in the address of the given string location. The array location is moved from the first element position to the last element position using " *for loop*". "**A[i]**" gives the value of the element at the **ith position** of the string. In the second way we access the values of the string together. This can be implemented as:

printf("%s", A);

The above 'printf' statement prints a string instead of a single character from the address of the given string location.

//Program 10.1: Program to enter and access elements in strings

```
# include<stdio.h>
# include<conio.h>
void main()
{
   // Program to enter and access elements in strings
   char str1[10], str2[10];
   int i;
   clrscr();
   // Entering values individually
   printf("Enter string str1:\n");
   for(i=0;i<9;i++)
   {
      scanf("%c",&str1[i]);
   }
   // Displaying entered values individually
   printf(“The values entered in string1 are:\n”);
   for(i=0;i<9;i++)
   {
      printf(“Value  in  string  1  str1[%d]  are
%c\n”,i,str1[i]);
   }
   // Entering values together as a string
   printf(“Enter string str2:\n”);
   scanf(“%s”,&str2);
   //  Displaying  entered  values  together  as  a
string
   printf(“The values entered in string2 are:\n”);
   printf(“%s\n”,str2);
   getch();
}
```

10.4 ARRAY OF STRINGS

Array of strings is a two dimensional character array. Similar to two dimensional arrays, strings can also be of two or more dimensions.

We can declare a two dimensional string in the following way:

char name_of_string [row_size] [column_size] ;

In the above declaration syntax, data type of the string has to be of character type. It is followed by the name of the string variable and the row size and column size of the string are given inside different square brackets. For example:

char array [5] [5];

char student_name [3] [10] ;

char book_name [2] [4] ;

Inserting and accessing of values in a two dimensional string is similar to insertion and access of data elements in 2-D arrays. For example, suppose we need to store the name of 5 best players in the team. We can declare a string **name [5][30]** to keep the names of the persons and this can be implemented as:

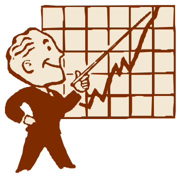
char name [5] [30] ;

In the above declaration, the string variable **name** has memory to store five names where each name can have a maximum of 29 characters. A *for loop* can be used for traversing from the first row to the 5th row of the string variable to insert and access the names of the five individuals. This can be implemented as:

```
for ( i=0 ; i < 5; i++ )
{
   scanf("%s",name[i]);
}
for ( i=0 ; i < 5; i++ )
{
   printf("%s",name[i]);
}
```

```
/*Program 10.2: Program to enter and access
elements in two dimensional strings */
# include<stdio.h>
# include<conio.h>
void main()
{
   // Program to enter and access elements in 2-
D strings
   char name[5][30];
   int i;
   clrscr();
   // Entering values in 2D string
   printf("Enter first name of five persons:");
   for(i=0;i<5;i++)
   {
      printf("\nEnter name of %d person:",i+1);
      scanf("%s",&name[i]);
   }
   // Displaying values in 2D string
```

```
        for(i=0;i<5;i++)
        {
           printf("\nName of %d person is :",i+1);
           printf("%s",name[i]);
        }
     getch();
     }
```



CHECK YOUR PROGRESS

Q.1: What is a string?

Q.2: Give the syntax for declaring a string.

Q.3: Can we declare a string without mentioning the size of a string?

10.5 STRING HANDLING FUNCTIONS

String handling contains functions that are used for manipulating and performing special operations on strings. The file **"*string.h*"** is available in the C library and contains many string manipulation functions. We can use the functions in this file by including the header file "string.h" in our C program. Some of the common functions are:

- strlen() function
- strcpy() function
- strcmp() function
- strrev() function
- strcat() function

Let us look at these functions in detail in the following section:

10.5.1 *strlen()* Function

The function *strlen()* is used for finding the number of characters present in a given string. It gives back the length of the specified string variable. The syntax is as follows:

len = strlen(str1);

where 'len' is an integer variable which keeps the length of the string variable 'str1'. The following program determines the length of a string using *strlen()* function.

```
//Program 10.3: Program to find the length of a
string
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   // Program to find the length of a string
   char str1[10];
   int len;
   clrscr();
   printf("Enter string :\n");
   scanf("%s",&str1);
   len = strlen(str1);
   printf("The length of string %s is
%d",str1,len);
   getch();
}
```

10.5.2 *strcpy()* Function

The function *strcpy()* is used for coping the contents of one string to another string. It copies the contents of the source string to a destination string. The syntax is as follows:

strcpy(str1,str2);

where 'str1' is the destination string and 'str2' is the source string. The following program copies contents of one string to another using *strcpy()* function.

```
/*Program 10.4: Program to copy the contents of
one string to another string*/
```

```
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   // Program to copy contents of one strin to
another string
   char str1[20], str2[10];
   clrscr();
   printf("\nEnter string str1:");
   scanf("%s",&str1);
   printf("\nEnter string str2:");
   scanf("%s",&str2);
   printf("\nValue of string str1 before copy is
:%s",str1);
   strcpy(str1,str2);
   printf("\nValue of string str1 after copy is
:%s",str1);
   getch();
}
```

10.5.3 *strcmp()* Function

The function *strcmp()* is used for comparing the contents of two strings. It compares the contents of the strings character by character and then returns an integer as the output. The syntax is as follows:

s = strcmp(str1,str2);

where 's' is an integer and 'str1', 'str2' are the two strings whose contents are compared. The value of the s has the following meanings:

If s = 0, then str1 and str2 are equal

If s = 1, then str2 > str1

If s = -1, then str1 > str2

The following program compares the values of two strings using strcmp() function.

```
/*Program 10.5: Program to compare the contents
of two strings*/
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   //Program to compare contents of one string to
another string
   char str1[10], str2[10];
   int result;
   clrscr();
   printf("\nEnter string str1:");
   scanf("%s",&str1);
   printf("\nEnter string str2:");
   scanf("%s",&str2);
   result = strcmp(str1,str2);
   if(result == 0)
      printf("\nString str1 is equal to String
str2 ");
   else
      printf("\nString str1 not equal String str2
");
   getch();
}
```

10.5.4 *strrev()* Function

The function *strrev()* is used to reverse the contents of any given string. It reverses all the contents of the string except the null character which is used to indicate the end of a string. The syntax is as follows:

strrev(str1);

where ‘str1’ is the string whose contents are to be reversed. The following program reverses the contents of a string using *strrev()* function.

```
//Program 10.6: Program to reverse the contents
of a string
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   // Program to reverse the contents of a string
   char str1[10];
   clrscr();
   printf("\nEnter string :");
   scanf("%s",&str1);
   printf("\nOriginal string: %s",str1);
   strrev(str1);
   printf("\nAfter string reversal. String is
%s",str1);
   getch();
}
```

10.5.5 *strcat()* Function

The function *strcat()* is used to combine the contents of one string with another string. It concatenates the contents of the source string to a destination string. The syntax is as follows:

strcat(str1,str2);

where ‘str1’ is the destination string and ‘str2’ is the source string. The following program concatenates the contents of one string to another string using strcat() function.

```
Program 10.7: Program to concatenate two strings
# include<stdio.h>
# include<conio.h>
# include<string.h>
void main()
{
   // Program to concatenate two strings
   char str1[20], str2[10];
   clrscr();
   printf("\nEnter string str1:");
   scanf("%s",&str1);
   printf("\nEnter string str2:");
   scanf("%s",&str2);
   strcat(str1,str2);
   printf("Conactenated string is %s\n",str1);
   getch();
}
```

CHECK YOUR PROGRESS

Q.4: What is string handling functions?

Q.5: Can we perform the operations done on strings by string handling functions without using these library functions?

10.6 LET US SUM UP

- An array of characters is called strings. In other words, strings are arrays where the data type is character.
- Strings are one-dimensional array of characters which are terminated by a **null character** represented by **'\0'**.
- We can declare a string in the following way:

  char name_of_string [size_of_string];
- The data type has to be a character. It is followed by the name of the string variable and the size of the string that is given inside square brackets.

- Array of strings is two dimensional character arrays. Similar to two dimensional arrays, strings can also be of two or more dimensions.
- We can declare a two dimensional string in the following way:

  char name_of_string [row_size] [column_size] ;
- String handling contains functions that are used for manipulating and performing special operations on strings. The file **"string.h"** is available in the C library and contains many string manipulation functions.
- The function *strlen()* is used to find the number of characters present in a given string. It gives back the length of the specified string variable.
- The function *strcpy()* is used to copy the contents of one string to another string. It copies the contents of the source string to a destination string.
- The function *strcmp()* is used to compare the contents of two strings. It compares the contents of the strings character by character and then returns an integer as the output.
- The function *strrev()* is used to reverse the contents of any given string. It reverses all the contents of the string except the null character which is used to indicate the end of a string.
- The function *strcat()* is used to combine the contents of one string with another string. It concatenates the contents of the source string to a destination string.

10.7 FURTHER READING

1) Kanetkar, Y. P. (2008); *Let us C*; Jones and Bartlett Publishers, Inc.
2) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
3) Thareja, R. (2015); *Introduction to C Programming*.

10.8 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: An array of characters is called strings. In other words, strings are arrays where the data type is character.

Ans. to Q. No. 2: We can declare a string in the following way:

char name_of_string [size_of_string];

The data type has to be a character. It is followed by the name of the string variable and the size of the string that is given inside square brackets.

Ans. to Q. No. 3: Yes, we can declare a string without mentioning its size provided we initialize it with the data elements.

Ans. to Q. No. 4: String handling contains functions that are used for manipulating and performing special operations on strings. The file **"string.h"** is available in the C library and contains many string manipulation functions.

Ans. to Q. No. 5: Yes, the operations performed by string handling functions can be performed without using them.

10.9 MODEL QUESTIONS

Q.1: Define strings. How are strings represented in memory?

Q.2: Write a program to find the length of a string without using library functions.

Q.3: Write a program to copy one string to another without using library functions.

Q.4: Write a program to combine two strings without using library functions.

Q.5: Write a program to reverse a string without using library functions.

Q.6: Write a program to compare between two strings without using library functions.

Q.7: Write a program to take input a number and a string and then display the string that many numbers of times.

*** ***** ***

UNIT 11: FUNCTIONS

UNIT STRUCTURE

11.1 LEARNING OBJECTIVES

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

- learn about function and its use in programs
- declare a function
- define a function
- describe function call
- learn about nesting of function
- learn about function parameters
- describe function categories
- illustrate recurrsive function

11.2 INTRODUCTION

In the earlier units, we have already used functions like *scanf( )*, *printf( )*, *clrscr( )*, *sqrt( )* etc. We have seen that C language supports the use of such **library** (or **built-in**) functions, which are used to carry out a number of commonly used operations or calculations.

However, C language also allows the users to define their own functions for carrying out various tasks. This unit concentrates on the creation and utilization of such ***user-defined*** functions. With the proper use of such user-defined functions, a large program can be broken down into a number of smaller, self-contained components, each of which has some unique purpose. This unit will help you in writing user-defined functions.

11.3 USE OF FUNCTIONS

A function is a set of statements that carries out some specific task in a program and it can be processed independently. Every C program consists of one or more functions. One of these functions must be called ***main***. Program execution will always begin by carrying out the instructions in *main*. There are many advantages in using functions in a program. They are:

- Many programs require that a specific function is repeated many times. Instead of writing the function code as many times as it is required, we can write it as a single function and access the same function again and again as many times as it is required.
- The length of the source program can be reduced by using functions at appropriate places.
- It is easy to locate and isolate a faulty function instead of modifying the whole program.
- If the whole large program is divided into subprograms, each subprogram can be written by one or two team members of the team rather than having the whole team to work on the complex program.
- A single function written in a program can be used in other programs also.

11.4 FUNCTION DECLARATION

A function declaration is also known as ***function prototype.*** Function prototypes are usually written at the beginning of the program, ahead of any user defined functions including *main*. It hints to the *compiler* that the ***main()*** function is going to call the function which is declared, later in the program. The general format of a function prototype is as follows:

return_type function_name(type1 arg1, type2 arg2,..,typeN argN);

where, ***return_type*** represents the data type of the item that is returned by the function, ***function_name*** represents the name of the function, ***type1, type2,...,typeN*** represent the data types of the arguments ***arg1, arg2, , , , ,argN.*** It is necessary to use a semicolon at the end of function prototype. Arguments name *arg1, arg2* etc. can be omitted. However, the argument data types are essential. For example:

int add(int, int);

In the above prototype decleration, *int* is the data type of the item returned by the function, *add* is the name of the function and *int* within the brackets '(' and ')' are the data types of the arguments.

Compiler: Program that decodes instructions written in a higher order language and produces an assembly language program.

Program 11.1: Program to find the sum of two numbers.

Solution:

```
#include<stdio.h>
#include<conio.h>
int add(int,int);    //function prototype or
declaration
void main()
{
   int a,b,s; // integer variable a,b,s are
declared
   clrscr();
   printf("Enter two integer number \n"); / /
display statement
   scanf("%d%d", &a,&b);
```

```
          s=add(a,b);   //function call
          printf("\nThe summation is  %d", s);
          getch();
          }
            int add(int a, int b)
          {
          int sum=0; //  local  variable  sum  and  it  is
intialised to zero
          sum=a+b;
          return sum;   // value of sum is returned
        }
```

CHECK YOUR PROGRESS

Q.1: State whether the following statements are True (T) or False (F):

i) Every C program should have atleast one function.

ii) In a function declaration arguments are separated by semicolon.

iii) The function prototype ends with a semicolon.

Q.2: Declare a function with function name "multiplication" to multiply between two floating point numbers. The function must have two fractional data type as argument which returns nothing to the calling function.

Q.3: What is the meaning of the statement *int substract(int,int);*?

11.5 FUNCTION DEFINITION

Functions can be defined anywhere in the program with a proper declaration, followed by the declaration of local variables and statements. A function definition should contain the following elements:

- name of the function
- list of parameters and their types
- body of the function
- return type

General format of function definition is as follows:

```
        return_type  function_name(parameter list)
        {
              local variable declaration;
              executable statement1 ;
              executable statement2 ;
              . . . . . . . . . . . . . . . . .
              . . . . . . . . . . . . . . . . .
              return statement ;
        }
```

The first line of function definition is known as ***function header*** which contains ***return_type***, ***function_name*** and ***parameter_list***. Function header is followed by an opening '{' and a closing brace '}' . The statements within the opening and closing braces constitute the ***function body.***

Function name should be appropriate to the task performed by the function. In C, two function name should not be same in a single program. If the function does not return anything then the return type will be ***void,*** otherwise it is the type of the value returned by the function. If the return type is not mentioned explicitly, C compiler assumes that it is an ***integer*** type.

Return statement contains the output produce by the function and its type. The *return* statement serves two purposes:

- On executing the *return* statement, it immediately transfers the control back to the calling program.
- It returns the value to the calling program.

Program 11.1: Let us consider the following program segment.

```
      int add(int a, int b)
      {
         int sum = 0; // local variable sum and it is
intialised to zero
         sum = a + b;
         return sum;  // value of sum is returned to
the calling function
      }
```

In the above program segment, the summation of the value stored in the variable *a* and *b* are returned. As the summation of two integers is also an integer, so the return type is *int .*

Example 11.2: Let us consider the following program segment

```
void display( )
{
     printf("State Open University ");
}
```

display( ) function does not return anything. So, the return type is ***void***. We can also write the ***display( )*** function as ***void display(void)*** by mentioning ***void*** explicitly within the bracket because no argument is passed.

11.6 FUNCTION CALL

Once a function has been declared and defined, it can be called from anywhere within the program: from within the *main()* function, from another function, and even from itself. We can call a function by simply using the function name followed by a list of parameters(or arguments) if any, enclosed in parentheses. For example,

```
s = add (a,b) ;   //Function call
```

In the above statement **add(a,b)** function is called and value returned by it is stored in the variable **s**. When the compiler encounters a function call, the control is transferred to **int add(int x, int y)**. This function is then executed line by line as described and a value is returned when a return statement is encountered. In our example, this value is assigned to **s**. This is illustrated below:

Program 11.2: Program to find the summation of two numbers using function

```
#include<stdio.h>
#include<conio.h>
int add(int, int);    // function declaration
void main()
{  int a,b,s;
   clrscr();
```

```
        printf(“Enter two integer number \n”);
        scanf(“%d%d”, &a,&b);
        s=add(a,b);  // function call
        printf(“\nThe summation is  %d”, s);
        getch();
     }
     int add(int x, int y) //function header
     {
        int sum=0;   //local variable sum and it is
intialised to zero
        sum=x+y;
        return sum;  //value stored in sum is returned
to the calling function
     }
```

Parameter passing is a method for communication of data between the *calling function* and *called function*. These can be achieved by two ways:

- Call-by-value
- Call-by-reference

11.6.1 Call-by-value

In case of *call-by-value*, the compiler copies the value of an argument in a *calling function* to a corresponding parameter in the *called function* definition. The parameter in the called function is initialized with the value of the passed argument. As long as the parameter has not been declared as constant, the value of the parameter can be changed, but the changes are performed only within the scope of the *called function*; they have no effect on the value of the argument in the *calling function*.

In the following example, the *calling function* **main()** passes two values 5 and 10 to the *called function* **func()**. The function **func()** receives copies of these values and accesses them by the identifiers **a** and **b**. The function **func()** changes the value of **a**. When control passes back to **main()**, the actual values of **x** and **y** are not changed.

Program 11.3: Program to illustrate calling a function by value.

```
#include<stdio.h>
void func(int, int);
void main(void)
{
   int x = 5, y = 10;
   clrscr();
   func(x, y);
   printf("In main, x = %d y = %d\n", x, y);
   }
      void func(int a, int b)
   {
   a = a + b;
   printf("In func, a = %d b = %d\n", a, b);
}
```

Output:

```
           In func, a = 15 b = 10
       In main, x = 5  y = 10
```

11.6.2 Call-by-reference

Call-by-reference refers to a method of calling a function by passing the address of an argument in the *calling function* to a corresponding parameter in the *called function*.

We have used an example below to illustrate the concept of *call-by-value* and *call-by-reference*. The aim of the two programs listed below is to perform swapping (interchange) of two values.

Program 11.4: Example to illustrate calling a function by value.

```
#include<stdio.h>
#include<conio.h>
void swap(int, int); //function prototype or
```

```
declaration
void main()
{
   int a,b;
   a=5;
   b=10;
   printf("a and b before interchange: %d %d", a,
b);
   swap(a,b);    //function call
   printf("\na and b after interchange: %d %d",
a, b);
   getch();
}
void swap(int i, int j)    //function definition
{
   int t;
   t = i;
   i = j;
   j = t;
}
```

Here, the value to function *swap()* is passed by value. When we execute this program, we will find that no swapping takes place. The values of **a** and **b** are passed to swap, and the swap function does swap them, but when the function returns to main() nothing happens.The values of **a** and **b** are still the same. The output will be:

```
a and b before interchange:  5   10
a and b after  interchange:  5   10
```

Pointer: A pointer is a variable that holds the address of another variable.

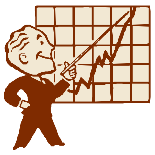
In the next program we use pointers to perform call-by-reference for swapping of two values. Our next unit will help you in understanding ***pointers***.

Program 11.5: Example to illustrate calling a function by reference

```
#include <stdio.h>
#include<conio.h>
void swap(int *, int *);   //function declaration
void main( )
{
   int a,b;
   a=5;
   b=10;
   clrscr( );      // clearing the screen
   printf("a and b before interchange: %d
%d\n",a,b);
   swap(&a,&b);    //function call
   printf("a and b after interchange: %d
%d\n",a,b);
}
void swap(int *i, int *j)
{
   int t;
   t = *i;
   *i = *j;
   *j = t;
}
```

Here, the function uses called-by-reference. In other words, address is passed by using the symbol **"&"** and the value is accessed by using the symbol **"*"**. When **swap( )** function is called, the addresses of **a** and **b** are passed to the function. Thus, **i** points to **a** and **j** points to **b**. Once the pointers are initialized by the function call, ***i** is another name for **a**, and ***j** is another name for **b**. When the code uses ***i** and ***j**, it really means **a** and **b**. So, when we interchange the values of **i** and **j** in function **swap()**, we interchange the values of **a** and **b**. Hence, when the function is complete, **a** and **b** have been interchanged. The output will be:

```
a and b before interchange:    5    10
a and b after  interchange:   10     5
```



CHECK YOUR PROGRESS

Q.4: Fill in the blacks:

i) When a function returns nothing then the return type is __________.

ii) If a C program has only one function then that function is __________.

iii) The parameters used in a function call are __________.

iv) When a variable is passed to a function by value, its value remains __________ in the calling program.

v) A function can be called either by __________ or __________ or both.

Q.5: Write down the syntax of function definition.

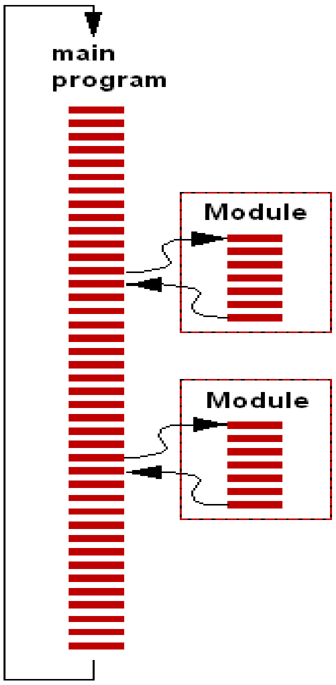
Q.6: Write the first line of the function definition, including the formal argument declarations, for each of the situations described below:

i) A function called ***average*** accepts two integer arguments and returns a floating-point result.

ii) A function called ***convert*** accepts a character and returns another character.

LET US KNOW

Modular programming is a strategy applied to the design and development of software systems. It is defined as organizing a large program into small, independent program segments called **modules** that are separately named and can be individually called. It is basically *"a divide-and-conquer"* approach to problem solving. In C, each module refers to a function that is responsible for a single task. The module performs a function and then returns control back to the program or instruction that called it.



11.7 NESTING OF FUNCTIONS

C permits nesting of functions freely. There is no limit to how deeply functions can be nested. A *nested function* is encapsulated within another function.

Suppose a function **a** can call function **b** and function **b** can call function **c** and so on. We have taken the following example to illustrate nesting of function.

Program 11.6: Program to illustrate the concept of nested function

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int a,b,c;
   float r;
```

```
        clrscr();
        float ratio(int,int,int);    //   function
ratio( ) declared
        printf("Enter a,b and c :");
        scanf("%d%d%d",&a,&b,&c);
        r=ratio(a,b,c);   // ratio( ) function called
        printf("%f\n",r);
        getch();
     }
     float ratio(int x, int y, int z)
     {
        int difference(int,int);     //   function
difference( ) declared
        if(difference(y,z))
           return(x/(y-z));
        else
           return(0.0);
        }
        int difference(int p, int q)
        {
        if(p!=q)
           return(1);
        else
           return(0);
     }
```

The above program calculates the ratio $\frac{a}{b-c}$ and prints the result. We have the following three functions:

main( )

ratio( )

difference( )

The **main( )** function reads the value of a,b,c and calls the function **ratio( )** to calculate the value a / (b-c). This ratio cannot be evaluated if (b-c) =0. Therefore, **ratio( )** calls another function **difference( )** to test whether the **difference(b-c)** is zero or not.

11.8 FUNCTION PARAMETERS

C functions exchange information by means of parameters. The term ***parameter*** refers to any declaration within the parentheses following the function name in a function declaration, definition or function call.

Formal Parameters: The parameters which appear in the first line of the function definition are referred to as *formal parameter*. Formal parameters are written in the function prototype and function header of the definition. Formal parameters are local variables which are assigned values from the arguments when the function is called.

Actual Parameters: When a function is called, the values (expressions) that are passed in the call are called the *actual parameters*. At the time of the call each actual parameter is assigned to the corresponding formal parameter in the function definition. It may be expressed in constants, single variables, or more complex expressions. However, each actual parameter must be of the same data type as its corresponding formal parameter.

The following rules apply to parameters of C functions:

- Except for functions with variable-length argument lists, the number of arguments in a function call must be the same as the number of parameters in the function definition. This number can be zero.
- Arguments are separated by commas.
- The scope of function parameters is the function itself. Therefore, parameters of the same name in different functions are unrelated.

Let us consider the following example to illustrate the concept of formal and actual parameters:

Program 11.7: Program to illustrate the concept of formal and actual parameters

```
#include<stdio.h>
#include<conio.h>
void display(int,int);
void main()
{  int a,b;
   display(a,b);
   getch();
}
void display(int x, int y)
{
   printf(“%d%d”,x,y);
}
```

Here, **x** and **y** are formal parameters and take the value (**a,b**) from the calling function **display(a,b)**.

11.9 CATEGORIES OF FUNCTION

Depending on whether arguments are present or not and whether a value is returned or not, functions are categorised as follows:

- Functions with no arguments and no return values
- Functions with arguments and no return values
- Functions with arguments and one return value
- Functions with no arguments but a return value
- Functions that return multiple values

We have illustrated the above categories of functions by using different programs. The concept of different categories of functions is explained using an example to “Multiply of two integer numbers”.

- **Functions with no arguments and no return values**

Program 11.8: Program to illustrate the concept of a function with no arguments and no return values

```
#include<stdio.h>
#include<conio.h>
void multi(void);     //function declaration with
no argument
void main( )
{
   clrscr( );
   multi( );
   getch( );
}
void multi(void)
{
   int a,b,m;
   printf("Enter two integers:");
   scanf("%d%d", &a,&b);
   m=a*b;
   printf("\nThe product is: %d",m);
}
```

- **Functions with arguments and no return values**

Program 11.9: Program to illustrate the concept of a function with arguments and no return values

```
#include<stdio.h>
#include<conio.h>
void multi(int,int); //function declaration with two argument
void main( )
{  int a,b;
   clrscr( );
   printf("Enter two integers:");
   scanf("%d%d", &a,&b);
   multi(a,b);
   getch( );
}
void multi(int a,int b )
{  int m;
   m=a*b;
   printf("\nThe product is: %d",m);
}
```

- **Functions with arguments and one return value**

Program 11.10: Program to illustrate the concept of a function with arguments and one return values

```
#include<stdio.h>
#include<conio.h>
int multi(int,int);   //function declaration with two argument
void main( )
{
   int a,b,m;
   clrscr( );
   printf(“Enter two integers:”);
   scanf(“%d%d”, &a,&b);
   m=multi(a,b);
   printf(“\nThe product is: %d”,m);
   getch( );
}
int multi(int a,int b )
{
   int z;
   z=a*b;
   return z;    /*return statement. the value of
                z is returned to the calling
                function*/
}
```

- **Functions with no arguments but a return value**

Program 11.11: Program to illustrate the concept of a function with no arguments but a return value

```
#include<stdio.h>
#include<conio.h>
int multi(void);      //function declaration with no argument
void main( )
```

```
{
   int m;
   clrscr( );
   m=multi( );
   printf("\nThe product is: %d",m);
   getch( );
}
int multi(void)
{
   int a,b,p;
   printf("Enter two integers:");
   scanf("%d%d", &a,&b);
   p=a*b;
   return p;
}
```

Return statement can return only one value. In C, the mechanism of sending back information through arguments is achieved by two operators known as the *address operator* (&) and *indirection operator* (*). Let us consider an example to illustrate this.

- **Functions returning multiple values**

Program 11.12: Program to illustrate the concept of a functions returning multiple values

```
#include<iostream.h>
#include<conio.h>
void calculate(int, int, int *, int *);
void main( )
{
   int a,b,s,d;
   clrscr( );
   printf("\nEnter two integer:");
   scanf("%d%d",&a,&b);
   calculate(a,b,&s,&d);
   printf("\nSummation is:%d \n Difference is:%d",
s,d);
        getch();
     }
     void calculate(int x,int y, int *sum, int *diff)
     {
        *sum=x+y;
        *diff=x-y;
     }
```

In the fuction call, while we pass the actual values of a and b to the function calculate(), we pass the address of locations where the values of s and d are stored in the memory.

When the function is called, the value of **a** and **b** are assigned to **x** and **y** respectively. Address of **s** and **d** are assigned to **sum** and **diff** respectively. The variables ***sum** and ***diff** are known as pointers and **sum** and **diff** pointer variables. Since they are declared as **int**, they can point to locations of **int** type data.

11.10 RECURSIVE FUNCTION

When a function calls itself it is called a ***recursive function***. Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. A very simple example is presented below:

Program 11.13: Program to illustrate the concept of recursive function

```
     #include<stdio.h>
     #include<conio.h>
     void main( )
     {
        printf(“Recursive function\n”);
        main( );
        getch( );
     }
```

The output of the above programme will be like this:

```
Recursive function
Recursive function
Recursive function
Recursive function
...........................
...........................
```

In the above case, we will have to terminate the execution abruptly; otherwise the program will execute indfinitely.

The factorial of a number can also be determined using recursion. The factorial of a number **n** is expressed as a series of repeatitive multiplications as shown below:

Factorial of n = n(n-1)(n-2)(n-3).....1

For example, factorial of 5= 5×4×3×2×1 =120

Program 11.14: Program to find factorial of an integer number

```
#include<stdio.h>
#include<conio.h>
long int factorial(int);
void main( )
{
   int n ;
   long int f ;
   clrscr( ) ;
   printf("\nEnter an integer number:") ;
   scanf("%d", &n) ;
   f=factorial(n) ;
   printf("\nThe factorial of %d is : %ld",n,f) ;
   getch() ;
}
long int factorial(int n)
{
        long int fact ;
        if(n<=1)
           return(1);
        else
           fact=n*factorial(n-1);
        return(fact);
     }
```

Let us see how recursion works assuming n = 5. If we assume n=1 then the factorial( ) function will return 1 to the calling function. Since n $\neq 1$, the statement

fact = n * factorial (n-1);

will be executed with n=5. That is,

fact = 5 * factorial (4);

will be evaluated. The expression on the right-hand side includes a call to factorial with n = 4 .This call will return the following value :

4 * factorial(3)

In this way factorial(3), factorial(2), factorial(1) will be returned. The sequence of operations can be summarized as follows:

```
fact  = 5 * factorial (4)
      = 5 * 4 * factorial (3)
      = 5 * 4 * 3 * factorial (2)
      = 5 * 4 * 3 * 2 * factorial (1)
      = 5 * 4 * 3 * 2 * 1
      =120
```

When we write recursive functions, we must have an *if* statement somewhere to force the function to return without the recursive call being executed. Otherwise, the function will never return.

Program 11.15: Program to find the sum of digits of a number using recursion.

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

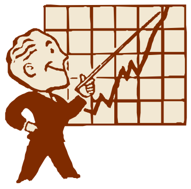
```
   int sum(int); //function prototype
   int n,s;
   clrscr();
   printf("\n Enter a positive integer:");
   scanf("%d",&n);
   s=sum(n);
   printf("\n Sum of digits of %d is %d ", n,s);
   getch();
}
int sum(int n)
{
   if(n<=9)
      return(n);
   else
      return(n%10+sum(n/10));    //   recursive
call of sum()
}
```

Output: Enter a positive integer: 125

Sum of digits of 125 is 8

EXERCISE

1) Write a C program to find the GCD (Greatest Common Divisor) of two positive integers using recursion.



CHECK YOUR PROGRESS

Q.7: State whether the following statements are True (T) or False(F).

i) The parameters which appear in the first line of the function definition are formal parameter

ii) Arguments are separated by semicolon.

iii) The 'C' language does not support recursion.

iv) The main( ) function can call itself recursively.

v) You can call main() from any other function.

vi) The same variable names can be used in different functions without any conflict.

Q.8: Write a C program to generate first n fibonacci terms using recursion.

11.11 LET US SUM UP

- A function is a self-contained program segment that carries out some specific, well-defined task.
- A function has three principal components: function prototype or declaration, function call, function definition.
- Function prototype or declaration is always followed by a semicolon.
- Call-by-value copies the *value* of an argument to the corresponding parameter in the called function
- Call-by-reference passes the *address* of an argument to the corresponding parameter in the called function.
- The argument that is passed is often called an actual argument while the received copy is called a formal argument or formal parameter.
- We can pass parameters to a function by value and by reference.
- Functions can return any type that we declare, except for arrays and functions. Functions returning no value should return void.
- A return statement is required if the return type is anything other than void.
- A function definition may be placed either after or before the main () function.
- Functions taking a variable number of arguments must take at least one named argument; the variable arguments are indicated by... as shown: **int func(int x, float y, ...);**
- When a function calls itself is called a recursive function.

11.12 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.

2) Gottfried Byron, S. Programming with C.; Tata McGraw Hill Education.

11.13 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) True, ii) False, iii) True

Ans. to Q. No. 2: void multiplication(float, float);

Ans. to Q. No. 3: The statement **int substract(int, int);** is a function declaration where the function name is "substract" and the function has two integer type arguments and its return type is also integer.

Ans. to Q. No. 4: i) void, ii) main( ), iii) actual parameters, iv) unchanged, v) call by value, call by reference

Ans. to Q. No. 5: General format of function definition is given below:

```
return_type  function_name(parameter list)
{
    local variable declaration;
    executable statement1 ;
    executable statement2 ;
    . . . . . . . . . . . . . . . . .
    . . . . . . . . . . . . . . . . .
    return statement ;
}
```

Ans. to Q. No. 6: i) float average(int a, int b)

ii) char convert(char a)

Ans. to Q. No. 7: i) True, ii) False, iii) False, iv) True, v) False, vi) True

Ans. to Q. No. 1: **Solution:**

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

```
#include<conio.h>
void main()
{
   unsigned long fibo(int);
   int i,n;
   clrscr();
   printf("\nHow many fibonacci terms do you want
?\n");
   scanf("%d",&n);
   printf("\n%d fibonacci terms are: \n\n",n);
   for(i=1;i<=n;i++)
      printf("%4lu",fibo(i));    //function call
   getch();
}
unsigned long fibo(int n)
{
   if(n==1)
      return(0);
   else
   {
   if(n==2)
      return(1);
   else
      return(fibo(n-1)+fibo(n-2));    /        /
recusive call
   }
}
```

11.14 MODEL QUESTIONS

Q.1: Explain the meaning of following function prototypes.

a) char func(void);

b) double f(double a, int b);

c) int calculate(int a, int b);

d) void change(int *, int *);

e) void display( );

Q.2: State three advantages to the use of functions.

Q.3: What is meant by a function call? From what part of a program can a function be called?

Q.4: What are formal and actual arguments? What is the relationship between formal and actual argument?

Q.5: What is the purpose of *return* statement?

Q.6: Can a function be called from more than one place within a program?

Q.7: What is recursion? Explain it with example.

Q.8: Write a complete C program that will calculate the real roots of the quadratic equation $ax^2+bx+c=0$.

Q.9: Write a C program using function to find the square of an integer number without using the library function sqrt( ).

Q.10: What is a *main()* function?

*** ***** ***

UNIT 12: POINTERS

UNIT STRUCTURE

12.1 LEARNING OBJECTIVES

After going through this unit you will be able to:

- learn the basic concept of pointer
- declaring the pointer variable
- access array elements using pointer
- relate pointers to functions

12.2 INTRODUCTION

A pointer is a variable that represents the location of a data item or memory area. In other words, pointer variable holds address of other memory location rather than a value. Pointers make C and C++ more reliable and flexible. One can access the memory location directly using pointer. Within the computer memory, every data item occupies one or more contiguous memory locations. The number of required memory location depends on the data type used. For example, a character type variable occupies 1 byte (8 bits) of memory, an integer usually requires two contiguous bytes (16 bits), a floating-point number requires 4 contiguous bytes(32 bits) and so on. Each and every memory location has a unique memory address or location number.

12.3 DECLARING POINTER VARIABLE

Declaration of pointer variable is just like normal variable declaration. Here, only the variable is followed by an *(asterisk). The general form is:

<data type> *<variable name>;

Example 1: int *p, *q;
char *name;
float *p;

Example 2:

	p	x	Variable name
int *p, x=10;	$\boxed{2001}$	$\boxed{10}$	Value at address
p=&x;	2001		Address of memory or variable

Here, p is a pointer variable of type integer i.e., p can store address of some other integer variable. Integer pointer can point only other integer variable. The symbol '**&'** is an **address of** operator. The statement p=&x assigns the address of variable x into p. We assume the address of x is 2001, it also may be some other value. Here is an example to look at the address of a pointer variable and value.

```
/*Program 12.1: Program to display address and
value of a variable using pointer.*/
#include<stdio.h>
#include<conio.h>
void main()
{
   int *p, x=10;
   clrscr();
   p=&x;
   printf("Address of x %d\n", &x);
   printf("Value of p %d\n",p);
   printf("Value at address %d ", *p);
   getch();
}
```

12.4 POINTER ARITHMETIC

The following arithmetic operations can be performed with pointer variables in C and C++:

Subtraction	–
Incrementation	++
Decrementation	– –

Pointer arithmetic follows data type size i.e., it causes the pointer to be incremented or decremented by the number of bytes occupied by a particular data type. For example, an integer pointer variable when incremented by 1, will increase by 2, as the size of an integer variable in windows environment is 2 bytes. This could be explained with the help of the following example.

```
//Program 12.2: Program showing pointer arithmetic.
#include<stdio.h>
#include<conio.h>
void main()
{
   int *p, x=10;
   clrscr();
   p=&x;
   printf("p= %d\n", p);
   p=p+1;
   printf("p= %d\n",p);
   p=p+1;
   printf("p=%d ", p);
   getch();
}
```

Output:
```
p= –12
p= –10
p= –8
```

Here, the value of p increases by 2 rather than by 1. It is because of the data type integer. One can change the data type from integer to float. In such situation p will increase by 4.

12.5 POINTERS AND ONE DIMENSIONAL ARRAYS

As discussed earlier, an array is a variable to represent multiple memory locations with the same name. Each and every element of an array has their unique memory addresses. These memory addresses can store into pointer variables. Therefore, if x is a one dimensional array, then the address of the first array element can be expressed as either &x[0] or simply as x. Moreover, the address of second array element can be written as either &x[1] or (x+1), and so on. This can be explained with one example.

Let int x[5] = {10,20,30,40,50};

Here,	0	1	2	3	4	index
x	10	20	30	40	50	Data
	2000	2002	2004	2006	2008	Address

Here, we assume that address of the first array element i.e., x[0] is 2000, so the address of second array element will be 2002, since one integer variable occupies 2 bytes in memory.

//Program 12.3: `Display the content of an array using pointer`

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int *p, x[5]={10,20,30,40,50};
   clrscr();
   p=&x[0];
   printf("First element=%d\t",*p);
   p=p+1;
   printf("Second element= %d\t",*p);
   p=p+1;
   printf("Third element= %d\t",*p);
   p=p+1;
   printf("Fourth element= %d\t",*p);
   p=p+1;
```

```
        printf(“Fifth element= %d\t”,*p);
        getch();
     }
```

The above program can also be written using loop construct, which reduces the number of statements.

```
     /*Program 12.4: Program to display the array
content using pointer and loop*/
     #include<stdio.h>
     #include<conio.h>
     void main()
     {
        int *p,i, x[5]={10,20,30,40,50};
        clrscr();
        p=&x[0];
        printf(“Elements are : “);
        for(i=0;i<=4;i=i+1)
        {
           printf(“%d”,*p);
           p=p+1;
        }
        getch();
     }
```

Here, p is a pointer variable of type integer and initially assigned the address of the first array element x[0]. In subsequent iteration or repetition the value of p is increased by 1. i.e., by 2 bytes, since p is an integer variable and displays the value of that address.

12.6 POINTERS AND CHARACTER ARRAYS

A character array forms a string. In other words, a string may be considered as a string of characters. So, string can be processed using pointer variables. Strings are always terminated with null character i.e. ’\0’.

```
     //Program 12.5: Program to display characters of
```

```
a string using pointer.
     #include<stdio.h>
     #include<conio.h>
     void main()
     {
        char name[ ]=”KKHO UNIVERSITY”;
        char *p;
        clrscr();
        p=&name[0];
        while(*p!=’\0')
        {
           printf(“%d”,*p);
           p++;
        }
        getch();
     }
```

Here, the base address of the character array “name” is stored into the pointer variable p. In each iteration, value at address of p is displayed, then p is incremented by 1 until ’\0’ is found. One can increment the pointer by using “++” operator and can decrement the pointer by using “- -” operator.

```
     /*Program 12.6: Program to display characters of
a string in reverse order using pointer.*/
     #include<stdio.h>
     #include<conio.h>
     void main()
     {
        char name[]=”KKHSO UNIVERSITY”;
        char *p,*q;
        clrscr();
        p=&name[0];
        q=&name[0];
        while(*p!=’\0')
        {
```

```
        p++;
    }
        p–;
    while(p!=q)
    {
        Printf(“%d\t”,*p);
        p–;
    }
    getch();
}
```

12.7 PASSING POINTERS TO FUNCTIONS AS ARGUMENTS

Pointers can pass to a function as arguments. Arguments can be passed to a function in two ways i.e. **by value** and **by reference**. When an argument is passed by value, the data item is copied to the function i.e. it makes a duplicate copy of the original variable. In such situation, any alteration or modification made in the variables of the function does not affect the value of the original variable. Some situation may arise where we want to change the value of variable in the original program with the help of a function. When calling a function using pointers, it copies the address of a variable to the function, not the value of the variable. The contents of that address can be accessed directly either within the called function or calling function. In other words, pointer variables are the direct means of communication among the function variables. Let us explain the use of a pointer variable with the help of the following example.

```
/*Program 12.7: Program to show passing address
of a variable to a function.*/
#include<stdio.h>
#include<conio.h>
void main()
{
   void passdata(int *p);
   int x=10;
   clrscr();
   passdata(&x);
   getch();
}
void passdata(int *p)
{
   printf(“Address of x : %d\n”,p);
   printf(“Value of x : \n”, *p)”;
}
```

Here, there are two functions viz. main() and passdata(). The main() function calls the passdata() function and passed the address of variable x to the function. In passdata(), p is a pointer variable to accept the address of variable x. Here, p is a formal parameter and x is actual parameter.

```
/*Program 12.8: Program to show call by value and
call by reference*/
#include<stdio.h>
#include<conio.h>
void main()
{
   void passdata1(int *p);
   void passdata2(int p);
   int x=10;
   clrscr();
   printf(“Value of x before passdata1 call %d\n”,
x);
   passdata2(x);
   printf(“Value  of  x  after  passdata1  call
%d\n”,x|);
   passdata1(&x);
   printf(“Value of x after passdata2 call %d\n”,
x);
        getch();
      }
        void passdata1(int *p)
      {
        *p=20;
      }
      void passdata2(int p)
      {
        p=20;
      }
```

This program contains two functions, called passdata1() and passdata2(). The function passdata1() receives one pointer to integer variable as its argument and passdata2() receives one integer variable as argument. This variable originally assigned a value 10. The value is then changed to 20 within passdata2() function. The new value is not reflected within main, because the argument x was passed by value. As we have discussed earlier, when a function is called by value, then a duplicate copy of the original variable is copied to the called function. Any changes to the variables of called function are local to that function only. In passdata1(), the statement *p=20 indicates that the value 20 is assigned to the contents of the pointer address, which is address of x. Since the address is recognised in both functions i.e. passdata1() and main(), the reassigned value will be recognised within main after call to passdata1().

Note : *Function declaration, call and definition need not to be in order*.

```
      /*Program 12.9: Program to exchange value of two
variable using function*/
      #include<stdio.h>
      #include<conio.h>
      void main()
      {
        void exchange(int *p, int *q);
        int x=10, y=20;
```

```
   clrscr();
   printf(“Value of x & y before function call”);
   printf(“x=%d y=%d\n”,x,y);
   exchange(&x, &y);  //passing address of x and
y
   printf(“Value of x & y after function call”);
   printf(“x=%d y=%d\n”,x,y);
   getch();
}
void exchange(int *p, int *q)
{
   int temp;
   temp=*p;
   *p=*q;
   *q=temp;
}
```

12.8 DYNAMIC MEMORY ALLOCATION

Memory allocation refers to the reservation of memory for storing data. Memory allocation is done in C language in two ways (i) Static allocation and (ii) Dynamic allocation. Static allocation is done by using array. The main disadvantage of static allocation is that the programmer must know the size of the array or data while writing the program. Generally, it is not possible to know the required memory in advance. To overcome this problem dynamic memory allocation is done. Dynamic memory allocation refers to the allocation of memory during program execution. The basic difference between static and dynamic memory allocation is that in static allocation memory is allocated during the compile time, whereas in dynamic allocation memory is allocated during the program execution time.

12.8.1 Library Function for Dynamic Memory Allocation

C language provides a set of library functions for dynamic memory allocation and de-allocation. There are basically four functions used for this purpose. They are:

malloc( ), ***calloc( )***, ***realloc( )*** and ***free( )***

- **malloc():** This function is used to allocate a block of memory. After allocating the memory it returns a pointer of type *void*. This means that one can assign it to any type of pointer. The syntax for using *malloc()* is as follows:

  ptr =(cast-type *)malloc(no. of bytes);

  Here, ptr is a pointer of type cast-type. For example,

  ```
  int x;
  x=(int *)malloc(100);
  ```

  If it is unable to find the requested amount of memory, malloc() function returns NULL.

- **calloc():** It is a library function to allocate memory. The difference between *calloc()* and *malloc()* is that *calloc()* initialize the allocated memory to zero where as *malloc()* does not initialize allocated memory to zero.

  Declaration Syntax: Following is the declaration for *calloc()* function.

void (cast-type*)calloc(no. of elements to be allocated, size of each element)

  For example:

  ```
  ptr = (int*) calloc(10, sizeof(int));
  ```

  This statement allocates contiguous space in memory for an array of 10 elements each of size of int, i.e., 2 bytes.

```
/*Program 12.10: Program to show the usage of
  calloc() function*/
#include <stdio.h>
#include <stdlib.h>
int main()
```

```
{
   int i,no;
   int *p;
   printf("Enter number of elements :");
   scanf("%d",&no);
   p = (int*)calloc(no, sizeof(int));
   printf("Enter %d numbers:\n",no);
   for( i=0 ; i < no ; i++ )
   {
      scanf("%d",&p[i]);
   }
   printf("The numbers are: ");
   for( i=0 ; i < n ; i++ )
   {
      printf("%d ",p[i]);
   }
      free( p );
      return(0);
}
```

- **free( ):** A memory area that is dynamically allocated using either calloc() or malloc() doesn't get freed automatically when the execution terminates. You must explicitly use free() library function to release the memory space.

  Syntax: free(ptr);

  This statement frees the space allocated in the memory pointed by ptr.

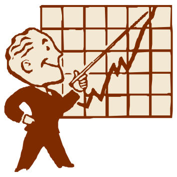
- **realloc( ):** The size of dynamically allocated memory can be changed by using realloc() library function.

  Syntax: void *realloc(void *ptr, size_t size);

  Example: int *ptr = (int *)malloc(sizeof(int)*2);//dynamic allocation for two integer value

  int *ptr_new;

  ptr_new = (int *)realloc(ptr, sizeof(int)*3);// dynamic reallocation for three integer value



CHECK YOUR PROGRESS

Q.1: Choose the appropriate option:

i) Which one of the following is valid pointer declaration:

a) int a*; b) int *a,

c) int *a d) int *a;

ii) What will be the output of the following statements if the variable 'a' located at address 50 [Assume no syntax error and working platform is MS Windows]

```
int *p, a=50;
p=&a
a++, p++;
a++, p++;
printf("%d%d", a,p);
```

a) 52, 54 b) 54, 52 c) 52, 52 d) 54, 54

iii) In a string '\0' is known as:

a) Back zero b) Slash zero

c) Null character d) End character

iv) The meaning of the following statements is:

void sum(int *p, int *q);

a) Function declaration

b) Call by address

c) Function does not return value

d) All of the above

v) malloc() is:

a) used to allocate memory statically

b) used to allocate memory dynamically

c) a user defined function

d) does not return value

12.9 LET US SUM UP

- In this unit we have discussed pointers and their relation to array and function.
- Pointers are variables that hold the address of other variables or memory locations.
- Pointer variables behave different way depending on the data type. For example a pointer to *char* is different from a pointer to *int*.
- There are two special operators used in pointer operation i.e. “address of” operator, **&** and “content of” or “value at address” operator *****.

12.10 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.

2) Gottfried Byron, S. Programming with C.; Tata McGraw Hill Education.

12.11 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: i) (d) int *a, ii) (a) 52, 54, iii) (c) Null character, iv) (d) None of the above, v) (b) used to allocate memory dynamically

12.12 MODEL QUESTIONS

Q.1: What is pointer? How pointer variables are declared?

Q.2: Explain the mechanism to access one dimensional array using pointer.

Q.3: What is pointer arithmetic? What are the operators used in pointer arithmetic.

Q.4: Write a program to input your name and display it in reverse order using pointer.

Q.5: Write a program to input your name and display it using pointer

Q.6: Explain how pointers are passed to a function.

Q.7: Write a program to input a string and count the number of vowels

Q.8: What is dynamic memory allocation? What are the functions used to allocate memory dynamically.

Q.9: What are the difference between ‘pass by value’ and ‘pass by reference’.

*** ***** ***

UNIT 13: STRUCTURE AND UNION

UNIT STRUCTURE

13.1 LEARNING OBJECTIVES

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

- write program using a structure rather than several arrays
- learn how structures are defined and how their individual members are accessed and processed within a program
- declare structure variables
- learn about array of structures
- declare and use pointer to structure
- learn about union
- describe enumerated data types
- learn about typedef

13.2 INTRODUCTION

We have already been acquainted with array which is a linear data structure. Array takes basic data types like *int*, *char, float* or *double* and organises them into a linear array of elements. The array serves most but not all of the needs of the typical C program. The restriction is that an array is composed of elements all of which are of the same type. If we need to use a collection of different data type items it is not possible by using an array. When we require using a collection of different data items of different data types we can use a *structure*.

In this unit we will learn about structure and union. Here we will see how a structure and union are defined, declared and accessed in C programming language.

13.3 STRUCTURE

A structure is similar to records. It stores related information about an entity. With the use of structures, programmers can conveniently handle a group of related data items of different data types.

In C language, structure is basically a user defined heterogeneous data type. The main difference between a structure and an array is that an array contains related information of the same data type.

13.3.1 Structure Declaration

A structure is declared using the keyword ***struct*** followed by a structure name. All the variables of the structure are declared within the structure. The data types of all these variables within a structure can be of different types. A structure is generally declared with the following syntax:

```
struct struct_name
{
    data_type variable_name;
    data_type variable_name;
    ..........................................
```

```
        ..........................................
    };
```

For example, to keep the details of a book, we have to declare a structure for the book containing variables like title, author, pages, price, publisher etc. This book structure can be declared as:

```
    struct book
    {
        char title[20];
        char author[15];
        char publisher[25];
        int pages;
        float price;
    };
```

In the above declaration, the book name (i.e., title), author name and publisher name would have to be stored as **string**, and the page and price could be **int** and **float** respectively. The keyword *struct* declares a structure to hold the details of five fields namely title, author, publisher, pages and price. These are members of the structures. Each member may belong to different or same data type. It is not always necessary to define the structure within the *main()* function.

Structure declaration acts as a template which conveys structure information and member names to the compiler. Structure is a user-defined data type. Now, let us discuss how to declare structure variables.

We can declare structure variables using the structure name (tag name) any where in the program. For example, the statement,

struct book book1, book2, book3;

declares *book1, book2, book3* as variables of type *struct book*. Each declaration has five elements of the structure *book*. The complete structure declaration might look like this:

```
struct book
{
    char title[20], author[15], publisher[25];
    int pages;
    float price;
};
struct book
{
    char title[20], author[15], publisher[25];
    int pages;
    float price;
}   book1,book2,book3;
```

The use of tag name is optional. In the declaration, ***book1***, ***book2***, ***book3*** are structure variables representing three books. Tag_name is not included in this declation. A structure is usually defined before ***main()***. In such cases, the structure assumes global status and all the functions can access the structure.

Again, let us consider another structure declaration "employee".

```
struct employee
{
    char fname[15];
    char lname[15];
    int id_no;
    int month;
    int day;
    int year;
}   emp1;
```

Here we have declared one variable, ***emp1***, to be structure with six fields, some integers, some strings. Right after the declaration, a portion of the main memory is reserved for the variable ***emp1***. This variable takes a size of 38 bytes for different members of struct ***employee***: 15 bytes for fname, 15 bytes for lname, 2 bytes for id_no, 2 bytes for month, 2 bytes for day, 2 bytes for year.

13.3.2 Initialization of Structures

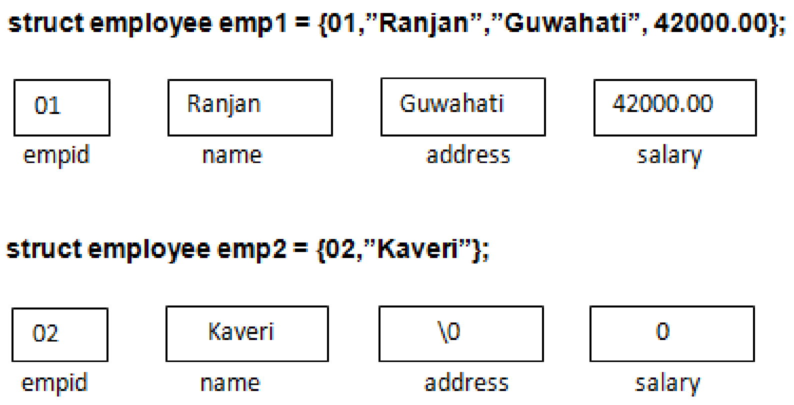
Initialization of structure means assigning some constants to the members of the structure. A structure can be initialised in the same way as other data types are initialized. The general syntax to initialize a structure variable is as follows:

```
struct struct_name
{
    data_type member_name1;
    data_type member_name1;
    data_type member_name1;
    ..........................................
    ..........................................
}   struct_var = {constant1, constant2, ....};
    or,
    struct struct_name
{
    data_type member_name1;
    data_type member_name1;
    data_type member_name1;
    ..........................................
    ..........................................
};  struct struct_name struct_var = {constant1, constant2, ....};
```

For example, let us initialize an employee structure as follows:

```
struct employee
{
    int empid;
    char name[20];
    char address[30];
    float salary;
}   emp1 = {01,"Ranjan","Guwahati", 42000.00};
    or by writing
    struct employee emp1 = {01,"Ranjan","Guwahati", 42000.00};
```

C language automatically initializes the structure members if the user does not explicitly initialize all the members. This is known as *partial initialization*. ***Integer*** and ***float*** members are initialized to ***zero*** and ***character arrays*** are initialized to '**\0**' (null value) by default. Pictorially we can represent it as follows:



```
struct employee emp1 = {01,"Ranjan","Guwahati", 42000.00};
```

```
struct employee emp2 = {02,"Kaveri"};
```

Fig. 13.1: Assigning valuesto a structure element

13.3.3 Accessing the Members of a Structure

The members of structure themselves are not variables. They should be linked to structure variables in order to make them meaningful members. The link between a member and a variable is established using the member operator '.' which is known as ***dot operator*** or ***period operator***. A structure member variable is generally accessed using the '.' dot operator. The syntax is:

struct_var . member_name;

For example, book1**.**price;

book1**.**pages;

book2**.**price;

book1.price is the variable representing the price of *book1* and can be treated like any other ordinary variable.

To assign value to the individual data members of the structure variable *book1*, we may write,

```
book1.price = 520.00;
book1.name = "Java";
book1.author="Kumar";
```

We can use *scanf()* function to input values for data members of the structure variable *book1* like this :

```
scanf(“%f”,&book1.price);
scanf(“%d”,&book1.pages);
```

For displaying the values of structure variable book1, we can use *printf()* function like this:

```
printf(“%f”, & book1.price);
printf(“%s”, &book1.author);
```

```
/*Program 13.1: Program to enter information of
one student and to display that information.*/
#include<stdio.h>
#include<conio.h>
void main()
{
   struct studentinfo
   {
      int roll;
      char name[20];
      char address[30];
      int age;
   }  s1;
   clrscr();
   printf("Enter the student information:");
   printf("\nEnter the student roll no.:");
   scanf("%d",&s1.roll);
   printf("\nEnter the name of the student:");
   scanf("%s",&s1.name);
   printf("\nEnter the address of the student:");
   scanf("%s",&s1.address);
   printf("\nEnter the age of the student:");
   scanf("%d",&s1.age);
   printf("\n\nStudent information:");
   printf("\nRoll no.:%d",s1.roll);
   printf("\nName:%s",s1.name);
   printf("\nAddress:%s",s1.address);
   printf("\nAge of student:%d",s1.age);
   getch();
}
```

EXERCISE

1) Modify the previous program(*Program 13.1*) for storing and displaying information of 2 students.

13.4 ARRAY OF STRUCTURES

In programming we may often need to handle many records. For example, in a class, there may be many numbers of students, say 50 students. So, to keep the records of 50 students we need an array of structures. This can be written as

```
struct student
   {
      int rollno;
      char name[20];
      char course[15];
      float fees;
   };
struct student s[50];     //s is an array of structure
```

Here, s is an array of structure of 50 students, each of which is of type *struct student*.

/*Program 13.2: Program for storing records of 50 students and displaying those records*/`

```
#include<stdio.h>
#include<conio.h>
void main()
{  struct student
   {
```

```
            int roll;
            char name[20];
            char address[30];
            int age;
         };
            struct student s[50];
            clrscr();
            int n, i;
            printf("\nHow many students information do
            you want to enter?");
            scanf("%d",&n);
            printf("Enter Student Information:");
            for(i=1;i<=n;i++)
         {
            printf("\nEnter Roll no.:");
            scanf("%d",&s[i].rollno);
            printf("\nEnter the name of the student:");
            scanf("%s",&s[i].name);
            printf("\nEnter the course of the student:");
            scanf("%s",&s[i].course);
            printf("\nEnter the dues of the student:");
            scanf("%f",&s[i].fees);
         }
            printf("\n\nInformation of all students:");
            for(i=1;i<=n;i++)
         {
            printf("\nRoll no.:%d",s[i].rollno);
            printf("\nName:%s",s[i].name);
            printf("\nCourse:%s",s[i].course);
            printf("\nSchool dues:%f\n\n",s[i].fees);
         }
         getch();
      }
```

CHECK YOUR PROGRESS

Q.1: Define a structure consisting of two floating point members, called *real* and *imaginary.* Include the tag *complex* within the definition. Declare the variables c1,c2 and c3 to be structure of type *complex*.

Q.2: Declare a variable ***"a"*** to be a structure variable of the following structure type:

```
struct   account{
        int ac_no;
        char ac_type;
        char name[30];
        float balance;
};
```

and initiaze ***a*** as follows:

```
        ac_no : 12437
        ac_type: Saving
        name: Rahul Anand
        balance: 35000.00
```

Q.3: State whether the following statements are True (T) or False (F)

i) Collection of different datatypes can be used to form a structure.

ii) Structure variables can be declared using the tag name any where in the program.

iii) Tag-name is mandatory while defining a structure.

iv) A program may not contain more than one structure.

v) We cannot assign values to the members of a structure.

vi) It is always necessary to define the structure variable within the main() function.

13.5 STRUCTURE WITHIN A STRUCTURE

A structure may be defined as a member of another structure. In such structures, the declaration of the embedded structure must appear before the declarations of other structures. For example,

```
struct date
{
    int day;
    int month;
    int year;
};
struct student
{
    int roll;
    char name[20];
    char combination[3];
    int age;
    struct date dob;        //structure within structure
}   student1,student2;
```

the structure ***student*** contains another structure ***date*** as one of its members.

13.6 PASSING STRUCTURES TO FUNCTIONS

Structure variables may be passed as arguments and returned from functions just like other variables. A structure may be passed into a function as individual member or a separate variable.

Passing individual members of structure to a function: To pass any individual member of the structure to a function as argument, we have to use the dot operator to refer to the particular member of the structure.

For example, a program to display the contents of a structure passing the individual elements to a function is shown below:

Program 13.3: Program to illustrate passing individual structure elements to a function.

```
      #include<stdio.h>
      #include<conio.h>
      void display(int, float);
      void main()
      {
         struct employee
         {
            int emp_id;
            char name[25];
            char department[15];
            float salary;
         };
         static    struct    employee    e1={15,
"Rahul","IT",8000.00}; clrscr();
         /* only emp_id and salary are passed to the
display function*/
         display(e1.emp_id,e1.salary);  //function call
         getch();
      }
      void display(int eid, float s)
      {
         printf("\n%d\t%5.2f",eid,s);
      }
```

Output: 15 8000.00

When we call the display function using *display(e1.emp_id,e1.salary);* we are sending the ***emp_id*** and ***name*** to function ***display( )***. It can be immediately realized that, passing individual elements would become more tedious as the number of structure elements increases. A better way would be to pass the entire structure variable at a time.

Passing entire structure to a function: There may be numerous structure members (elements) in a structure. Passing these individual elements as argument to a function would be a tedious task. Just like any other variable, we can pass an entire structure as a function argument. A structure is passed as an argument using the call by value method. This means a copy of each member of the structure is made. This method is very inefficient, especially when the structure is very big or the function is called frequently. Use of pointers in such sitiation is more suitable.

In the following program, we are passing a whole structure to a function.

//Program 13.4: Program to illustrate passing a whole structure to a function.

```
#include<stdio.h>
#include<conio.h>
struct employee
{
   int emp_id;
   char name[25];
   char department[10];
   float salary;
};
static struct employee
e1={10,"Palash","Sales",26000.00};
void display(struct employee e);//prototype
decleration
void main()
{
   clrscr();
   display(e1);    /*sending entire employee
structure*/
   getch();
}
void display(struct employee e)
{
printf("%d\t%s\t%s\t%5.2f",
e.emp_id,e.name,e.department,e.salary);
}
```

Output:
```
10    Palash     Sales     26000.00
```

```
      //Program 13.5: Program to illustrate structure
working within a function
      #include<stdio.h>
      #include<conio.h>
      struct item
      {
         int code;
         float price;
      };
      struct item a;
      void display(struct item i);    //prototype
decleration
      void main()
      {
         clrscr();
         display(a);  /*sending entire item
structure*/
         getch();
      }
      void display(struct item i)
      {
         i.code=20;
         i.price=299.99;
         printf("Item Code and Price of the
item:%d\t%5.2f", i.code,i.price);
      }
```

Output : 20 299.99

13.7 POINTER TO STRUCTURE

Instead of passing a copy of the whole structure to the function, we can pass only the address of the structure in the memory to the function. Then, the program will get access to every member in the function. This can be achieved by creating a pointer to the address of a structure using the indirection operator "*".

To write a program that can create and use pointer to structures, first, let us define a structure:

```
struct item
{
    int  code;
    float price;
};
```

Now let us declare a pointer to struct type *item*.

```
struct  item *ptr;
```

Because a pointer needs a memory address to point to, we must declare an instance of type *item*.

```
struct item p;
```

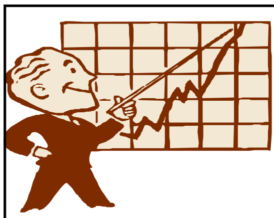
The following program shows the relationship between a structure and a pointer.

//Program 13.6: Program to demonstrate pointers to structure.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   struct item
   {
      int code;
      float price;
   };
   struct item i;
   clrscr();
```

```
        struct item *ptr;  //declare pointer to ptr
structure
        ptr=&i;         // assign address of struct
to ptr
        ptr->code=20;
        ptr->price=345.00;
        printf("\nItem Code: %d",ptr->code);
        printf("\tPrice: %5.2f",ptr->price);
        getch();
     }
```

Output: Item Code: 20 Price: 345.00



CHECK YOUR PROGRESS

Q.4: State whether the following statements are True (T) or False (F)

i) It is possible to pass a structure to a function in the same way a variable is passed.

ii) When one of the fields of a structure is itself a structure, it is called nested structure.

iii) We cannot create structures within structure in C.

iv) It is illegal for a structure to contain itself as a member.

v) A sstructure can include one or more pointers as members.

Q.5: Fill in the blanks:

i) __________ can be used to access the members of structure variables.

ii) The name of a structure is referred to as __________.

13.8 UNION

In some situations we may wish to store information about a person. The person may be identified either by name or by an identification number, but never both at the same time. We could define a structure which has both an integer field and a string field; however, it seems wasteful to allocate memory for both fields. This is particularly important if we are maintaining a very large list of persons, such as payroll information for a large company. In addition, we wish to use the same member name to access the information for a person.

C provides a data structure which fits our needs for the above scenario, called a ***union*** data type. A union type variable can store objects of different types at different times; however, at any given moment, it stores an object of only one of the specified types. Unions are also similar to structure data type except that members are overlaid one on top of another, so members of union data type share the same memory.

The declaration of a union type must specify all the possible different types that may be stored in the variable. The form of such a declaration is similar to declaring a structure data type. For example, we can declare a union variable, person, with two members, a string and an integer. Here is the union declaration:

```
union human
   {
      int id;
      char name[30];
   }  person;
```

This declaration differs from a structure in that, when memory is allocated for the variable ***person***, only enough memory is allocated to accommodate the largest of the specified types. The memory allocated for person will be large enough to store the larger of an integer or an 30 character array. Like structures, we can define a tag for the union, so the union template may be later referenced by the tag name.

Unions obey the same syntactic rules as structures. We can access elements with either the dot operator (.) or the right arrow operator (->). There are two basic applications for union. They are:

i) Interpreting the same memory in different ways.
ii) Creating flexible data structure that can hold different types of data.

//Program 13.7: Program demonstrating initializing union members and displaying the contents.

```
#include<stdio.h>
#include<conio.h>
void main()
{
   union data
   {
      int a;
      float  b;
   };
   union data d;
   d.a=20;
   d.b= 195.25;
   printf("\nFirst member is %d",d.a);
   printf("\nSecond member is %5.2f",d.b);
   getch();
}
```

Output:

```
First member is 16384
Second member is 195.25
```

Here only the float values are stored and displayed correctly and the integer values are displayed wrongly as the union only holds one value for one data type.

13.9 ENUMERATED DATA TYPES

In addition to the predefined types such as int, char, float etc., C allows us to define our own special data types, called enumerated data types.

An enumeration type is an integral type that is defined by the user. The syntax is:

enum typename {enumeration_list};

Here, ***enum*** is keyword, ***type*** stands for the identifier that names the type being defined and ***enumeration list*** stands for a list of identifiers that define integer constants. For example:

enum color {yellow, green, red, blue, pink};

defines the type *color* which can then be used to declare variables like this:

color flower=pink;

color car[ ]={green, blue, red};

Here, *flower* is a variable whose value can be any one of the 5 values of the type *color* and is initialialized to have the value pink.

//Program 13.8: Program to illustrate the concept of enumerated data type

```
#include<stdio.h>
#include<conio.h>
void main()
{
   enum month { jan, feb, mar, apr, may, jun,
jul, aug, sep,
   oct, nov, dec };
   month m;
   clrscr();
   for(m=jan;m<=dec;m++)
      printf("%d\t", m+1);
   getch();
}
```

Output :

```
1  2  3  4  5  6  7  8  9  10  11  12
```

In the above declaration, *month* is declared as an enumerated data type. It consists of a set of values, jan to dec. Numerically, jan is given the value 1, feb the value 2, and so on. The variable *m* is declared to be of the same type as month, m cannot be assigned any values outside those specified in the initialization list for the declaration of month.

13.10 DEFINING YOUR OWN TYPES (TYPEDEF)

Using the keyword ***typedef*** we can rename basic or derived data types giving them names that may suit our program. A typedef declaration is a declaration with typedef as the storage class. The declarator becomes a new type. We can use typedef declarations to construct shorter or more meaningful names for types already defined by C or for types that we have declared. Typedef names allow us to encapsulate implementation details that may change.

A typedef declaration is interpreted in the same way as a variable or function declaration, but the identifier, instead of assuming the type specified by the declaration, becomes a synonym for the type.

For example: **typedef unsigned long int ulong;**

The new type (***ulong***) becomes known to the compiler and is treated the same as ***unsigned long int.*** If we want to declare some more variables of type unsigned long int, we can use the newly defined type as:

ulong distance;

We can use the typedef keyword to define a structure as folllows:

```
typedef struct
          {
              type member1;
              type member2;
              ....
          }type_name;
```

type_name can be used to declare structure variables as follows:

type_name variable1,variable2,...;

CHECK YOUR PROGRESS

Q.6: Mentions the features of union data type.

Q.7: Mention the advantages of structure type over the union type.

13.11 LET US SUM UP

- Structure is basically a user-defined data type that can store related information together.
- The main difference between a structure and an array is that, an array contains related information of the same data type.
- A structure is declared using the keyword ***struct*** followed by a structure name. Elements of the structure are declared within the { and } brackets with their respective data types. For example, the following is a structure declaration for storing customer information.

```
struct customer
{
    int cid;
    char name[20];
    char address[30];
    long int mobile_no;
};
```

- Memory is allocated only when we declare variables of the structure. In other words, memory is allocated only when we instantiate the structure. For example, for the above structure customer, memory will be reserved when we declare structure variable.

```
        struct customer c;
```

- C permits the use of arrays as structure members.
- A member of the structure cannot be accessed directly using its name. We must use the structure name followed by the dot '.' operator before specifying the member name.
- In case of partial inialization, first few members of the structure are initialized and the uninitialized members are assigned default values.
- A structure can be placed within another structure.
- Unions are concept borrowed from structures and therefore they follow the same syntax.
- In structure, each member has its own storage location, where as

all members of a union use the same location.

- C allows us to define our own special data types, called enumerated data types.
- The keyword *typedef* is used to define a new data type of our own choice. We can use the typedef keyword to define a structure.

13.12 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.
2) Gottfried Byron, S. Programming with C.; Tata McGraw Hill Education.

13.13 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1:

```
struct complex
{
    float real, imaginary;
};
struct complex c1,c2,c3;
```

Ans. to Q. No. 2: Static struct account a={12437, “Saving”, “Rahul Anand”, 35000.00};

(*a* is a static structure variable of type account, whose members are assigned initial values.)

Ans. to Q. No. 3: i) True, ii) True, iii) False, iv) False, v) False, vi) False

Ans. to Q. No. 4: i) True, ii) True, iii) False, iv) True, v) True

Ans. to Q. No. 5: i) Pointers, ii) tag name

Ans. to Q. No. 6: The main characteristics of Union data type are:

a) The size of union is equal to the size of number of bytes occupied by the largest data member in it.

b) only one data member in union is active at a time.

Ans. to Q. No. 7: The structure data type can hold many data related to an entity like person, book, student etc., but a union type holds only one data active at a time.

13.14 MODEL QUESTIONS

Q.1: What is a structure? How is a structure different from an array?

Q.2: How is structure declared? Define a structure to represent a date.

Q.3: What is meant by array of structure?

Q.4: How are the data elements of a structure accessed and processed?

Q.5: Write a program in C to prepare the marksheet of a college examination and the following items will be read from the keyboard.

Name of the student,

Subject name,

Internal marks,

External marks

Prepare a list separately of those students who failed and passed in the examination.

Q.6: What is meant by union? Differentiate between structure and union.

Q.7: What is the purpose of typedef feature? How is this feature used with structure?

Q.8: Write short notes on:

a) Enumerated data type

b) Type definition.

Q.9: Write a C program to read and display the information of all the students in a class.

*** ***** ***

UNIT 14: FILE HANDLING

UNIT STRUCTURE

14.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- learn about various file operations
- read data from a file
- write data to a file
- describe various input/output functions for file operations

14.2 INTRODUCTION

In the previous units, we have used *scanf()*, *printf()* functions to read and write data. These functions are console oriented input/output (I/O) functions. For such functions there is always a need of a keyboard for reading inputs and a monitor to display the output. This works fine as long as the input/output data is small enough to read and write.

However in some situations, huge amount of data have to be read and written and the console oriented I/O function cannot handle it efficiently.

This is because the entire data is lost when the program is terminated or the computer is terminated or the computer is turned off. Therefore, it is necessary to have a more flexible approach where large amount of data can be stored permanently in disks and we can read them whenever required. The concept of file helps us to store and handle this type of data easily. In this unit we are going to discuss handling of files in C.

14.3 FILE SYSTEM

A ***file*** is a storage place where a group of related data is stored. C language supports a number of functions that have the ability to perform the basic file operations like:

- naming a file
- opening a file
- reading data from a file.
- writing data to a file.
- closing a file.

There are two ways to perform the file operations in C:

a) Low level Input/Output

b) High level Input/Output

The low level disk I/O functions are more closely related to the computer's operating system and it uses UNIX system calls.It is harder to program for general users as compared to the high level I/O. However lower level I/O functions are more efficient both in terms of operation and the amount of memory used by the program.

The high level I/O functions are more commonly used in C programs and are easier to use than low level I/O functions. We are going to discuss high level I/O functions in this unit. Let us take a brief overview of Disk I/O functions using the figure given below.

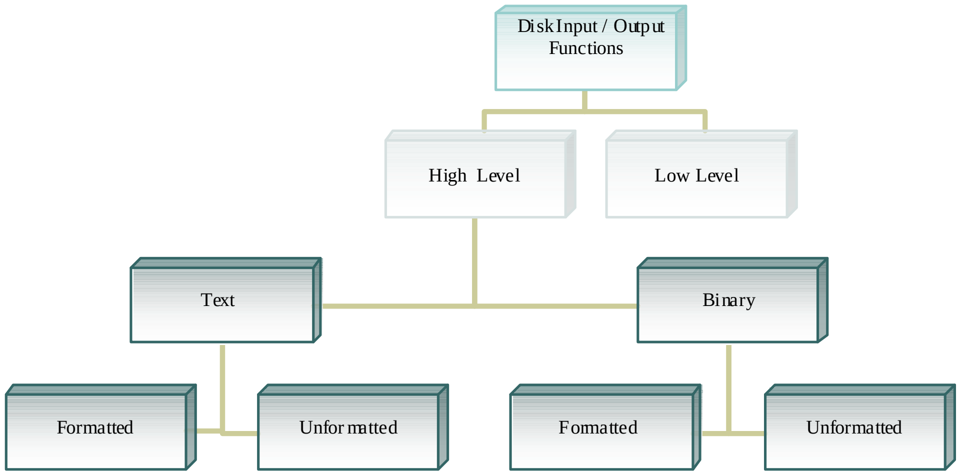


Fig 14.1: Disk I/O Functions

The High level input/output functions includes the following functions which will be discussed broadly in the next section.

i) **fopen():** fopen() function is used for opening an existing file and also used to create a new file.

ii) **fclose():** fclose() function is used to close a file which has been already opened.

iii) **getc():** getc() function is used to read single characters from a file.

iv) **putc():** putc()function is used to write single characters to the file.

v) **fputs():** fputs() function is used to writes strings to the file.

vi) **fgets():** fgets() function is used to read strings from file

vii) **fprintf():** fprintf() function is used to write a set of values to a file.

viii) **fscanf():** fscanf() funcyion is used to read a set of values from a file.

Files are used to store data in the secondary memory permenantly.

Let us look at some of the basic file information associated with files like:

a) naming the file (i.e., file name)

b) data structure that links the file. and

c) purpose of opening the file.

a) File name is a string or group of characters. It may contain two parts: a primary name and an optional period with the extension. Some examples of file names are:

i) add.c (Here **add** is primary part and **.c** is the extension)

ii) kkshou.c

iii) output.txt

b) The data structure of a file is defined as **FILE** where **FILE** is a kind of data type.

c) The purpose of opening a file can be to perform read or write operation on the files.

Note: All the files should be declared as type **FILE** before they are used.

14.4 OPENING AND CLOSING FILE

Opening a File: The pupose of opening a file was mentioned in the previous section. In addition to reading and writing a new file, we can also add data to a already existing file. Let us now try to understand how we can open and close any given file. The general syntax for opening a file is illustrated in the following program segment:

```
FILE *fptr;
fptr=fopen("filename","mode");
```

In the above program segment, the first statement declares the variable **fptr** as a pointer to the data type **FILE**. The second statement opens the file named *"filename"* and asigns the identifier to the FILE type pointer **fptr**; *mode* specifies the purpose of opening the file. Depending upon the mode, a file can be used for one of the specific purposes as listed below.

Mode	Meaning
"r"	Opens an existing file for reading purposes
"w"	Opens a new file for writing only. If a specified file name already exists, it will be destroyed and a new file is created in its place.
"a"	Opens an existing file for adding new data at the end of a file. New file is created if there is no existing file with that filename.

"r+"	Opens a file for both reading and writing purposes.
"w+"	Open a file for both reading and writing. If file name already exists, it will be destroyed and a new file is created in its place.
"a+"	Open a file for both reading and appending. New file is created if the file does not exist.

Return Value: On successful execution, fopen() returns a pointer to opened stream. As shown above, the return value is assigned to **fptr**. If the specified file could not be opened successfully, then it returns **NULL**; sometimes fopen() may fail in opening a file with a specified mode because of some unwanted reason. This can be easily checked by comparing value of **fptr** with **NULL**. This is illustrated with the following program segments:

```
FILE *fptr;
fptr=fopen("data.txt","r")
if(fptr==NULL)
        printf("Error in opening the file ");
else
        printf("Sucessfully opening the file ");
```

EOF: EOF is a constant defined in the header file stdio.h.

In the above program segment, if ***"data.txt"*** does not exist, then the value of ***fptr*** will be NULL and the output will be:

```
Error in opening the file
```

Closing File: It is always a good programming practice to close the file using fclose() as soon as all operations on the file have been completed. This ensures that all the information associated with the file is cleared out from the *buffer,* thus preventing any accidental misuse of the file. Closing the file releases the space taken by the FILE structure which is returned by the fopen().

The general syntax of fclose() is:

```
int fclose(file_pointer)
```

Here, fclose() function closes the file and returns zero on success or **EOF** if there is an error in closing the file. EOF stands for end of file.

14.5 INPUT/OUTPUT OPERATION ON FILES

The function **fopen()** can be used for opening a file for reading or writting purposes. To read data from a file, we need a mechanisim to read the contents of the file.

14.5.1 getc() and putc()

The function **getc()** is used to read the contents of a file which is opened by the ***fopen()*** function. A file pointer is associated with *fopen()* function after successful opening of the file which always points the first character of a file. For example, let us consider the following statement:

ch= getc(fptr)

In the above statement, *getc()* reads a character from the current position of the file, increments the pointer position so that it points to the next character. Then it returns the character that is read and is collected in the variable **ch**.

A question that may arise in our mind now is that while reading from the file continously, how can we determine whether that the file has been completly read or if we have reached the end of the file. Generally, we say the file has reached its end, if there is no character to be read. The function ***getc()*** returns **EOF** or **-1** when the end of file is reached. So we can check if we have reached the end of the file by comparing the value of **ch** with **EOF**.

Similarly, **putc()** can be used for writting data to a file character by character. The file must be opened in *write* mode using *fopen()* before writing data into the file. For example,

putc(ch,fptr)

In the above statement, **ch** is a character to be written to the file referrenced by the file pointer **fptr**.

Program 14.1: Write a program to read the contents of a already existing file which

```
contains some data. Display the data in the
file.
#include<stdio.h>
#inlcude<conio.h>
void main()
{
   FILE *fptr ;         // File pointer
declaration
   char ch;
   fptr=fopen(“kkhsou.org”,”r”); // file open
for reading
   if(fptr==NULL)       // check the file is
opened or not
      printf(“\n Error in opening file”);
   else
   {
   do
   {
      ch=getc(fptr);
      printf(“%c”,ch);   // display to the
monitor screen
   }while(ch!=EOF);
   }
   fclose(fptr);        // closes the file
   getch();
}
```

In this example, we have considered a already existing file named “kkhsou.org”. We want to read and display the contents of this file. At first we have to open the file in reading mode and then read it using *getc()* until the end of file(EOF) is encountered. While reading from the file, we display the contents simultaneously to the monitor. We need to close that file using *fclose()*.

Program 14.2: Write a program to read your name, roll number from the keyboard and then write it to a file named “address.txt ”.

```
#include<stdio.h>
#iinclude<conio.h>
void main()
{
   FILE *fptr;
   char ch;
   fptr=fopen(“address.txt”,”w”);     // open
file  writting
   if(fptr==NULL) // checking whether file is
                  //succesfully opened or not
      printf(“\n Error in opening file”);
   else
   {
      printf(“\nEnter Your Name and Roll
Number”);
      do{
      ch=getchar();   // read from the keyboard
character
                      // by character
      putc(ch,fptr);  // write it to address.txt
refferenced by fptr
   } while(ch!=’\n’); // untill  we press ENTER
key (newline)
   }
   fclose(fptr);
   getch();
}
```

Output: Enter Your Name and Roll Number **Nayan 4** [press enter]

In the above program, we are asked to read our name, roll number directly from the keyboard and then to write it to the file *named “address.txt”*. We can read the name and roll number using *scanf()* or *getchar()* functions or can use other input functions.Once we finish reading we need to open the file *“address.txt”* in write mode to write the data to the file.

In the output, we have entered “Nayan 4”, which is then written to the file *“address.txt”* and we can open the file directly to view the output. In C language ‘**\n**’ is used to denote a new line character. In the above program when we encounter ’\n’ in gecthar() then we stop the loop, thinking that the user has finished entering his data. However, there is another way of reading more than one line.

Program 14.3: Write a program to copy a file named “first.txt ” to another file named “second.txt”. Assume that file *first.txt* already exists.

```
#include <stdio.h>
#include <conio.h>
void main()
{
   FILE *fp1,*fp2; //  Since we need two file
   char ch;
   fp1=fopen(“first.txt”,”r”);
   fp2=fopen(“second.txt”,”w”);
   // checking whether two file succesfully opend
or not
   if(fp1==NULL && fp2==NULL)
      printf(“\n Error in Opening the File”);
   else
   {
   do
   {
      ch=getc(fp1); // read from the first file
      putc(ch,fp2); // write to the second file
   }while(ch!=EOF);
```

```
    }
    fclose(fp1);
    fclose(fp2);
    getch();
}
```

In this example, since the file *first.txt* aready exists and we simply need to write a program that will copy the content of *“first.txt”* to *“second.txt”.* We need to open the *“first.txt”* file in *read* mode and then *second.txt* file in *write* mode.

14.5.2 fputs() and fgets()

In the previous section, we have discussed how to read from the files and also how to write to the file. But we have been reading or writting to the files only character by character using loops.We can also read or write to the files as strings, using *fputs()* and *fgets()*. Functions like *gets()* and *fputs()* are related to only keyboards input/ output, whereas *fgets()*,*fputs()* are related to files. The ***fputs()*** **f**unction writes a string to a file. For example, say if **fptr** is a file pointer which opens a file for writting, then :

```
fputs(“Welcome to KKHSOU”, fptr) ;
```

The sentence “Welcome to KKHSOU” is written to a file pointed by fptr. We can also use fputs() in following manner:

```
char str[10];
gets(str);          // read from the keyboard
fputs(str,fptr);    // write to the file
```

So, the general way of representing fputs() is:

```
fputs(strings, file_pointer);
```

fgets() can be used for reading strings from a file. The general way of representing fgets() is:

```
fgets(strings,no_bytes,file_pointer);
```

The above statement reads a strings having (no_bytes-1) from a file pointed by file_pointer. “no_bytes” indicates how many bytes of strings are to be selected. We give -1 here because the file always read from starting 0 for the first character.

Program 14.4: Write a program to read a line from the keyboard and then write it to a file named *"output.txt".*

```
#include<stdio.h>
#include<conio.h>
void main()
{
   char line[80];
   FILE *fp;
   fp=fopen("output.txt","w");
   if(fp==NULL)
   puts("Error in opening the file ")
   else
   {
      puts("Enter a line: ");
      gets(line); // read from keyboard
      fputs(line,fp); // write to the file
     }
   fclose(fp);
   getch();
   }
}
```

Output: Enter a line: **Hello learners** (press enter key)

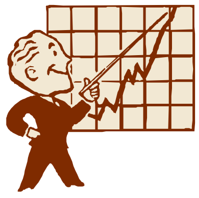
In this example, we are asked to read a few lines using keyboard, so we will need a character array to store the lines read. Then we have to open a file *"output.txt"* in write mode and write the lines (character array) to it. The text *"Hello learners"* in this example is given by the user from the keyboard is written to the file output.txt.

Program 14.5: Write a program to read from a file then display it to the monitor screen.

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

```
void main()
{
   FILE  *fp;
   char strngs[80];
   fp=fopen(“output.txt”,”r”);
   if(fp==NULL)
      puts(“Error in opening file”);
   else
   {
      while(fgets(strngs,79,fp)!=NULL)
      puts(strngs);
   }
   fclose(fp);
}
```

In the above program we have read from the existing file and then displayed it to the output screen i.e., monitor. Firstly, we open an existing file “*output.txt*” assuming that the file exists with some content. Then we use **fgets()** function to read the file. We have used a character array of size 79 in fgets(), since there can be a maximum of 80 characters in a line of a file. After reading from the file we store the variable **strngs** array using fgets() and then we display it to the output screen using puts().



CHECK YOUR PROGRESS

Q.1: State whether the following statements are True or False:

a) fopen() is used to open and close a file.

b) File concept is used mainly to store data permanently and to use them later.

c) fopen(“file.c”,”w”) opens file.c for reading purposes only.

Q.2: Fill in the blanks:

a) putc() is used to ___________ to a file.

b) fgets() is mainly used for __________ strings.

c) fopen() returns __________ on error.

14.5.3 fprintf() and fscanf()

We have till now discussed how to read or write a single character from a file using getc(), putc(). We have also discussed about reading and writing the strings using fgets() and fputs() to the files. All these functions are related to a particular data type i.e. characters. For writing and reading data to the files having different data type variables, we can use **fprintf()** and **fscanf()** respectively. These functions are mostly same as printf() and scanf() function that we have used in earlier programs except that fprintf() and fscanf() are used for writing and reading from the **files**. The general form of fprintf() is:

fprintf(fptr,"control string",list);

where fptr is a file pointer associated with the file opened for writing. The control string contains output specification for the items in the list. The list may include variables, constants and strings. We will get a clear idea of control string when we explain the examples. For example,

fprintf(fptr,"%s,%d,%f",name,roll,per);

where **name** is an array variable of type character and **roll** is integer variable and **per** is a float variable.These different data type variables are written to the file pointed by *fp*. The general format of fscanf() is:

fscanf(fptr,"control string",list);

The above statement causes the reading of the items in the list from the file specified by the fptr, according to the specifications contained in the control string. For example,

fscanf(fptr,"%s,%d,%f",name,&roll,&per);

where name, roll and per are read from the file specified by **fptr**.

Program 14.6: Write a program to read name,roll number from the keyboard for a student and then write it to a file using fprintf(). Then

```
use fscanf() to read the from the file written
by the code in the above program segment.
#include<stdio.h>
void main ()
{
   char name[10];
   int roll_no;
   FILE *fptr;
   /*Open the file */
   fptr=fopen("out.txt","w");
   if(fptr==NULL)
      printf("Error in opening file ");
   else
   {
      /*Read from the keyboard */
      printf("Enter name ");
      scanf("%s",name);
      printf("Enter roll no ");
      scanf("%d",&roll_no);
      /* Write to the file */
      fprintf(fptr,"%s %d",name,roll_no);
   }
   fclose(fptr);
}
```

In the above program the name and roll number given by the user is written to the file **“out.txt”**.

In the next program, we open the file in read mode and then read the data using fscanf(). After that we may display it using printf() function in order to view the result.

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

```
void main()
{
   char name[10];
   int roll_no;
   FILE *fptr;
   fptr=fopen("out.txt","r");
   if(fptr==NULL)
      printf("Error in opening file ");
   else
   {
      /* Read from  file */
      fscanf(fptr,"%s %d",&name,&roll_no);
      /* Write to the screen */
      printf("\nName:   %s   Roll   No::
%d",name,roll_no);
   }
   fclose(fptr);
}
```

14.6 SEEKING FORWARD AND BACKWARD

We have till now, discussed about reading/writing files only from the starting position. But sometimes we may encounter situations where we have to read or write files at random positions. Some basic functons that can be used for reading or writing files at random positions are explained briefly below:

a) **ftell():** The **ftell()** function gives the current positions of a particular file. The general syntax for **ftell()** function is given below.

n = ftell (fptr)

The return value of **n** is in bytes which indicates that **n** bytes have already been read or written.

b) **rewind():** The **rewind()** function is used to reset position of the file pointer to the start of the file. For example,

rewind(fptr);

n=ftell(fp)

Here, because of the rewind() function file pointer, position is set to the starting of the file and therefore the value of **n** would be 0. Remember that first byte in the file is numbered as 0, second as 1 and so on.

c) **fseek():** This **fseek()** function is used to move the file position to a desired location within the file. General syntax:

fseek (file_ptr, offset , position);

Here, **file_ptr** denotes the file to be processed, **offset** is a variable of type long which specifies the number of position to be moved from the location specified by the **position.** The value of **position** can be any one of the following:

Value	**Meaning**
0	Begining of the file.
1	Current position
2	End of file

The value of **offset** may be positive which means that file pointer is to be moved forwards or negative meaning that it is to be moved backwards. Let us try to understand the meanings using the statements given below.

Statement	**Meaning**
fseek (fptr, 0L,0)	Go to the begining of the file.
fseek (fptr, 0L,1)	Stay at the same i.e current position.
fseek (fptr, 0L,2)	**Go to end of the file.**
fseek (fptr, m,0)	**Move m+1 bytes from** begining**.**
fseek (fptr, m,1)	Go forward by m bytes from current positions**.**
fseek (fptr, -m,1)	Go backward by m bytes from current positions.
fseek (fptr, -m,2)	Go backward by m bytes from end.

The function **fseek()** returns zero after successful operations and-1 otherwise.

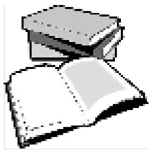
CHECK YOUR PROGRESS

Q.3: Give the syntax of ftell().

Q.4: What file functions can be used to read/write different data type variables from file?

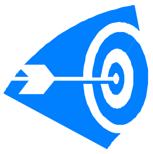
14.7 LET US SUM UP

- File concept is generally used to store data permanantly and to use it later. Opening, Reading, Writing Closing etc. are some of the operations that can be performed on files.
- By specifying the modes in fopen() function we can treat the file in various ways such as: file can read (e.g **r**) or write(e.g **w**) or both(e.g **r+**) etc.
- If the specified file does not exist or there is error in the file while opening, then fopen() function returns NULL.
- putc(), fputs() and fprintf() are used mainly for reading data from a file pointed by a file pointer. Similarly, getc(), fscanf(), fgets() are used for reading information from the file.



14.8 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw Hill Education.
2) Gottfried Byron, S. *Programming with C*; Tata McGraw Hill Education.



14.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: a) False, b) True, c) False.

Ans. to Q. No. 2: a) write a character, b) reading, c) -1

Ans. to Q. No. 3: The general syntax for **ftell()** function is:

n = ftell (fptr),

where, the return value of **n** is in bytes which indicates that **n** bytes have already been read or written.

Ans. to Q. No. 4: **fprintf()** and **fscanf()**

14.10 MODEL QUESTIONS

Q.1: Explain the functions of fopen() and fclose() with one example each.

Q.2: Write the name of two file functions for input operations.

Q.3: Give the main difference between getc() and fgets() functions.

Q.4: Mention the various modes used with the fopen() functions.

Q.5: Explain briefly: ftell(), rewind(), fseek().

Q.6: Write C programs to perform the following:

a) To read your name, roll and percentage from the keyboard and then write it to a file.

b) To copy a file to another file using getc().

c) To count number of vowel, blank space from an existing file.

*** ***** ***

UNIT 15: PREPROCESSOR DIRECTIVES

UNIT STRUCTURE

15.1 LEARNING OBJECTIVES

After going through this unit, you will able to:

- define preprocessor directives
- learn about macro substitution
- define macro, parameterised macro etc.

15.2 INTRODUCTION

In this unit we will learn about preprocessor directives available in C language. We will also learn to define macros and invoke them from C program. The preprocessor is a program that processes the source code before it passes through the compiler. It operates under the control of the preprocessor directive which is placed in the source program before the *main( )* function. A C preprocessor instruct compiler to do required pre-processing before actual compilation.

Besides macros, we will also learn about file inclusion and conditional directives in this unit.

15.3 PREPROCESSOR

A unique feature of C language is the preprocessor. The prepocessor is a collection of special statements, called *directives,* which are executed at the begining of program compilation. Before the source code is passed through the compiler, it is examined by the preprocessor for any preprocessor directives. In case the program has some preprocessor directives, appropriate actions are taken depending on the directives and the source program is handed over to the compiler. The general rules for defining a preprocessor are as follows:

a) Preprocessor directives are always preceded by a hash sign (#).
b) They must start in the first column.
c) Preprocessor directive should not be terminated by semicolon (;).
d) There should be only one preprocessor directive in one line.

The advantage of using preprocessor directives in a C program are as follows:

- The program becomes readable and easy to understand.
- The program becomes portable.
- The program can be easily modified.
- As a result, the program becomes more efficient to use.

Some examples of preprocessor directives in C language are:

Directive	***Function***
#define	Defines a macro substitution
#undef	Undefines a macro
#include	Specifies the files to be included
#ifdef	Test for a macro definition
#endif	Specifies the end of #if
#ifndef	Tests whether a macro is not defined
#if	Tests a compile time condition
#else	Specifies alternatives when #if test fails

a) #define: C language allows defining an identifier having constant value using #define directive. The #define statement is also called *macro definition* or simply a *macro*. This directive is placed at the begining of a C program. The symbol # occurs at the first column and no semicolon is allowed at the end. The general syntax for defining a macro is as follows:

```
#define identifier string
```

The preprocessor replaces every occurence of the identifier in the source code by the string. For example, few macro directives are:

```
#define PI
#define ROWS
#define COLS
```

Program 15.1 shows the use of macro directive. In the program, the statement `#define PI  3.14` defines a macro PI as an abbreviation for the token 3.14.

`/*Program 15.1: Program to illustrate the use of #define directive*/`

```
#include<stdio.h>
#define PI 3.14
void main()
{
   float r= 5.25;
   float area;
   area = PI * r * r;
   printf("\n Area of a Circle = %f", area);
}
```

In the above program, PI will be replaced by its value 3.14, which is defined using #define.

b) **#include (File Directive):** The *#include* directive causes one file to be included in another file. With *#include* directive, an external file containing functions, variables, or macro definitions can be included as a part of our program. This avoids the effort to rewrite the code that is already written. The *#include* directive is used to inform the preprocessor to treat the contents of a specified file as if those contents had already appeared in the source program at the point where directive appears. The general format is :

```
#include <filename>
```

When we include a file using angular brackets, a search for thefile named “filename” in a standard list of system directories.

For example, *#include<stdio.h>,* which appears at the start of a program. This #include statement causes the contents of the file *stdio.h* to be inserted into the program at the start of the compilation process. The information contained in the file *stdio.h* is essential for the proper functioning of the library functions such as *getchar()*, *printf()*, *scanf()* and *putchar()* etc.

c) **#undef :** Sometimes it is essential to undefine a macro that is already defined by #define. This can be accomplished by #undef directive. The #undef removes a macro definition from the macro symbol table. Once a macro name is undefined, the name the macro ceases to exist from the point of undefinitionand the preprocessor directive behaves as if it had never been a macro name. Undefining a macro means to cancel its definition.

In the following example the #undef directives removes whatever is defined with #define.

```
#define MAX 100
.
.
.
#undef MAX
```

d) **#Conditional Directive:** The preprocessor conditional compilation command allow lines of source text to be parsed through or eliminated by the preprocessor on the basis of a computed condition. Some examples of preprocessor conditional commands are **#if, #endif, #else, #elif** etc.

- The **#if** directive is used to test whether an expression evaluates to a nonzero value or not. While using #if directive in a program, we should make sure that each #if directive must be matched by a closing #rndif directive. Let us look at the statements below:

```
#if MAXMARKS >= 40
     Statement1;
     Statement2;
#else
     Statement4;
     Statement5;
#endif
```

- The **#elif** directive is used when there are more than two possible alternatives. The **#elif** directive is embedded within the #if directive and has the following syntax:

```
#if condition
       Statement1;
 #elif new_condition
        Statement2;
 #else
        Statement2;
 #endif
```

- The **#else** directive can be used within the controlled text of a #if directive to provide alternative text to be used if the condition is false.

```
#if condition
       Statement1;
 #else
       Statement2;
 #endif
```

- The **#endif** directive ends the scope of the #if, #else, or #elif directives. The number of #endif directives required depends on whether the #elif or #else directive is used.

15.4 MACRO SUBSTITUTION

Constant values or expressions can be identified using symbolic names. At the time of preprocessing, all these symbolic names are replaced by values/ expressions. This process is referred to macro substitution. The general form for a simple macro definition is:

```
#define  macro-name  value or string
```

It should be remembered that there is no white space(blank) between **#** and **define**, but single white space are used to separate the identifier i.e., **macro-name** and the **value** or **string**.

The **value** of the macro does not end with a semicolon. The preprocessor replaces every occurrence of a simple macro in the program text by a copy of the body of the macro, except that the macro name that are within comments or string constants. It is not appropriate to end a macro with a semicolon because the macros are used within expressions in the body of the program. Macros that represent single numeric, string or character values can also be referred to constants. Some examples of simple macro definitions are given below:

```
#define  PI  3.1415926                /* the value of Pi */
#define ELECTRON  9.107e-28           /*mass of an electron
                                      at rest in grams */
#define  PROTON 1837 * ELECTRON       /*mass of a proton at
                                      rest in grams */
#define  N 50
#define  TITLE “ABC & Co”
```

In the last two example, the constant value 50 is identified by N and “ABC & Co” is identified by TITLE which are actually considered as strings. At the time of preprocessing, the value 50 and “ABC & Co” are substituted in the place N and Title. The identifier N and Title are considered as macro.

The **#define** directive can also be used for defining parameterized macros. The general form for defining a parameterized macro is:

```
    #define macro-name(p1, p2,....)body-of-macro
```

Here, p1,p2, ..etc. are parameters. Parameterized macros are primarily used to define functions that expand into in-line code. Some examples of parameterized macro definitions are:

```
     #define    ABS(N)     ( (N) >= 0 ? (N) : -(N) )
     #define    READ(I)    scanf( “%d”, &I )
```

Let us consider a simple C program to illustrate the use of parameterized macro.

```
/*Program 15.2: Program to illustrate the use of macro
     #include<stdio.h>
     #define SQR(x)  x*x
     void main()
     {
        int result;
        int a=5, b=6;
        result = SQR(4);
        result = SQR(a+b);
        printf(“Result: %d”,&result);
     }
```

Output: 41

In the above example, the passing of **a+b** to the macro results in the following expanded code:

result = a+b * a+b;

which is evaluated as **a + (b * a) + b** which will not give the expected answer. The expected answer was **121** while output we recieved was **41**.

The problem, ofcourse, is with the evaluation of the operators involved in with the expanded macro. The multiplication operator is evaluated before the addition operator which makes the result 41. Even by adding parentheses as shown below the same result is produced.

```
#define SQR(x) (x * x)
```

This is because when a+b is passed it expands to **(a + (b*a) + b)** which again does not give us the desired result. Therefore, more parenthesies are required to get the desired result.

```
#define SQR(x) ((x)*(x))
```

The above macro definition will finally give the desired result to the macro expansion.

CHECK YOUR PROGRESS

Q1. Write down the rules for defining a preprocessor.

Q2. Give the syntax for defining macro.

15.5 LET US SUM UP

- Preprocessor is a program that processes the source code before it passes through the compiler.
- A programmer can use preprocessor to make the program easy to read, portable and more efficient.
- The preprocessor directives are always preceded by a hash sign (#).
- To define processor macros, we use *#define* directive.
- A macro is a simple identifier which will be replaced by a code fragment.
- The *#include* directive causes one file to be included in another file.
- The *#include* directive is used to inform the preprocessor to treat

the contents of a specified file as if those contents had appeared in the source program at the point where the directive appears.

- The *#undef* directive undefines or removes a macro name previ ously creadted with *#define*.
- The *#if* directive is used to control the compilation of a source file.

15.6 FURTHER READING

1) Balagurusamy, E. (2002); *Programming in ANSI C*; Tata McGraw-Hill Education.
2) Gottfried Byron, S; *Programming with C*; Tata McGraw-Hill Education.

15.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1: The rules for defining a preprocessor are as follows:

a) All the preprocessor directives begin with hash (#) sign.
b) They must start in the first column.
c) The preprocessor directive should not be terminated by semicolon (;)
d) There should be only one preprocessor directive in one line.

Ans. to Q. No. 2: The general form for a simple macro definition is:

#define macro-name value

15.8 MODEL QUESTIONS

Q1. Define a macro. Summarize the similarities and differences between macros and functions.

Q2. Give the syntax and use of preprocessor directives *#include* and *#define*.

Q3. What is the scope of a preprocessor directive within a program file?

Q4. Can we have a C program that does not use any preprocessor directive?

Q5. Why should we incorporate preprocessor directives in our programs? Give at least one example to support your answer.

Q6. Explain the importance of the #define preprocessor directive.

Q7. How can #include directive be used in your program?

Q8. Can the #undef directive be applied to a macro name that has not been previously defined? If yes, why?

*** ***** ***