GCA S2 03

Data Structure through C Language

SEMESTER - II

BACHELOR OF COMPUTER APPLICATION

BLOCK - 1



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

Course Coordinator

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

SLM Preparation Team

Units	Contributor
1	Arabinda Saikia, KKHSOU
2, 6	Dr. Khirod Sarma, NERIM
3, 4	Sruti Sruba Bharali, KKHSOU
5, 7	Dr. Tapashi Kashyap Das, KKHSOU

Editorial Team

Content Dr. Jyotiprokash Goswami, Assam Engineering College

Language Prof. Robin Goswami, Retd.Professor, Cotton College

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

Jan, 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

City Office : Housefed Complex, Dispur, Guwahati-781 006; 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.

BACHELOR OF COMPUTER APPLICATION

DATA STRUCTURE THROUGH C LANGUAGE

DETAILED SYLLABUS

Block-1

Unit-1 : Introduction to Data Structure **Pages** 1-25

Basic concept of data, Data type, Abstract Data Type, Types of Data Structure, Dynamic Memory Allocation

Unit-2 : Algorithms **Pages** 26-35

Algorithm, Complexity, Time-Space Complexity, Algorithmic Notation

Unit-3 : Arrays **Pages** 36-57

Arrays: Types, One dimensional array, Multidimensional arrays, memory representation, address translation functions, Applications of array, Random accessing of array element

Unit-4 : Operations on Arrays **Pages** 58-78

Traversal, Search, Problems associated with Insertion and Deletion, Sorting, Reversing, Merging

Unit-5 : Recursion **Pages** 79-94

Function, Rules for Recursive Function, Recursion versus Iteration, Advantages and Disadvantages

Unit-6 : Linked List **Pages** 95-102

Introduction to Linked List, Important terms associated with Linked list, Representation of Single linked list using Static and Dynamic Memory Allocation. Circular linked list, doubly linked list, Applications of Linked List

Unit-7 : Operations on Linked List **Pages** 103-122

Insertion of node in a linked list (at Start, at End, at a Given Position), deletion of node in a linked list (at Start, at End, at a Given Position), searching and traversal of elements and their comparative studies with implementations using array structure, Reversing a singly lined list.

COURSE INTRODUCTION

Data structure provides different ways of organising the same data. The purpose of learning about different data structures in C language is so that we can store data in an efficient way. Data structures are also needed to perform different functions on the data and to retrieve data in an efficient way. Choosing the correct data structure for the data to be stored is one of the essential problems in C programming language. This course gives us a detailed information on some of the most common data structures used in C language.

This course on **Data Structure Through C Language** comprises of fourteen units and seeks to present an overview of the different data structures that can be used in C language. This course includes an introduction to data struture and algorithms. Array, linked list, stack, queue data structures and the different operations that can be performed on these data structures are also discussed in this course. The concepts of searching and sorting are also introduced in this course. Apart from these, advance data structures like trees and graphs have also been included in this course.

The course is divided into two blocks:
Block 1 introduces us to different types of data structures. The concepts of algorithms and recursion are discussed in this unit. Two most common and basic data structures arrays and linked list and the operations that can be performed on these data structures are discussed in this block.

Block 2 concentrates on the data structures like stacks and queues. Different searching and sorting algorithms are introduced in this block. In addition to this, advance data structures like trees and graphs are also included in this block.

Each unit of these blocks includes some along-side boxes to help you know some of the difficult, unseen terms. Some “ACTIVIITY” have been included to help you apply your mind. You may find some boxes marked with: “LET US KNOW”. These boxes will provide you with some additional interesting and relevant information. Again, you will get “CHECK YOUR PROGRESS” questions. These have been designed to self-check your progress of study. It will be helpful for you if you solve the problems put in these boxes immediately after you go through the sections of the units and then match your answers with “ANSWERS TO CHECK YOUR PROGRESS” given at the end of each unit.

BLOCK INTRODUCTION

This is the *First Block* of the course '***Data Structure Through C Language***'. After completing this block, you will be able to use different data structures for diffrent data efficiently.

This block comprises of the following seven units:

Unit - 1 introduces you to the concept of data structures and different types of data structures.

Unit - 2 concentrates on the concept of algorithms and complexity. Concept of time and space complexity are also introduced in this unit.

Unit - 3 introduces the concepts of arrays. Different types of arrays like one dimensional array, multidimensional arrays are discussed in this unit.

Unit - 4 concentrates on the operations performed on arrays. This unit includes operations on array like insertion, deletion, sorting, merging and reversing of data elements.

Unit - 5 introduces concepts on recursion and iteration.

Unit - 6 introduces the concepts of linked list. Representation of linked list using static and dynamic memory allocation is included in this unit. Different types of linked list are also included in this unit.

Unit - 7 concentrates on the operations performed on single linked list. This unit includes operations on linked list like insertion, deletion, sorting, merging and reversing of data elements.

The structure of Block 1 is as follows:

UNIT 1 : Introduction to Data Structure

UNIT 2 : Algorithms

UNIT 3 : Arrays

UNIT 4 : Operations on Arrays

UNIT 5 : Recursion

UNIT 6 : Linked List

UNIT 7 : Operations on Linked List

UNIT 1 : INTRODUCTION TO DATA STRUCTURE

UNIT STRUCTURE

1.1 Learning Objectives
1.2 Introduction
1.3 Data and Information
1.4 Data Structure and its Types
1.5 Data Structure Operations
1.6 Concept of Data Types
1.7 Dynamic Memory Allocation
1.8 Abstract Data Types
1.9 Let Us Sum Up
1.10 Answers to Check Your Progress
1.11 Further Reading
1.12 Model Questions

1.1 LEARNING OBJECTIVES

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

- distinguish between data and information
- learn about data structure
- define various types of data structures
- learn about different data structure operations
- describe data types in C
- define abstract data types.

1.2 INTRODUCTION

A **data structure** in Computer Science is a way of storing and organizing data in a computer's memory or even the disk storage so that it can be used efficiently. It is an organization of mathematical and logical concepts of data. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, in both execution time and memory space, as possible. Data structures are implemented by a

programming language by the data types and the references and operations provided by that particular language.

Different kinds of data structures are suited to different kinds of applications and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases. In the design of many types of computer program, the choice of data structures is a primary design consideration. Experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure.

In this unit, we will introduce you to the fundamental concepts of data structure. Here, we shall discuss the data and information and overview of data structure. We will also discuss the types of data structure, data structure operations and basic concept of data types.

1.3 DATA AND INFORMATION

The term ***data*** comes from its singular form ***datum*****,** which means a fact. The data is a fact about people, places or some entities. In computers, data is simply the value assigned to a variable. The term *variable* refers to the name of a memory location that can contain only one data at any point of time. For example, consider the following statements :

Vijay is 16 years old.

Vijay is in the 12^{th} standard.

Vijay got 80% marks in Mathematics.

Let ‘name’, ‘age’, ‘class’, ‘marks’ and ‘subject’ be some defined variables. Now, let us assign a value to each of these variables from the above statements.

Name = Vijay

Class = 12

Age = 16

Marks = 80

Subject = Mathematics

In the above example, the values assigned to the five different variables are called data.

We will now see what is information with respect to computers. Information is defined as a set of processed data that convey the relationship between the data considered. *Processing* means to do some operations or computations on the data of different variables to relate them so that these data, when related, convey some meaning. Thus, *information* is a group of related data conveying some meaning. In the examples above, when the data of the variables 'name' and 'age' are related, we get the following information:

Vijay is 16 years old.

In the same example, when the data of the variables 'name' and 'class' are related we get another information as

Vijay is in the 12th standard.

Further, when we relate the data of the variables, 'name', 'marks', and 'subject', we get more information that Vijay got 80% marks in Mathematics. The following figure shows how information can be drawn from data.

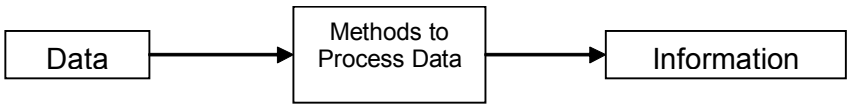


Fig. 1.1 : Relation between data and information

1.4 DATA STRUCTURE AND ITS TYPES

The way in which the various data elements are organized in memory with respect to each other is called a ***data structure***. Data structures are the most convenient way to handle data of different types including *abstract data type* for a known problem. Again, problem solving is an essential part of every scientific discipline. To solve a given problem by using a computer you need to write a program for it. A program consists of two components : *algorithm* and *data structure.*

Many different algorithms can be used to solve the same problem. Similarly, various types of data structures can be used to represent a problem in a computer.

To solve the problem in an efficient manner you need to select a combination of algorithms and data structures that provide maximum

efficiency. Here, efficiency means that the algorithm should work in minimal time and use minimal memory. In addition to improve the efficiency of an algorithm, the use of appropriate data structures also allows you to overcome some other programming challenges, such as :

– simplifying complex problems

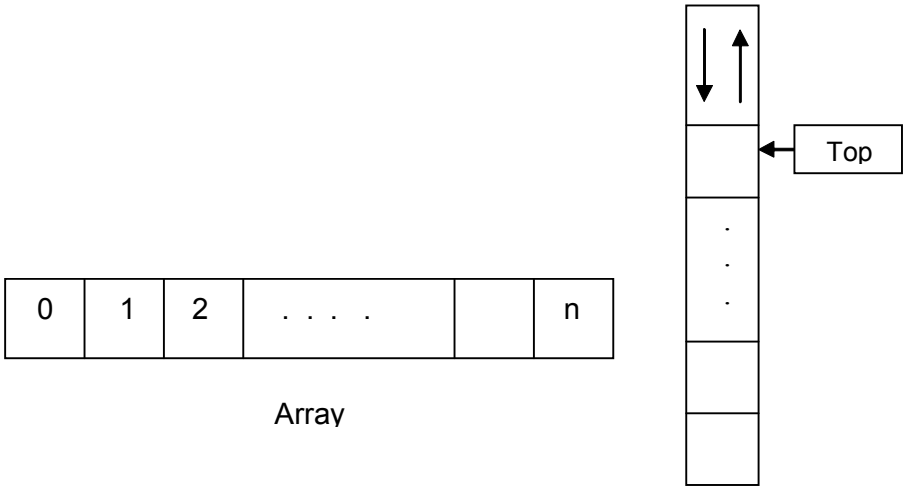
– creating standard, reusable code components

– creating programs that are easy to understand and maintain

Data can be organized in many different ways. Therefore, you can create as many data structures as you want. However, data structures have been classified in several ways. Basically, data structures are of two types : **linear data structure** and **non linear data structure**.

• **Linear data structure :** a data structure is said to be linear if the elements form a sequence i.e., while traversing sequentially, we can reach only one element directly from another. For example : Array, Linked list, Queue etc.

• **Non linear data structure :** elements in a nonlinear data structure do not form a sequence i.e each item or element may be connected with two or more other items or elements in a non-linear arrangement. Moreover, removing one of the links could divide the data structure into two disjointed pieces. For example, Trees and Graphs etc. The following figure shows the linear and nonlinear data structures.



Array

Stack

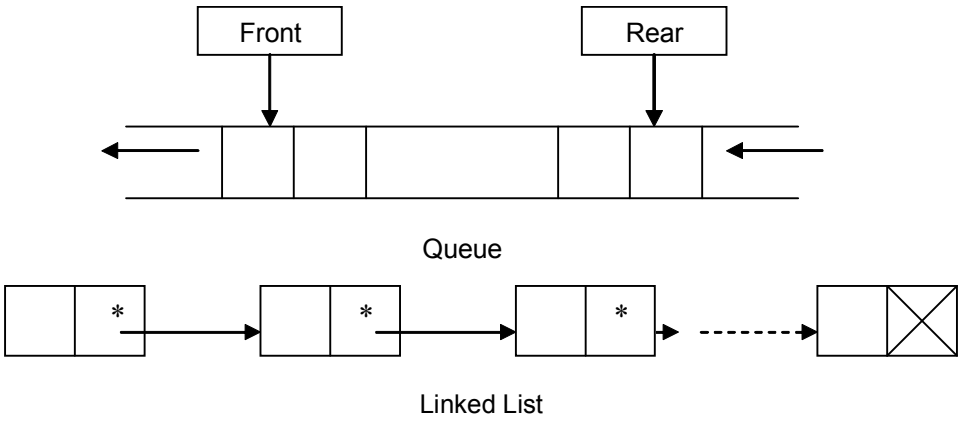


Fig. 1.2 : Linear data structure

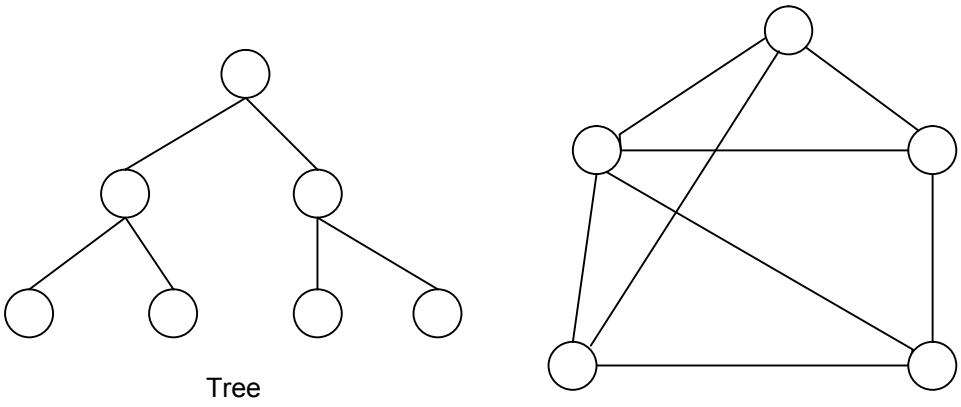


Fig. 1.3 : Non linear data structure

All these data structures are designed to hold a collection of data items.

1.5 DATA STRUCTURE OPERATIONS

We come to know that data structure is used for the storage of data in computer so that data can be used efficiently. The data manipulation within the data structures is performed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. The following four operations play a major role on data structures.

a) *Traversing :* accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called “visiting” the record.)

b) *Searching :* finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.

c) *Inserting :* adding a new record to the structure.

d) *Deleting :* removing a record from the structure.

Sometimes two or more of these operations may be used in a given situation. For example, if we want to delete a record with a given key value, at first we willl have to search for the location of the record and then delete that record.

The following two operations are also used in some special situations :

i) *Sorting :* operation of arranging data in some given order, such as increasing or decreasing, with numerical data, or alphabatically, with character data.

ii) *Merging :* combining the records in two different sorted files into a single sorted file.

1.6 CONCEPT OF DATA TYPES

We have already become familiar with the term '*data type*'. A *data type* is nothing but a term that refers to the type of data values that may be used for processing and computing. The fundamental data types in C are *char, int, float* and *double*. These data types are called built-in data types. There are three categories of data types in C, they are:

a) **Built-in** types that include *char, int, float* and *double*

b) **Derived** data types that include *array* and *pointers*

c) **User defined** types that include *structure, union and enumeration.*

In this section, we will briefly review the data types ***array***, ***pointers*** and ***structures***.

i) Array : An array is a collection of two or more adjacent memory locations containing the same types of data. These data are the array elements. The individual data items can be characters, integers, floating-point numbers, etc. However, they must all be of the same type and the same storage class.

Each array element (i.e., each individual data item) is referred to by specifying the array name followed by one or more ***subscripts,*** with each subscript enclosed in square brackets. The syntax for declaration of an array is :

(Storage Class) datatype arrayname [expression];

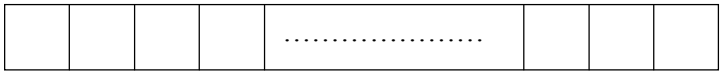
Here, *storage class* may be *auto, static or extern*, which is optional. *Storage class* refers to the permanence of a variable, and its scope within the program, i.e., the portion of the program over which the variable is recognized. If the storage class is not given then the compiler assumes that it is an *auto* storage class.

Some examples of array declaration are :

int x[15]; ***x*** is a 15 element integer array

char name[25]; ***name*** is a 25 element character array

In the array x, the array elements are x[0], x[1], ..., x[14] as illustrated in the fig.



x[0] x[1] x[2] x[3] …….. x[12] x[13] x[14]

Fig. 1.4 : An array data structure

An array is specified by two indices viz, the lower and the upper array index. The smallest or the lower element of an array's index is called its lower bound and in C, it is always zero, and the highest element is called its upper bound. The number of elements in the array is called its range and is given by

upper -lower+1.

Array can be initialize at the time of the declaration of the array. For example,

int marks [5] ={ 85, 79, 60, 87, 70 };

Then, the ***marks*** array can be represented as follows :

85	79	60	87	70

marks [0] marks [1] marks [2] marks [3] marks [4]

Fig. 1.5 : the marks array after initialization

Again for example, a character array can be declared as follows:

char name [20] = { “Krishna” };

K	r	i	s	h	n	a	\0
name [0]	name [1]	name [2]	name [3]	name [4]	name[5]	name[6]	name[7]

Fig. 1.6 : name array after initialization

We know that every character string terminated by a null character (\0). Some more declarations of arrays with initial values are given below :

char **vowels** [ ] = { ‘A’, ‘E’, ‘I’, ‘O’, ‘U’ };

int **age** [ ] = { 16, 21, 19, 5, 25 }

In the above case, the compiler assumes that the array size is equal to the number of elements enclosed in the curly braces. Thus, in the above statements, size of array would automatically be assumed to be 5. If the number of elements in the initializer list is less than the size of the array, the rest of the elements of the array are initialized to zero.

The number of the subscripts determines the dimensionality of the array. For example,

marks [i];

refers to an element in the one dimensional array. Similarly, **matrix [i] [j]** refers to an element in the two dimensional array.

Two dimensional arrays are declared in the same way as one dimensional arrays. For example,

int matrix [3] [5];

is a two dimensional array consisting of 3 rows and 5 column for a total of 15 elements. Two dimensional array can be initialized in the same way like one dimentional array. For example,

```
int matrix [ 3 ] [ 5 ] =   {
                               { 10, 5,  -3, 9, 2 },
                               {  1 , 0, 14, 5, 6 },
                               { -1, 7,  4,  9,  2 }
                           };
```

The matrix array can be represented as follows:

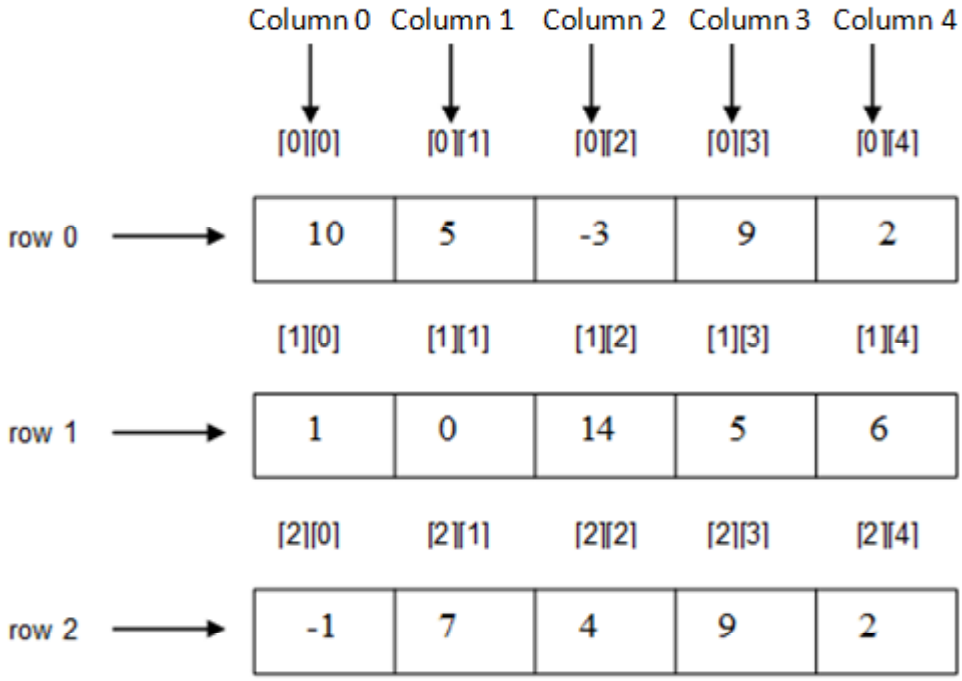


Fig. 1.7 : Matrix array after initialization

The above statement can be written as :

```
int matrix [ 3 ] [ 5 ] = { 10,5,-3,9,2,1,0,14,5,6,-1,7,4,9,2 }
```

A statement such as

```
int matrix [ 3 ] [ 5 ] =   {
                               { 10, 5,  -3 },
                               {  1 , 0, 14 },
                               { -1, 7,   4  }
                           };
```

only initializes the first three elements of each row of the two

dimensional array *matrix*. The remaining values are set to 0.

A Simple Program Using One – dimensional Array : A one dimensional array is used when it is necessary to keep a large number of items in memory and reference all the items in a uniform manner. Let us try to write a program to find average marks obtained by a class of 30 students in a test.

```
#include<stdio.h>
#include<conio.h>
void main( )
    {
        int avg, sum = 0 ;
        int i ; int marks[30] ;          /* array declaration */
        clrscr( );
            for ( i = 0 ; i <= 29 ; i++ )
            {
            printf ( "\nEnter marks " ) ;
            scanf ( "%d", &marks[i] ) ; /* store data in array */
            }
        for ( i = 0 ; i <= 29 ; i++ )
        {
            sum = sum + marks[i] ; /* read data from an array*/
        }
            avg = sum / 30 ;
            printf ( "\nAverage marks = %d", avg ) ;
            getch( );
    }
```

A Simple Program Demonstrating the use of Two Dimensional Array : A transpose of a matrix is obtained by interchanging its rows and columns. If A is a matrix of order m x n then its transpose A^T will be of order n x m. Let us implement this program using two dimensional array as shown below :

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

```
void main()
    {
        int matrix1 [20][20], matrix2 [20][20], i, j, m, n; matrix1, matrix2
are two 2D array
        clrscr();
        printf("Enter the Rows and Column of matrix \n");
        scanf("%d %d", &m, &n);
        printf("Enter the elements of the Matrix \n");
        for( i=1; i<m+1; i++)
            for(j=1; j<n+1; j++)
            scanf("%d", &matrix1 [i][j]);
        for(i=1; i<n+1; i++)
        for(j=1; j<m+1; j++) /* stores the elements in matrix2 */
                matrix2 [i][j] = matrix1 [j][i];
            printf("\n \t Transpose of Matrix \n");
            for(i=1; i<n+1; i++)
            {
                for(j=1; j<m+1; j++)
                printf( "%3d", matrix2 [i][j]);
                printf("\n");
            }
        getch();
    }
```

ii) Pointers : You have already been acquainted with the concept of pointer. At this moment we will recall some of its properties and applications.

A ***pointer*** is a variable that represents the ***location*** (rather than the ***value)*** of a data item, such as a variable or an array element.

Suppose, we define a variable called ***sum*** as follows :

int sum = 25;

Let us define another variable, called ***pt_sum*** like the following way

int *pt_sum;

It means that ***pt_sum*** is a pointer variable pointing to an integer, where * is a unary operator, called the ***indirection operator,*** that

operates only on a pointer variable.

We have already used the '**&**' *unary operator* as a part of a *scanf* statement in our C programs. This operator is known as the ***address operator***, that evaluates the address of its operand.

Now, let us assign the address of sum to the variable **pt_sum** such as

pt_sum = ∑

Now, the variable **pt_sum** is called a ***pointer*** to **sum**, since it "points" to the location or address where **sum** is stored in memory. Remember that ***pt_sum*** represents ***sum's*** address, not its value. Thus, ***pt_sum*** is referred to as a ***pointer variable***.

The relationship between pt_sum and sum is illustrated in Fig.

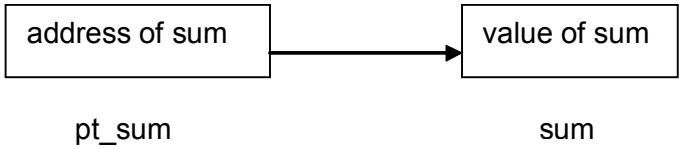


Fig.1.8 : Relationship between pt_sum and sum

The data item represented by **sum** (i.e., the data item stored in sum's memory cells) can be accessed by the expression ****pt_sum***.

Therefore, ****pt_sum*** and **sum** both represent the same data item i.e. 25.

Several typical pointer declarations in C program are shown below

```
int *alpha ;
char *ch ;
float *s ;
```

Here, alpha, ch and s are declared as pointer variables, i.e., variables capable of holding addresses. As addresses (location nos.) are always whole numbers; therefore pointers always contain whole numbers.

The declaration **float *s** means that **s** is going to contain the address of a floating-point value. Similarly, **char *ch** means that **ch** is going to contain the address of a character value.

Let us try to write a program that demonstrates the use of a pointer:

```
#include <stdio.h>
#include<conio.h>
void main( )
    {
        int a = 5;
        int *b;
        b = &a;
        clrscr();
        printf (“value of a = %d\n”, a);
        printf (“value of a = %d\n”, *(&a));
        printf (“value of a = %d\n”, *b);
        printf (“address of a = %u\n”, &a);
        printf (“address of a = %d\n”, b);
        printf (“address of b = %u\n”, &b);
        printf (“value of b = address of a = %u”, b);
        getch();
    }
```

[Suppose address of the variable a = 1024, b = 2048]



OUTPUT : value of a = 5
value of a = 5
value of a = 5
address of a = 1024
address of a = 1024
value of b = address of a = 1024

Pointer to Pointer : A pointer to a pointer is a technique used frequently in more complex programs. To declare a pointer to a pointer, place the variable name after two successive asterisks (*). In this case, one pointer variable holds the address of the other

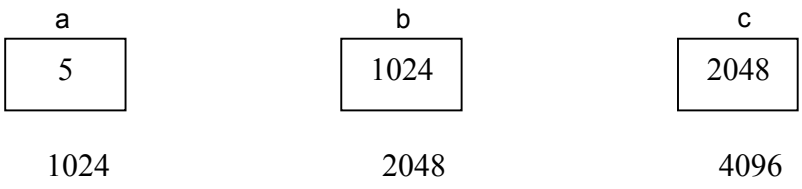
pointer variable. The following declaration is a of pointer to pointer declaration :

int **x;

The following program shows the use of pointer to pointer techniques :

```
#include <stdio.h>
#include<conio.h>
void main( )
{
        int a = 5;
        int *b;
        int **c;
        b = &a;
        c = &b;
        printf (“value of a = %d\n”, a);
        printf (“value of a = %d\n”, *(&a));
        printf (“value of a = %d\n”, *b);
        printf (“value of a = %d\n”, **c);
        printf (“value of b = address of a = %u\n”, b);
        printf (“value of c = address of b = %u\n”, c);
        printf (“address of a = %u\n”, &a);
        printf (“address of a = %u\n”, b);
        printf (“address of a = %u\n”, *c);
        printf (“address of b = %u\n”, &b);
        printf (“address of b = %u\n”, c);
        printf (“address of c = %u\n”, &c);
        getch();
    }
```

[Suppose address of the variable a = 1024, b = 2048]



OUTPUT : value of a = 5
value of a = 5
value of a = 5
value of a = 5
value of b = address of a = 1024
value of c = address of b = 2048
address of a = 1024
address of a = 1024
address of a = 1024
address of b = 2048
address of b = 2048
address of c = 4096



CHECK YOUR PROGRSS

Q.1. Describe the array that is defined in each of the following statements. Indicate what values are assigned to the individual array elements.

a) char flag[5]= { ' T ' , ' R ' , ' U ' , 'E' };

b) int p[2][4] = {
{1, 3, 5, 7 },
{2, 4, 6, 8 }
};

Q.2. Define a one-dimensional, six-element floating-point array called **consts.** Assign the following values to the array elements: 0.005, -0.032, 1e-6, 0.167, -0.3e8, 0.015

Q.3. Explain the meaning of each of the following declarations

a) float a, b;
float *pa, *pb;

b) float a = -0.167;
float *pa = &a;

c) char *d[4] = {"north', 'south", "east", "west"};

iii) **Structure :** Structure is the most important user defined data type in C. A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used to select it from the structure. But in an array, all the data items are of same type. The individual variables in a structure are called *member variables*. A structure is a convenient way of grouping several pieces of related information together.

Here is an example of a student structure :

```
struct student
{
    char name[25];
    char course[20];
    int age;
    int year;
};
```

Declaration of a structure always begins with the key word ***struct*** followed by a structure name (here **student)**. After the opening brace there will be the member of the structure containing different names of the variables and their data types followed by the closing brace and the semicolon.

Graphical representation of the structure student is shown below :

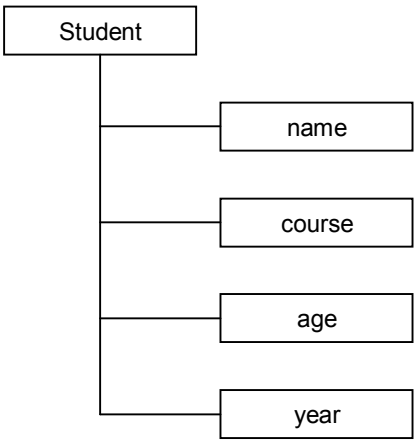


Fig. 1.9 : A structure named student

Now, let us see how to declare a structure variable. In the

following statement we have declared a variable **st_rec** of type student :

student st_rec[100];

In this declaration **st_rec** is a 100 element array of structures.

Hence each element of **st_rec** is a separate structure of type student (i.e., each element of **st_rec** represents an individual student record.)

After the declaration of structure type and the structure variables, now let us see how the elements of the structure can be accessed. In arrays we can access individual elements of an array using a subscript. Structures use a different scheme. They use a dot (.) operator. As an example if we want to access the name of the 10th student (i.e. **st_rec[9]** since the subscript begins with (0) from the above structure then we will have to write

st_rec[9].name

Similarly, course and age of the 10th student can be accessed by writing

st_rec[9].course and **st_rec[9].age**

The members of a structure variable can be assigned initial values in much the same manner as the elements of an array. Example of assigning the values for the 10th student record is shown below :

struct student **st_rec[9]** = { “Arup Deka”, “BCA”, 21, 2008 };

A simple example of using structure data type is shown below:

```
#include<stdio.h>
#include<conio.h>
void main( )
    {
        struct book
        {
            char  name ;
            float  price ;
            int  pages ;
        } ;
```

```
    struct book  b1, b2, b3 ;
    clrscr();
    printf ( "\nEnter names, prices & no. of pages of 3 books\n"
    scanf ( "%c %f %d", &b1.name, &b1.price, &b1.pages ) ;
    scanf ( "%c %f %d", &b2.name, &b2.price, &b2.pages ) ;
    scanf ( "%c %f %d", &b3.name, &b3.price, &b3.pages ) ;
    printf ( "\nAnd this is what you entered" ) ;
    printf ( "\n%c %f %d", b1.name, b1.price, b1.pages ) ;
    printf ( "\n%c %f %d", b2.name, b2.price, b2.pages ) ;
    printf ( "\n%c %f %d", b3.name, b3.price, b3.pages ) ;
    getch();
}
```

Self-referencial structure : When a member of a structure is declared as a pointer that points to the same structure (parent structure), then it is called a *self-referential structure*. It is expressed as shown below :

```
struct node
{
    int data;
    struct node *next;
};
```

where '***node***' is a structure that consists of two members of which one is the data item and the other is a pointer variable holding the memory address of the other node.The pointer variable **next** contains an address of either the location in memory of the successor node or the special value **NULL**.

Self-referential structures are very useful in applications that involves linked data structures such as linked list and trees.

The basic idea of a linked data structure is that each component wiithin the structure includes a pointer indicating where the next component can be found. Therefore, the relative order of the components can easily be changed simply by altering the pointers. In addition, individual components can easily be added or deleted again by altering the pointers.

The diagramatic representation of a node is shown below :

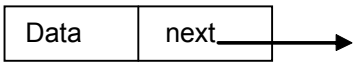


Fig. 1.10 : A node structure

1.7 DYNAMIC MEMORY ALLOCATION

The memory allocation process may be classified as static allocation and dynamic allocation. In static allocation, a fixed size of memory are reserved before loading and execution of a program. If that reserved memory is not sufficient or too large in amount then it may cause failure of the program or wastage of memory space. Therefore, C language provides a technique, in which a program can specify an array size at run time. The process of allocating memory at run time is known as *dynamic memory allocation*.

There are three dynamic momory allocation functions and one memory deallocation (releasing the memory) function. These are **malloc( ), calloc( ), realloc( )** and **free( )**.

malloc( ) : The function *malloc( )* allocates a block of memory. The *malloc( )* function reserves a block of memory of specified size and returns a pointer of type void. The reserved block is not initialize to zero. The syntax for usinig *malloc( )* function is :

ptr = (cast-type *) malloc(byte-size);

where *ptr* is a pointer of type cast-type. The *malloc( )* returns a pointer (of cast-type) to an area of memory with size byte-size.

Suppose x is a one dimensional integer array having 15 elements. It is possible to define *x* as a pointer variable rather than an array. Thus, we write,

int *x;

instead of int x[15]; or #define size 15
int x[size];

When *x* is declared as an array, a memory block having the capacity to store 15 elements will be reserved in advance. But in this case, when *x* is

declared as a pointer variable, it will not be assigned a memory block automatically.

To assign sufficient memory for *x*, we can make use of the library function *malloc*, as follows :

x = (int *) malloc(15 * sizeof (int));

This function reserves a block of memory whose size (in bytes) is equivalent to 15 times the size of an integer. The address of the first byte of the reserved memory block is assigned to the pointer *x* of type ***int***. Pictonial representation is shown below :

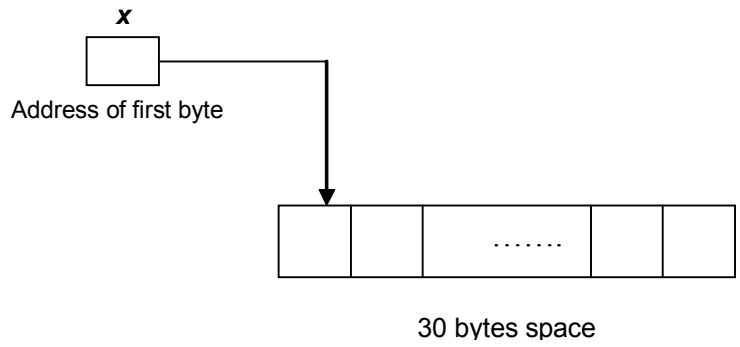


Fig. 1.11 : Representation of dynamic memory allocation

- **calloc( ) :** *calloc* is another memory allocation function that is normally used for requesting memory space at run time for storing derived data types such as arrays and structures. The main difference between the *calloc* and *malloc* function is that *malloc* function allocates a single block of storage space while the calloc function allocates a multiple blocks of storage space having the same size and intialize the allocated bytes to zero. The syntax for usinig *calloc( )* function is :

ptr = (cast-type *) calloc (n, element-size)

where n is the number of contiguous blocks to be allocate each of having the size *element-size*.

- **realloc( ) :** *realloc( )* function is used to change the size of the previouslly allocated memory blocks. If the previously allocated memory is not sufficient or is much larger then by using the realloc function block size can be maximized or minimized. The syntax for usinig *realloc( )* function is :

ptr = realloc (ptr, newsize);

where *newsize* is the size of the memory space to be allocated.

free( ) : It is necessary to free the memory allocated previously so that the memory can be reused. The free( ) function deallocates memory that was previously allocated with malloc( ), calloc( ) or realloc( ). The syntax for usinig *free( )* function is :

free(ptr);

where *ptr* is a pointer to a memory block, which has already been created by *malloc()* or *calloc()*.

1.8 ABSTRACT DATA TYPES

By now, you are well acquainted with data types, like integers, arrays, and so on. To access the data, you have used operations defined in the programming language for the data type. For example, array elements are accessed by using the square bracket notation; or scalar values are accessed simply by using the name of the corresponding variables.

This approach doesn't always work on large and complex programs in the real world. A modification to a program commonly requires a change in one or more of its data structures. It is the programmer's responsibility to create special kind of data types. The programmer needs to define everything related to the new data type such as :

- how the data values are stored,
- the possible operations that can be carried out with the custom data type
- new data type should be free from any confusion and must behave like a built-in type

Such custom data types are called **abstract data types**.

Thus, an abstract data type is a formal specification of the logical properties of a data type such as its values, operations that are to be defined for the data type etc. It hides the detailed implementation of the data type and provides an interface to manipulate them.

Examples of abstract data types are – stacks, queues etc. We will discuss abstract data types in details in the next block.



CHECK YOUR PROGRESS

Q.4. Define a structure named 'complex' having two floating point members *real* and *imaginary*. Also declare a variable x of type complex and assign the initial vaules 3.25 & -2.25.

Q,5. Declare a one dimensional away of 75 elements called *sum* whose elements are stucture of type complex (declared in the above question).

Q.6. Define a self referential structure named *team* with the following three members :

a) a character array of 30 elements called name

b) an integer called age

c) a pointer to another strucutre of this same type, called next.

Q.7. a) The free( ) function is used to

i) release the memory ii) destroy the memory

iii) create a link iv) none of the above

b) The data type defined by the user is known as–

i) abstract data type ii) classic data type

iii) built-in data type iv) all of the above



1.9 LET US SUM UP

- The *data* is a fact about people, places or some entities. In computers, data is simply the value assigned to a variable.
- *Information* is a group of related data conveying some meaning.
- Data structures are of two types : linear data structure and non linear data structure. For example Array, Linked list, Queue etc. are linear

data structure and Trees and Graphs etc are non-linear data structure.

- The possible data structure operations are - traversing, searching, inserting, deleting, sorting and merging.
- An array is a collection of two or more adjacent memory locations containing same types of data.
- A pointer is a memory variable that stores a memory address of another variable. It can have any name that is valid for other variable and it is declared in the same way as any other variable. It is always denoted by '*'.
- A structure is a collection of variables under a single name. These variables can be of different types, and each has a name which is used for selecting it from the structure.
- A strcture which contains a member field that points to the same structure type are called a self-referential structure.
- There is a technique in C language in which a program can specify an array size at run time. The process of allocating memory at run time is known as dynamic memory allocation. There are three dynamic momory allocation functions : **malloc( ), calloc( ), realloc( )** and one memory deallocation function which is **free( )**.
- An abstract data type is a formal specification of the logical properties of a data type such as its values, operations that are to be defined for the data type etc.



1.10 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : a) flag[0] ='T', flag[1] ='R', flag[2] ='U', flag[3] ='E' and flag[4] is assigned to zero.

b) p[0][0] =1, p[0][1] =3, p[0][2] =5, p[0][3] =7, p[1][0] =2, p[1][1] =4, p[1][2] =6, p[1][3] =8

Ans. to Q. No. 2 : float consts[6] =

{ 0.005, -0.032, 1e-6, 0.167, -0.3e8, 0.015 }

Ans. to Q. No. 3 : a) a and b are floating point variables, pa and pb are pointers to floating point quantities (though not necessarily to a & b)

b) a is a floatinig point variable whose initial value is -0.167; pa is a pointer to a floating point quantity, the address of a is assigned to pa as an intial value.

c) d is a one dimensional array of pointers to the string 'north', 'south', 'east' and 'west'.

Ans. to Q. No. 4 :

```
struct complex
    {
        float real;
        float imaginary;
    };
struct complex x = { 3.25, -2.25 }
```

Ans. to Q. No. 5 :

```
struct complex sum[75];
```

Ans. to Q. No. 6 :

```
struct team
    {
        char name[30];
        int age;
        struct team * next;
    };
```

Ans. to Q. No. 7 : a) i.; (b) i.



1.11 FURTHER READING

1. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.
2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
3. Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..



1.12 MODEL QUESTIONS

Q.1. What is information? Explain with examples.

Q.2. What is data? Explain with examples.

Q.3. Name and describe the four basic data types in C.

Q.4. What is a data structure? Why is an array called a data structure?

Q.5. How does a structure differ from an array? How is a structure member accessed?

Q.6. What is a pointer? How is a pointer initialized?

Q.7. What do you mean by dynamic memory allocation? How is it useful?

Q.8. What is the principal difference between the functions *malloc()* and *calloc().*

Q.9. What do you mean by abstract data type? Explain.

Q.10. What is a self-referential structure? For what kind of applications are self referential structures useful?

Q.11. Write short notes on -

i) dynamic memory allocation

ii) abstract data type

iii) pointers

iv) structures

v) self-referential structure.

*****

UNIT 2: ALGORITHMS

UNIT STRUCTURE

2.1 Learning Objectives
2.2 Introduction
2.3 Algorithm
2.4 Complexity
    2.4.1 Space Complexity
    2.4.2 Time Complexity
2.5 Algorithmic Notation
    2.5.1 Big Theta Notation
    2.5.2 Big Oh Notation
    2.5.3 Big Omega Notation
2.6 Let Us Sum Up
2.7 Answers to Check Your Progress
2.8 Further Reading
2.9 Model Questions

2.1 LEARNING OBJECTIVES

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

- define and understand the term algorithm
- describe time and space complexity for algorithm
- define big theta asymptotic notation
- define big oh asymptotic notation
- define big omega asymptotic notation

2.2 INTRODUCTION

In the previous unit (Unit 1), we have learnt about the basics of data structure. We have also learnt about the types of data structure.

In this unit we will learn to define an algorithm and learn the basics of time and space complexity. In addition to these, different types of algorithmic

notations will also be discussed. Moreover, we will be able to compare between different algorithms by using the concepts covered in this unit.

2.3 ALGORITHM

An algorithm is a step by step procedure for solving a problem. Algorithms can also be described as a sequence of computational steps that transform the input into output. There are five basic properties to be fulfilled by an algorithm. They are:

Input: An algorithm should have one or more inputs.

Output: An algorithm should have one or more outputs.

Finite: An algorithm should be a finite one in the sense that it must have a clear stopping point. It must be of fixed length.

Unambiguous: The algorithm must be unambiguous, which means that it should not contain any ambiguous or confusing part.

Easy to understand: The algorithm should be easy to understand so that it should be solvable even without using a computer or any computational tools.

Algorithms can be expressed with the help of both natural languages and programming languages. English is one of the common natural languages that is used to describe an algorithm. Let us now look at some examples to understand how to write an algorithm.

Example 2.1 Algorithm to calculate average and sum of three numbers

Step 1: Input A, B, C

Step 2: Compute Sum (S) as A + B + C

Step 3: Compute Average (AVG) as S/3

Step 4: Display Sum, Average

Example 2.2 Algorithm to find the smaller of two numbers

Step 1: Input A, B

Step 2: If A is less than B

Small = A

```
        Else
                Small = B
Step 3: Display Small
```

Example 2.3 Algorithm to find factorial of a number

```
Step 1: Input A
Step 2: Set Fact = 1, I = 1
Step 3: While I <= A
                F = F X I
                I = I + 1
        End
Step 4: Display Fact
```

An algorithm is written with the objective to solve a problem. For example, let us consider the sorting problem. Suppose we have 15 numbers and we want to sort the numbers in ascending order. Sorting is a common problem and we may be presented with many different solutions to the same sorting problem. Each solution presented can be described using an algorithm. But now, if we have to pick an algorithm from the presented solutions then which solution do we choose and how do we compare the different algorithms to find the best algorithm for solving the problem? We compute the complexity of the different algorithms for the purpose of comparison.

2.4 ALGORITHMIC COMPLEXITY

The measure of an algorithm is made based on its algorithmic complexity. Algorithmic complexity can be of two types – in terms of space and in terms of time. We use data structures to implement an algorithm. Some of the data structure will require more space but less time while others may take less space and more time. We will need to find a trade-off between these two to find the most efficient algorithm. Let us look at the basics of

space and time complexity now.

2.4.1 SPACE COMPLEXITY

Space complexity can be defined as the amount of space an algorithm needs from its start to its completion. The amount of space used by the algorithm from its start to its end is the space complexity of the algorithm. The memory space is measured in terms of bits and bytes. An algorithm with lesser memory space is preferred over the one with more memory space.

To understand what we mean by amount of space taken let us take a simple example. Suppose we need to swap the values of two numbers. The following two programs show the swapping of two nos. in two different ways :

Example 2.4 Swap the values of two numbers

Algorithm A: Swap two numbers using third variable

Step 1: Input A, B
Step 2: Set C = A
Step 3: Set A = B
Step 4: Set B = C
Step 5: Display A, B

Algorithm B: Swap two numbers without using third variable

Step 1: Input A, B
Step 2: Set A = A + B
Step 3: Set B = A – B
Step 4: Set A = A - B
Step 5: Display A, B

Here, we see that both the algorithms have the same number

of steps (5). Still, in the first algorithm, we have used a third additional variable to swap the values between the first two numbers. The space required for algorithm “A” will be more to store the third variable C. In the second algorithm “B”, we have swapped the values of the two numbers without using any third variable. Even though we had to perform additional computation we did not require any additional space in the second algorithm. The space required for the second algorithm “B” will be the space required to store only two variables. Here, the second algorithm “B” requires less space than algorithm “A”.

2.4.2 TIME COMPLEXITY

Time complexity of an algorithm can be defined as the running time of the algorithm. Running time is measured not by computing the actual execution time needed by the algorithm. If this is the case, then the running time would be different for different machines. So, we want a measure that is machine independent, that is, it does not depend on the machine. The running time is measured based on the number of primitive or key operations performed by the algorithm. The number of operations does not change from one machine to another. The number of computational steps performed in the algorithm gives us the running time of that algorithm.

To understand what we mean by primitive or key operations let us take an example. Suppose we need to find the maximum of “N” numbers. Following is an algorithm for this problem.

Example 2.5 Algorithm to find maximum of N numbers

```
Step 1: Input N
Step 2: Read NUM
Step 3: Set MAX = NUM
Step 4: For I = 2 to N Do
                Read NUM
                If NUM > MAX then
                        MAX = NUM
Step 5: Display MAX
```

For the above algorithm, we need to find out the key operations first. The comparison and exchange of elements are the key operations in the above algorithm. Once we know the key operation we need to count the number of times that operation has been executed. For the above case, the comparison instruction is inside for loop which goes from 2 to N so the number of comparisons will be “N-1”.



CHECK YOUR PROGRESS

Q1. What are the properties of an algorithm?

Q2. Define space and time complexity.

Q3. Write an algorithm to find the product of three numbers.

Q4. Write an algorithm to find the greatest of three numbers.

2.5 ASYMPTOTIC NOTATION

The time complexity of an algorithm is measured with the help of asymptotic notations. There are three basic asymptotic notations. They are as follows:

1) Big Omega Ω(f)
2) Big Oh O(f)
3) Big Theta θ(f)

The notations used to describe the asymptotic running time of an algorithm are defined in terms of function whose domains are the set of natural numbers. Let us look at the definitions of these three basic asymptotic notations.

2.5.1 BIG OMEGA NOTATION Ω(f)

Big Omega notation provides an asymptotic lower bound for a function f(n). A function f(n) is said to be Ω(g(n)) if there exists positive values k and c, such that

f(n) >= c* g(n), for all n >= k.

For a function g(n), we denote by Ω(g(n)) the set of functions

(g(n)) = { f(n) : there exist positive constants c and k such that

0 <= cg(n) <= f(n) for all n >= k

For all values n that are at or to the right of k, the value of function f(n) is on or above cg(n).

2.5.2 BIG OH NOTATION O(f)

Big Oh notation provides an asymptotic upper bound for a function f(n). A function f(n) is said to be O(g(n)) if there exist positive values k and c, such that

f(n) <= c* g(n), for all n >= k.

For a function g(n), we denote by O(g(n)) the set of functions

O(g(n)) = { f(n) : there exist positive constants c and k such that

0 <= f(n) <= cg(n) for all n >= k

For all values n that are at or to the right of k, the value of function f(n) is on or below cg(n).

2.5.3 BIG THETA NOTATION

Big Theta function provides both an asymptotic upper and lower bound for a function f(n). A function f(n) is said to be (g(n)) if there exist positive values k, c1 and c2, such that

c1*g(n) <= f(n) <= c2* g(n), for all n >= k.

For a function g(n), we denote by θ(g(n)) the set of functions

θ(g(n)) = { f(n) : there exist positive constants c1, c2 and k such that

0 <= c1*g(n) <= f(n) <= c2* g(n), for all n >= k.

For all values n that are at or to the right of k, the value of function f(n) lies at or above c1g(n) and at or below c2g(n).



CHECK YOUR PROGRESS

Q5. Define omega notation.

Q6. Define big oh notation.

Q7. Define theta notation.



2.6 LET US SUM UP

- An *algorithm* is a step by step procedure for solving a problem. In general terms, an algorithm provides a blueprint to writing a program to solve a particular problem.
- The measure of an algorithm is made based on its algorithmic complexity. Algorithmic complexity can be of two types: in terms of *space* and in terms of *time*.
- *Space complexity* can be defined as the amount of space an algorithm needs from its start to its completion.
- *Time complexity* of an algorithm can be defined as the running time of the algorithm. Running time is measured not by computing the actual execution time needed by the algorithm.
- The time complexity of an algorithm is measured with the help of asymptotic notations. There are three basic asymptotic notations. They are: Big Omega Ω(f), Big Oh O(f) and Big Theta (f)



2.7 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: The properties of an algorithm are:

Input: An algorithm should have one or more inputs.

Output: An algorithm should have one or more outputs.

Finite: An algorithm must have a finite in the sense that it must have a clear stopping point. It must be of fixed length.

Unambiguous: The algorithm must be unambiguous, which means that it should not contain any ambiguous part.

Easy to understand: The algorithm should be easy to understand such that it should be solvable even without using a computer or any computational tools.

Answer to Q2 : Space complexity can be defined as the amount of space an algorithm needs from its start to its completion. The amount of space used by the algorithm from its start to it end is the space complexity of the algorithm.

Answer to Q3 : Algorithm for finding the product of three numbers:

```
Step1:  Input A, B, C
Step 2: Prod = A * B * C
Step 3: Display Prod
```

Answer to Q4 : Algorithm for finding the greatest of three numbers:

```
Step1: Input A, B, C
Step 2: IF A > B do
                IF A > C
                        Set Great = A
                ELSE
                        Set Great = C
                End IF
        Else IF B > C do
                        Set Great = B
                ELSE
                        Set Great = C
        End IF
Step 3: Display Great
```

Answer to Q5 : For a function g(n), we denote by Ω (g(n)) the set of functions

&!(g(n)) = { f(n) : there exist positive constants c and k such that

0 <= cg(n) <= f(n) for all n >= k

Answer to Q6 : For a function g(n), we denote by O(g(n)) the set of functions

O(g(n)) = { f(n) : there exist positive constants c and k such that

0 <= f(n) <= cg(n) for all n >= k

Answer to Q7: For a function g(n), we denote by (g(n)) the set of functions

È(g(n)) = { f(n) : there exist positive constants c1, c2 and k such that 0 <= c1*g(n) <= f(n) <= c2* g(n), for all n>= k.



2.8 FURTHER READING

1. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.
2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
3. Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..

θ



2.9 MODEL QUESTIONS

Q1. Define algorithm?

Q2. What are the properties of an algorithm?

Q3. Write an algorithm to find the factors of a given number.

Q4. Write an algorithm to find whether a given number is prime or not.

Q5. Write an algorithm to display the Fibonacci series till its 10^{th} element.

Q6. Define asymptotic notations.

Q7. Define Big Theta and Big Oh asymptotic notation.

Q8. Define omega notation.

Q9. How the asymptotic notations are used in evaluating algorithms?

****

UNIT 3: ARRAY

UNIT STRUCTURE

3.1 Learning Objectives

3.2 Introduction

3.3 Array

3.4 Array Declaration and Initialization

3.5 One-Dimensional Array (1-D array)

3.5.1 Memory representation of 1-D arrays

3.5.2 Entering data values in 1-D array

3.5.3 Accessing values from 1-D array

3.6 Multi-Dimensional Array

3.6.1 Memory representation of 2-D arrays

3.6.2 Entering data values in 2-D array

3.6.3 Accessing values from 2-D array

3.6.4 3-dimensional arrays

3.7 Application of array

3.8 Let Us Sum Up

3.9 Answers to Check Your Progress

3.10 Further Reading

3.11 Model Questions

3.1 LEARNING OBJECTIVES

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

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

3.2 INTRODUCTION

In the previous unit, we have learnt about algorithms and the time and space complexity related to algorithms.

In this unit we will deal with array and learn to define it. We will also learn to declare and initialize an array and discuss the different types of arrays. The unit will also take into account the basics of one dimensional and multi dimensional arrays.

3.3 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 linear way. By 'homogeneous' we mean that all the elements in the array must belong to the same data type.

Terminologies is associeated with arrays :

Let us look at some of the terminologies associated with array: these are Size, Type, Base and Index. Let us explain them one by one.

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 using index or subscripts, the array is also called an indexed or subscripted variable.

3.4 ARRAY DECLARATION AND INITIALIZATION

Declaration of Array :

An array can be declared just as like we declare any other variable. We give the data type of the variable followed by the name of the variable. In case of an array also we give the data type followed by 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 and 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 ] ;
```

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 have 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

Q1. What is an array?

Q2. Give the syntax for declaring an array.

Q3. Fill in the gap : Arrays can store elements of _____ data types.

Q4. Can we declare an array without mentioning the size of an array?

3.5 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 initialization that we have learnt so far has been for a single dimensional array.

3.5.1 Memory representation of 1-D arrays

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 an array of size 10 in computers memory.

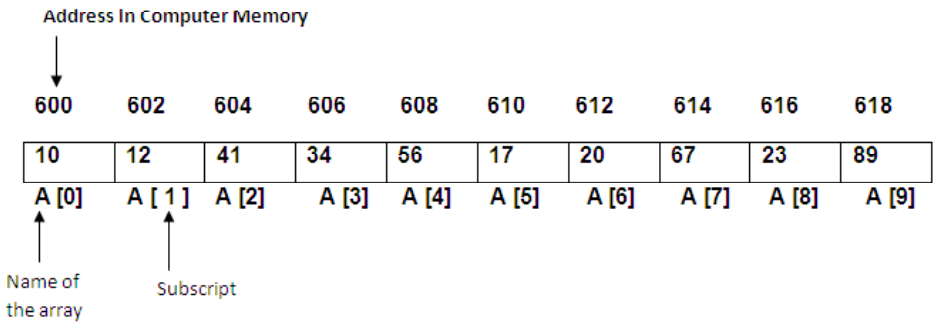


Fig.3.1 Physical representation of array in memory

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 memory is byte-addressable and to store an integer value two bytes of storage are needed 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.

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’ bytes 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 × w

For example if we consider the figure 3.1, then the base address, b

= 600. Now if we consider an integer array that requires byte 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 × 2) = 600 + 18 = 618

Note: Byte addressable: A memory word consists of 2 bytes and each byte can be addressed separately.

3.5.2 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 consider 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.

3.5.3 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 with the procedure of entering the values to it. 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.

Example 3.1 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;
}
```

3.6 MULTI-DIMENSIONAL ARRAY

One dimensional array is the simplest form of arrays. We have so far learnt about one dimensional array. But arrays can have more than one dimension. These are called multi-dimensional arrays since they can have more than one dimension. Multi dimensional arrays can be said to be the arrays of arrays. The most common multi-dimensional arrays are the two-dimensional arrays. Let us discuss the two dimensional arrays in detail now.

Two-Dimensional Arrays (2-D arrays)

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 like we declare a one dimensional array variable. We give the data type of the array variable followed by 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_of_array] [column_size_of_array];

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 computers memory.

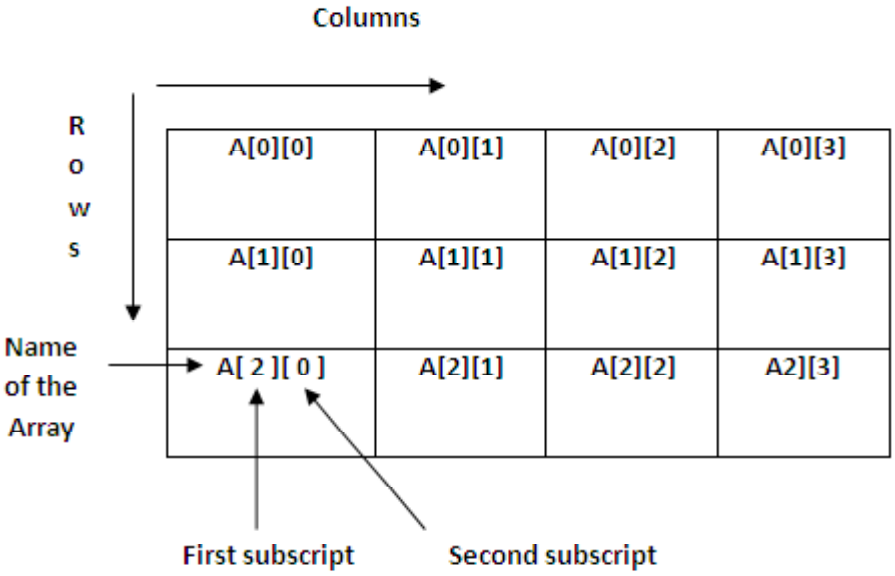


Fig.3.2: Representation of two-dimensional array A of size [3][4]

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.

3.6.1 Memory 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 9 element values { 1, 2, 3 4, 5, 6, 7, 8, 9 } 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 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 9 element values { 1, 2, 3 4, 5, 6, 7, 8, 9 } 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 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

$$B [i] [j] = b + (i \times n + j) \times w$$

where, i represents the row value,

j represents the column value and

n represents the column size.

For column-major, the address of element

$$B [i] [j] = b + (j \times m + i) \times w$$

where, i represents the row value,

j represents the column value and

n represents the column size.

3.6.2 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.

3.6.3 Accessing values from 2-D array

We have learnt about 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 to be stored.

Example 3.2 :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 elemnts 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();
}
```

3.7 3-D ARRAYS

A three dimensional array is an array of arrays. A three dimensional array can be declared just as we declare other array variables. We give the data type of the array variable followed with the name of the array variable and the sizes of the array in square brackets. In case of a three dimensional array we give three sizes for representing the three dimensions of the array.

The first size represents the number of arrays we want of a fixed row and column size. The second size represents the row size and the third size represents the column size of the array.

We can declare a three dimensional array in the following way:

data_type name_of_array [size_of_dim1] [size_of_dim2] [size_of_dim3];

In the above declaration syntax, data type can be any of the valid data types for 1-D or 2D arrays. The data type can be integer, floating point or a character. It is followed by the name of the array variable and the three sizes for the three dimensions of the array within individual square brackets. For example:

int array [5] [5] [5];

char arr [4] [10] [3] ;

float balance [3] [2] [4] ;

Entering values in 3-D arrays

We can enter values in 3-D arrays by using loop control structure like we used in 1-D and 2-D arrays. Let us look at the code where with the help of nested for loop we can easily enter the data values for the array C [2] [3] [4]. This can be implemented as:

```
for ( i = 0; i < 2;  i++ )
{
        for ( j = 0; j < 3; j++ )
        {
                for ( k = 0 ; k < 4; k++ )
                {
                        scanf("%d", &C[ i ] [ j ] [k]);
                }
        }
}
```

In the above code for each iteration of the first for loop, the second for loop traverses through the first to the last row in the array. Again, for

each iteration of the second for loop, the third for loop traverses from the first to the last column in the array. "**C[i][j][k]"** gives the address of the element at the "**j**th" **row** and **"k**th" **column** of the **"i**th**"** array where the current data value needs to be stored.

Accessing 3-D arrays

We can access the values in 3-D arrays by using loop control structure as we have used in 1-D and 2-D arrays. Let us look at the code where, with the help of nested for loop we can easily access the data values for the array C [2] [3] [4]. This can be implemented as:

```
for ( i = 0; i < 2;  i++ )
{
        for ( j = 0; j < 3; j++ )
        {
                for ( k = 0 ; k < 4; k++ )
                {
                        printf("%d", C[ i ] [ j ] [k]);
                }
        }
}
```

In the above code for each iteration of the first for loop, the second for loop traverses through the first to the last row in the array. Again, for each iteration of the second for loop, the third for loop traverses from the first to the last column in the array. "**C[i][j][k]"** gives the address of the element at the "**j**th" **row** and **"k**th" **column** of the **"i**th**"** array from where the current data value needs to be accessed.

Example 3.3: Program to enter and access data elements in a 3-D array

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

void main()
```

```
{
        int C[10][10][10], size_dim1, size_dim2, size_dim3, i, j, k;
        clrscr();

        printf("Give the first dimnsion: ");
        scanf("%d",&size_dim1);
        printf("Give the second dimnsion: ");
        scanf("%d",&size_dim2);
        printf("Give the third dimnsion: ");
        scanf("%d",&size_dim3);

        // Entering elements in 3D array
        printf("\n Enter elemnts in an array C:\n");
        for(i=0;i<size_dim1;i++)
        {
                for(j=0;j<size_dim2;j++)
                {
                        for(k=0;k<size_dim3;k++)
                        {
                                scanf("%d",&C[i][j][k]);
                        }
                }
        }

// Accessing and printing elements in 3D array
printf("\n The elemnts in the array C are:\n");
for(i=0;i<size_dim1;i++)
{
        for(j=0;j<size_dim2;j++)
        {
                for(k=0;k<size_dim3;k++)
                {
```

```
                    printf("\t%d",C[i][j][k]);
                }
                printf("\n");
            }
            printf("\n");
}

getch();

}
```



CHECK YOUR PROGRESS

Q5. The base address of a char array C [15] is 210. What is the memory address of the C [14]?

Q6. Find the memory address of D [4] [8] for an array D [10] [15] when the word size is 4 and base address is 100.

Q7. What are multi-dimensional arrays?



3.8 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 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 × 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:

datatype name_of_array[row_size_of_array] [column_size_of_array];

- 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.
- A three dimensional array is an array of arrays. The first size represents the number of arrays we want of a fixed row and column size. The second size represents the row size and the third size represents the column size of the array.
- We can declare a three dimensional array in the following way:

data_type name_of_array [size_of_dim1] [size_of_dim2] [size_of_dim3];



3.9 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: An array can be defined as a finite, ordered collection of homogeneous elements that are stored in contiguous memory locations.

Answer to Q2: The syntax for declaring an array is:

data_type name_of_array [size_of_array];

Answer to Q3: Arrays cannot store elements of same data types.

Answer to Q4: Yes, arrays can be declared without mentioning the size provided we initialize the array with the value of the elements.

Answer to Q5: 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 × w)

= 210 + (14*1)

= 224

Answer to Q6 : Let us assume that row-major storage representation is 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 X n + j)

= 100 + (4X15 + 8) X 4

= 100 + 68 X 4

= 100 + 272

= 372

Answer to Q7: Multi-dimensional arrays are arrays which have more than one dimension.



3.10 FURTHER READING

1. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.
2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
3. Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..



3.11 MODEL QUESTIONS

Q1. Define array.

Q2. What is row major order and column major order of representation?

Q3. Write a program to input a one dimensional array of 10 elements. Also write a function to print the elements on computer screen.

Q4. How is address translation done in case of 1-D arrays?

Q5. Write a program to input a two dimensional array **A[5][3]**. Also write a function to print the elements on computer screen.

Q6. How is address translation done in case of 2-D arrays?

Q7. What do you mean by multi-dimensional arrays? Can there be arrays of more than three dimensions?

Q8. Why do we need arrays?

*****

UNIT 4: OPERATIONS ON ARRAYS

UNIT STRUCTURE

4.1 Learning Objectives
4.2 Introduction
4.3 Array Traversal
4.4 Searching in Array
4.5 Insertion into an Array
4.6 Deletion from an Array
4.6.1 Problems associated with Insertion and Deletion of elements in an array
4.7 Reversing an Array
4.8 Merging of Array
4.9 Sorting an Array
4.10 Let Us Sum Up
4.11 Answers to Check Your Progress
4.12 Further Reading
4.13 Model Questions

4.1 LEARNING OBJECTIVES

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

- search for an element in an array
- insert an element at any position in an array
- delete an element at any position in an array
- reverse and sort an array
- merge two array's into one

4.2 INTRODUCTION

In the previous unit (Unit 3), we have learnt about the basics of array

and how to declare and define an array. We have learnt to declare and define an array. We have also learnt about one dimensional and multi-dimensional arrays.

The unit will teach how to insert and delete elements at different positions of the array as well as to traverse and search for an element in the array. In addition to these we will learn to merge two different arrays into one array and also to sort and reverse an array.

You must have already come to know how to declare and define an array and to insert elements in it. We will also know how to insert and delete elements in an already existing array in the next few pages.

4.3 ARRAY TRAVERSAL

We have already learnt that array is a collection of homogeneous elements. All the elements in an array belong to the same data type. This data type could be integer, character or a floating point. The elements in an array could be inserted at the time of defining an array.

Array traversal can be defined as the process of accessing each element of an array at least once. Traversing an array means to visit all the elements of an array from the first element to the last element in the array. Traversing an array can be done for a specific purpose like displaying the elements of an array on the computer screen or in searching for a specific element in the array. It can be implemented as:

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

4.4 SEARCHING IN ARRAY

Searching refers to the finding of an item or data from a given list of elements. In case of array searching means finding a specific element from a list of array elements. We compare the **key** element (the element to be

searched) with each element in the array. If we get a match, we display that the element has been found. It can be implemented as:

```
for ( i=0; i < size_array;  i++ )
{
        if( key == array [i] )
        {
                printf("Element %d found at array[%d]", key,i);
        }
}
```

4.5 INSERTION INTO AN ARRAY

Insertion of elements in an array means to add an element at a particular position of an array provided that the array is not full. Insertion of elements in an already exiting array can be done in any of the three ways given below:

1) Insertion at the beginning
2) Insertion at the end
3) Insertion at any given position in the middle

Insertion at the beginning of an array:

To insert an element at the beginning of an array is very simple. First we will need to shift all the elements in the existing array to one place towards the end. When all the elements would have been shifted then the new element can be easily fitted into the starting position. Let us look at the figure below.

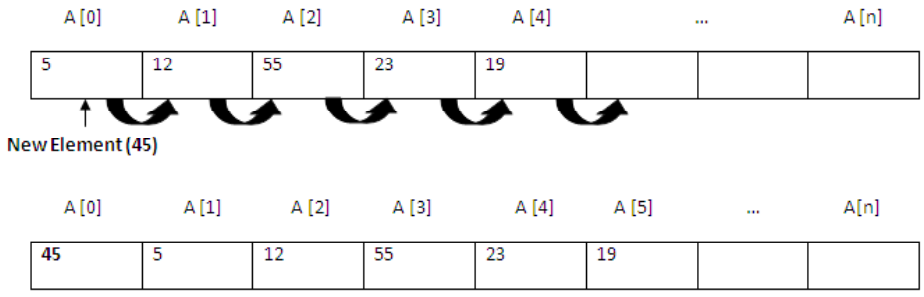


Fig. 4.1: Insertion of element at the beginning

In the above figure, we can see that in order to enter the new element (45) at the starting position of the array A, all the elements of the array have

been pushed to one place towards the end position. Thus the new element can be easily placed in the starting position now. This can be implemented as:

```
n = size_of_array;
for (  i= n-1; i > 0 ; i — )
{
        A [ i ] = A [ i-1 ];
}
A [ 0 ] = new_ele;
```

Insertion at the end of an array

To insert an element at the end of an array is quite easy. We need to go to the next of the last occupied position of the array and insert the new element in that position. Let us first look at the figure below.

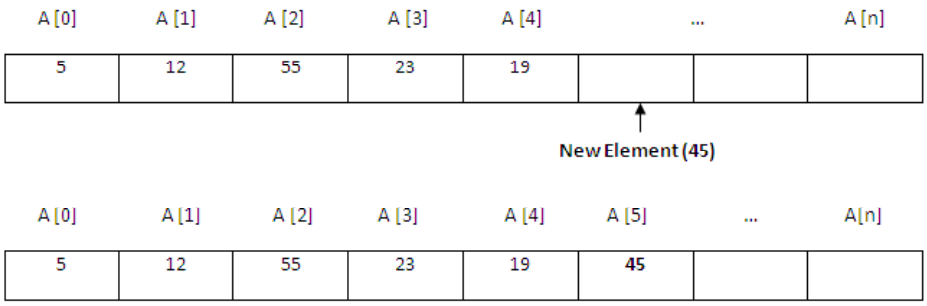


Fig. 4.2: Insertion of element at the end

In the above figure, we see that in order to enter the new element at the end position of the array A, we need to first check if the array is full. If the array is full then we cannot insert the new element at the end without modifying the contents of the last position of the array. On the other hand, if the array is not full then we simply go to the next position of the last occupied position and insert the new element in that position. Let us consider that the array is not full. Then the insertion of an element at the end can be implemented as:

```
n = size_of_array;
occ = occupied_array; // keeps track of last occupied position in array A
if (occ < n)
{
        A [ occ ] = new_ele;
        occ = occ +1;
```

}

Insertion at any given position at the middle of an array

We will now learn to insert an element at any given position in an array. For insertion in any given position at the middle of an array we first will need to shift all the elements after the given location in the existing array to one place towards the end. When all the elements after that location would have been shifted then the new element can be easily fitted into the given position. Let us look at the figure below.

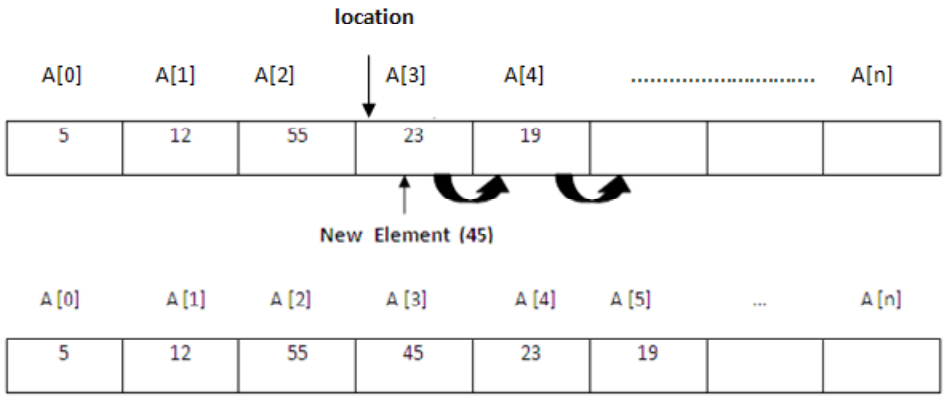


Fig. 4.3: Insertion of element at any given location

In the above figure, we see that in order to enter the new element (45) at the given location or position (A[3]) of the array A, all the elements of the array from that given position have been pushed to one place towards the end position. Thus, the new element can be easily placed in the given position now. This can be implemented as:

```
n = size_of_array;
for (  i= n-1; i > location ; i — )
{
        A [ i ] = A [ i-1 ];
}
A [ location ] = new_ele;
```

4.6 DELETION FROM AN ARRAY

Deletion of elements in an array means to remove an element from a particular position of an array provided that the array is not empty. Deletion of elements in an already existing array like insertion can be done in any of the three ways given below:

1) Deletion at the beginning
2) Deletion at the end
3) Deletion at any given position in the middle

Deletion at the beginning of an array

To delete an element from the beginning of an array is very simple. To delete the starting element from the array we need to shift all the elements in the existing array to one place towards the beginning except for the element at the starting position. Let us look at the figure below.

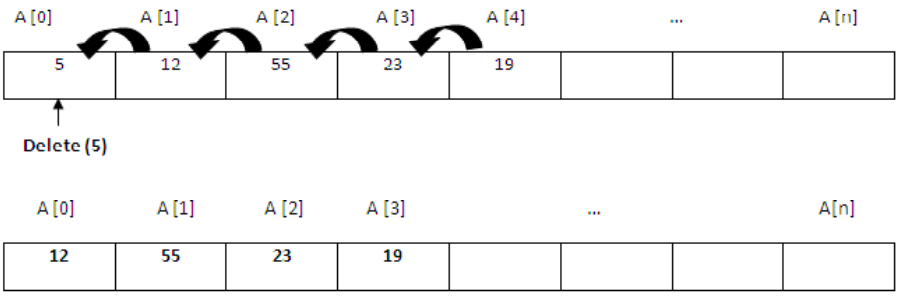


Fig. 4.4: Deletion of element at the beginning

In the above figure, we see that in order to delete the element (5) at the starting position of the array A, all the elements of the array have been pushed to one place towards the beginning position. Thus, the array now no longer contains the first element. This can be implemented as:

```
occ = occupied_array;
for (  i= 1; i <= occ ; i ++ )
{
        A [ i-1 ] = A [ i ];
}
```

occ = occ - 1; // the array contains one less element now so the occ

// variable has been decremented by 1

Deletion at the end of an array

To delete an element from the end position of an array we simply need to go decrement the value of the occupied variable by one. This will indicate that the array does not contain the last element at the occupied position any more. This can be better understood by looking at the figure below.

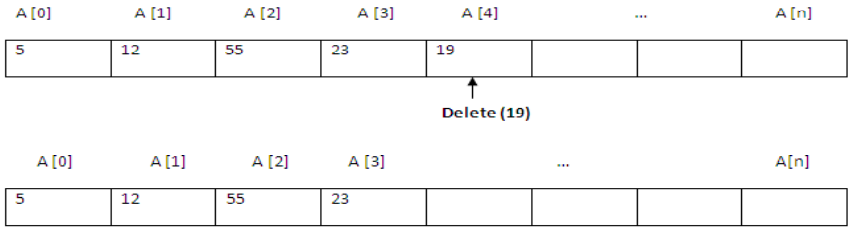


Fig. 4.5: Deletion of element at the end

In the above figure, we see that in order to delete the element at the end position of the array A, we decrement the value of the occupied position variable by one. This means the array now no longer contains the last element. So we simply modify the variable that contains the number of elements of the array. This can be implemented as:

occ = occupied_array; // contains number of elements in array

occ = occ -1;

Deletion at any given position in the middle of an array

To delete an element from any given position of an array we will need to shift all the elements from the given position in the existing array to one place towards the beginning. Let us look at the figure below.

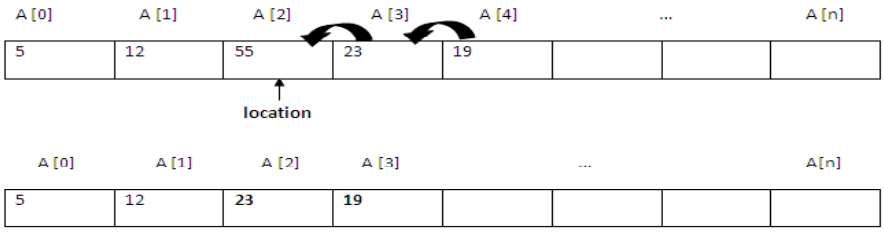


Fig. 4.6: Deletion of element at any given location

In the above figure, we see that in order to delete the element (55) at the given position (A[2]) of the array A, all the elements of the array from next to that location have been pushed to one place towards the beginning position. Also we need to decrement the number of occupied position by one. This can be implemented as:

```
occ = occupied_array;
for (  i=location ; i < occ ; i ++ )
{
        A [ i ] = A [ i+1 ];
}
occ = occ - 1;
```

4.6.1 Problems associated with Insertion and Deletion of elements in Array

We have now learnt how to insert and delete elements at various positions in an array. When we insert an element at the beginning of an array we need to shift the rest of the elements in the array to one place towards the end. Again, when we delete an element from the beginning of an array we need to shift all the rest of the elements to one step towards the beginning. Shifting elements left or right for insertion and deletion of one element needs additional computation time. Similarly, inserting and deleting from any position in the middle also requires additional computation time as it also involves shifting of the elements left or right from any given location.

Some other data structure may provide a better way to add or delete elements to and from the existing list. We will look at those data structures in the next few chapters.

Example 4.1 :Program for insert and delete operation in an array

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

```
 int A[10], n, i, item, loc, new_ele, ch,occ=0;
clrscr();
printf("Give the number of elements you want to enter in the array first: ");
scanf("%d",&n);
printf("\n Enter %d elements in an array\n",n);
for(i=0;i<n;i++)
{
        scanf("%d",&A[i]);
    occ = occ+1;   // keeps track of number of elements in array
}
do
{
printf("\nEnter 1 for insertion in the beginning\n");
printf("Enter 2 for insertion in the end\n");
printf("Enter 3 for insertion in the middle\n");
printf("Enter 4 for deletion in the beginning\n");
printf("Enter 5 for deletion in the end\n");
printf("Enter 6 for deletion in the middle\n");
printf("Enter 7 to display the array\n");
printf("Enter 8 to search a element\n");
printf("Enter 9 to exit \n");
printf("Enter  your choice: ");
scanf("%d", &ch);
switch(ch)
{
case 1: printf("\nEnter new element :");
        scanf("%d",&new_ele);
        for(i=occ; i>0; i—)
            A[i] = A[i-1];
```

```
            A[0] = new_ele;
            occ = occ+1;
            break;

case 2: printf("\nEnter new element :");
            scanf("%d",&new_ele);
            A[occ] = new_ele;
            occ = occ+1;
            break;

case 3: printf("\nEnter new element :");
            scanf("%d",&new_ele);
            printf("\nEnter location :");
            scanf("%d",&loc);
            for(i=occ; i>loc-1; i—)
                        A[i] = A[i-1];
            A[loc-1] = new_ele;
            occ = occ+1;
            break;

case 4: for(i=0;i<occ-1;i++)
            A[i] = A[i+1];
            occ = occ-1;
            break;

case 5: occ = occ -1;
            break;

case 6: printf("\nEnter the location at where element is to be deleted: ");
            scanf("%d",&loc);
```

```
                for(i=loc-1;i<occ-1;i++)
                A[i] = A[i+1];
                occ = occ-1;
                break;

        case 7: printf("The elements in Array A are: \n");
                for(i=0;i<occ;i++)
                printf("A[%d] = %d\n", i,A[i]);
                break;

        case 8:printf("\nEnter element to be serached: ");
                scanf("%d", &item);
                for(i=0; i< occ; i++)
                {
                        if(item == A[i])
                        {
                        printf("\nElement %d found at A[%d]\n", A[i], i);
                        break;
                        }
                }
                if(i==occ)
                printf("\nElement %d not found\n",item);
                break;

        case 9: exit(1);
                break;
        }
} while(ch !=9);
getch();
}
```



CHECK YOUR PROGRESS

Q1. What is an array?

Q2. What are the different operations that can be done on array?

Q3. Fill in the gap. To delete the element at the starting position of an array we need to move the other array elements _____.

Q4. What is the basic difference between searching and traversing an array?

4.7 REVERSING AN ARRAY

To reverse an array means to turn around the array. This means that the first element is now the last, the second element the second last and so on till the second last element becomes the second element and the last element the first element in the list. This can be implemented as:

```
for  ( i = n-1, j = 0;  i  >=  0;  i—,  j++ )
                    B [ j ] = A [ i ] ;
```

In the above implementation of the for loop, the variable “i” goes from the last element to the first element of array A and the variable “j” goes from the first element of array B to its last element simultaneously. For example, let us consider that array A has 5 elements, then the for loop will run for i=4, j=0 to i=0,j=4. This means

B[0] = A[4]

B[1] = A[3]

B[2] = A[2]

B[3] = A[1]

B[4] = A[0]

Thus, the last element of array A is put into the first element of newly

created array B. Again, second last element of A is put as second element of array B. This goes on till all the elements of array A are put into their respective reversed positions in array B.

4.8 MERGING AN ARRAY

Merging an array means to combine two or more arrays into a single array. This operation is used to concatenate any two arrays into one array. Let us look at the figure below.

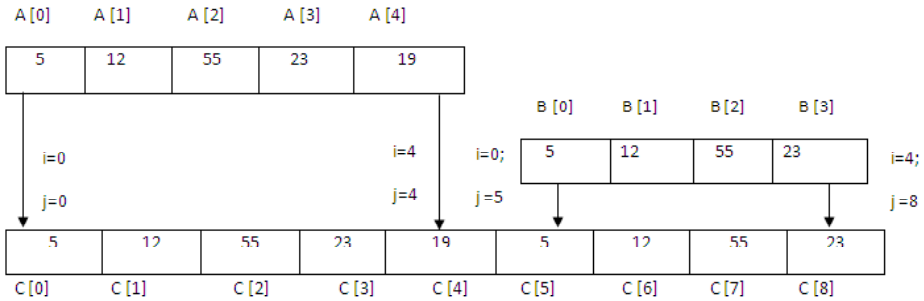


Figure 4.7: Merging of array 'A' and 'B' in array 'C'

In the above figure we can see that we first enter the elements of array A in new array C. Once all the elements of array A are entered into array C, we then start entering the elements of array B into array C after A. This can be implemented as:

```
m= size_of_array_A;
n= size_of_array_B;
j=0;
for ( i = 0; i < m; i++ )
{
        C [ j ] = A [ i ];
        j = j+1;
}
for ( i = 0; i < n; i++ )
{
        C [ j ] = B [ i ];
```

```
            j = j+1;
        }
```

4.9 SORTING AN ARRAY

Sorting means to arrange in a given order. Sorting an array is done to sort the elements of an array in either ascending or descending order. This can be implemented as:

```
for(i=0;i<size_a;i++)
        B[i]=A[i]; // putting elements of array A in array B

// sorting starts
for(i=0;i<size_a-1;i++)
{
        for(j=i+1; j< size_a; j++)
        {
                if(B[i]>B[j])
                {
                        temp = B[i];
                        B[i] = B[j];
                        B[j] = temp;
                }
        }
}
```

In the above implementation, the contents of array A are first transferred into array B. Then we use a nested for loop where the first for loop goes from the first element in B to the second last element in B. For each iteration of the first for loop the second for loop goes from the "i+1"th the last element in the array B. In the first instance of the loop, the value of "i" is 0 that means the second for loop goes from the second element to the last element in the loop and compares the two elements together at a time. The smaller element is shifted to the left at every comparison. First iteration

of the first for loop puts the element with the minimum value at the first position in array B. Second iteration puts the second minimum value at the second position in array B and so on till the whole array is sorted.

Example 4.2 :Program to perform reverse, merge and sorting operations on array

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

void reverse(int A[],int B[], int size_a);
void merge(int A[], int B[], int size_a, int size_b);
void sort(int A[],int size_a);



void main()
{
 int A[15], B[15], size_a, size_b, i, j, ch;
clrscr();

printf("Give the number of elements you want to enter in the array first: ");
scanf("%d",&size_a);
printf("\n Enter %d elemnts in an array\n",size_a);
for(i=0;i<size_a;i++)
{
        scanf("%d",&A[i]);
}

do
{
printf("\nEnter 1 for reversing the array\n");
printf("Enter 2 for merging two array's\n");
```

```
printf("Enter 3 sorting the array in ascending order\n");
printf("Enter 4 to display the array\n");
printf("Enter 5 to exit \n");
printf("Enter  your choice: ");
scanf("%d", &ch);
switch(ch)
{
        case 1: printf("\nYou have chosen to reverse the given array A\n");
                reverse(A,B,size_a);
                break;

        case 2: printf("\nYou have chosen to merge two array's\n");
                printf("Enter the no of elements in second array B");
                scanf("%d",&size_b);
                printf("\nEnter %d elements for array B:",size_b);
                for(j=0;j<size_b;j++)
                        scanf("%d",&B[j]);
                merge(A,B,size_a,size_b);
                break;

        case 3: printf("\nYou have chosen to sort array A\n");
                sort(A,size_a);
                break;

        case 4: printf("The elements in Array A are: \n");
                for(i=0;i<size_a;i++)
                printf("A[%d] = %d\n", i,A[i]);
                break;

        case 5: exit(1);
                break;
        }
```

```
} while(ch !=5);

getch();
}

void reverse(int A[],int B[], int size_a)
{
        int i,j;
        for(i=0, j=size_a-1;i<size_a;i++,j—)
        {
                B[j] = A[i];
        }

        printf(“\nThe original array is :\n”);
        for(i=0;i<size_a;i++)
                printf(“%d\t”,A[i]);

        printf(“\nThe reverse array is :\n”);
        for(i=0;i<size_a;i++)
                printf(“%d\t”,B[i]);
}

void merge(int A[], int B[], int size_a, int size_b)
{
        int C[30],i,j=0;
        for(i=0;i<size_a;i++)
        {
                C[j]      = A[i];
                j++;
        }

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

```
        {
                C[j]    = B[i];
                j++;
        }

        printf("\nThe merged array is:");
        for(i=0; i<j; i++)
        {
                printf("\nC[%d] = %d",i,C[i]);
        }
}

void sort(int A[ ],int size_a)
{
        int B[15],i,j,temp;

        for(i=0;i<size_a;i++)
                B[i]=A[i]; // putting elements of array A in array B

        // sorting starts
        for(i=0;i<size_a-1;i++)
        {
                for(j=i+1; j< size_a; j++)
                {
                        if(B[i]>B[j])
                        {
                                temp = B[i];
                                B[i] = B[j];
                                B[j] = temp;
                        }
                }
        }
```

```
        printf("\nOriginal Array A\n");
        for(i=0;i<size_a;i++)
                printf("\tA[%d]=%d",i,A[i]);

        printf("\nSorted array :");
        for(i=0;i<size_a;i++)
            printf("\tA[%d]=%d",i,B[i]);
}
```



ACTIVITY

1. Explain two scenarios you observe around your daily life where we can use an array.

2. Also explain another two scenarios where usage of array would not be recommended according to you. Give reasons.



CHECK YOUR PROGRESS

Q5. Define sorting.

Q6. Can two arrays of different sizes but of the same data type be merged into a single array? Also can two arrays of different data types but of the same sizes be merged into one?



4.10 LET US SUM UP

Array Traversal can be defined as the process of accessing each element of an array at least once. Traversing an array means to visit all the elements of an array from the first element to the last element in the array.

Searching refers to the finding of an item or data from a given list of elements. In case of array searching means finding a specific element from a list of array elements.

Insertion of elements in an array means to add an element at a particular position of an array provided that the array is not full.

Insertion of elements in an already existing array can be done in any of the three ways given below:

4) Insertion at the beginning
5) Insertion at the end
6) Insertion at any given location in the middle

Deletion of elements in an array means to remove an element from a particular position of an array provided that the array is not empty.

Deletion of elements in an already existing array like insertion can be done in any of the three ways given below:

4) Deletion at the beginning
5) Deletion at the end
6) Deletion at any given position in the middle

Reversing an array means to turn around the array.

Merging an array means to combine two or more arrays into a single array. This operation is used to concatenate any two arrays into one array.

Sorting means to arrange in a given order. Sorting an array is done to sort the elements of an array in either ascending or descending order.



4.11 ANSWERS TO CHECK YOUR PROGRESS

Answer to Q1: An array is a collection of homogeneous elements.

Answer to Q2: The operations that can be performed on an array are: insertion of element, deletion of element, searching for a given element, Traversing an array, sorting an array, reversing an array and merging two arrays.

Answer to Q3: To delete the element at the starting position of an array we need to move the other array elements to one place towards the beginning of the array.

Answer to Q4: The basic difference between searching and traversing is that in traversing we simply access each element of the array once without doing any particular function but in searching while we access each element of the array we compare each element to some searched item.

Answer to Q5: Sorting means to arrange in a given order. Sorting an array is done to sort the elements of an array in either ascending or descending

order.

Answer to Q6: Yes, two arrays of different sizes but often same data type can be merged into one array. But no, two arrays of same sizes but different data types cannot be merged intone array.



4.12 FURTHER READING

1. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.
2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.



4.13 MODEL QUESTIONS

Q1. Write a program to sort an array of 20 elements in descending order

Q2. Give two advantages and two disadvantages of using array

Q3. Write a program to reverse an array of 10 elements

Q4. Consider an existing array with 6 elements. Write a program to add the following elements in an array : 26 at the end
98 at the beginning
66 at the 3rd position

Q5. Consider an existing array with 8 elements in the following order:
12, 34, 45, 67, 81, 25, 96, 17
Write a program to delete the following elements in an array:
17 at the end
12 at the beginning
81 at the 4th position

Q6. Write a program in C to merge the following two arrays:
A: 99, 77, 55, 44, 22
B: 42, 34, 61, 18, 10, 83

Q7. Write a program to search for the element 99 in an array of 10 elements. Give the output either as “Element found” or “Element not found”

****

UNIT 5: RECURSION

UNIT STRUCTURE

5.1 Learning Objectives
5.2 Introduction
5.3 Function
5.4 Recursive Functions
5.5 Iteration versus Recursion
5.6 Rules for Recursive Function
5.7 Advantages and Disadvantages
5.8 Let Us Sum Up
5.9 Answers to Check Your Progress
5.10 Further Reading
5.11 Model Questions

5.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- learn the concept of recursion
- write recursive function
- compare iterative and recursive function
- define the rules of recursive function
- learn about the advantages and disadvantages of using recursion

5.2 INTRODUCTION

In the previous unit (Unit 4) we have learnt about array data structure and the various operations associated with arrays. We have also been acquainted with functions while studying C Programming language. C allows the user to define their own functions for carrying out various individual tasks. With the proper use of function, a large program can be broken down into a number of smaller, self-contained components, each of which has some unique purpose.

In this unit we will discuss another problem solving approach, called ***recursion***. The most common application of recursion is in *mathematics* and

computer science, where a function being defined is applied within its own definition. Recursion is used to solve various mathematical problems by dividing it into smaller problems. In this technique, the solution to a problem depends on solutions to smaller instances of the same problem.

5.3 FUNCTION

Before discussing recusion, let us begin with the concept of function. A function is a set of program statements that carries out some specific task and 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 task is repeated many times. Instead of writing the task 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 wrong function instead of modifying the whole program.

- If the program is divided into functions, each function can be written by one or two team members of the team rather than having the whole team to work on the whole complex program.

- A single function written in a program can be used in other programs also.

5.4 RECURSIVE FUNCTIONS

In C programming language, when a function calls itself over and over again, that function is called ***recursive function***. The process of function calling itself repeatedly is known as ***recursion***. The recursion continues until some condition is met to prevent it.

The general syntax for recursion is as follows:

```
returntype recursive_function (argument list)
{
        statements;
        ... ... ...
        recursive_function (actual argument); //function calling itself
        ... ... ...
 }
```

Recursion can be pictorially represented with the following flowchart:

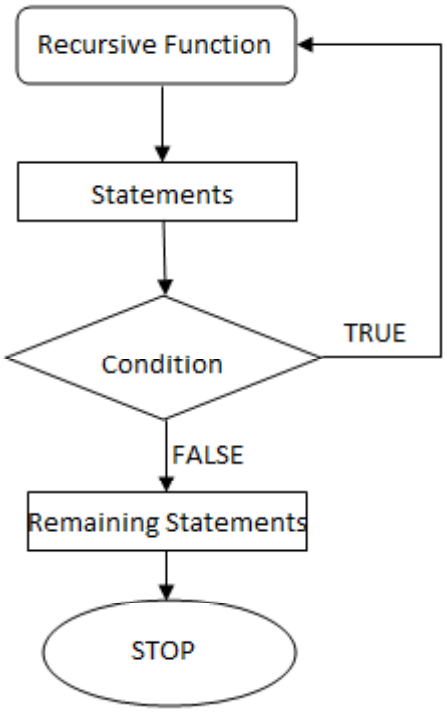


Fig. 5.1: Flowchart for recursion

Let us start with a very basic example of recursion :

//Program 5.1: Program showing infinite recursive function

```
#include<stdio.h>
#include<conio.h>
void func( );   //function prototype
```

```
void main( )
{
      clrscr( );
      func( );      //function call
}
void func( )    //function definition
{
      printf("\n This is a recursive function \n");
      func();       //function calling itself
      return;
}
```

In this code, we have seen that the function **func()**, in its definition, calls itself and thus **func()** becomes a recursive function. When we execute this program, the **main()** function would call **func()** once and then **func()** would continue calling itself forever. The output will be like this:



The statement "**This is a recursive function**" will be displayed continuously many times. At this, we may assume that the program would continue executing forever because recursive function **func()** would continue calling itself forever but it did not happen so. The program crashed.

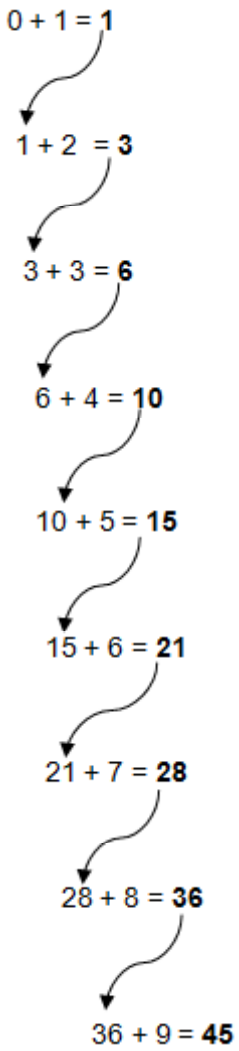
//Program 5.2: Program to show infinite recursive function

```
#include<stdio.h>
#include<conio.h>
void main()
{
   printf("KKHSOU");
   main();
   getch();
}
```

In this program, we are calling **main()** from **main()** which is recursion. But we have not defined any condition for the program to exit. Hence, this code will display "KKHSOU" infinitely in the output screen.

Therefore, in order to prevent infinite recursive call, we need to define proper **exit (terminating) condition** in a recursive function.

Let us discuss a practical example where recursion can make things really easy. Suppose we have given the numbers from 0 to 9 and we need to calculate the sum of these numbers in the following way :



Here, we start with 0 and 1, sum them up and add the result into next number i.e., 2; then again, we add this result to 3 and continue like this. Now we will implement the above addition recursively in C language.

// Program 5.3: Program to find the sum of 0 to 9 recursively

```
#include <stdio.h>
#include<conio.h>
void main()
{
        int sum = 0;
        func(sum); //calling func() from main
}
int count = 1;
void func(int sum)
{
            sum  = sum + count;
            count ++;
             if(count <= 9)
            {
                  func(sum); //function calling itself
            }
            else
            {
                 printf("\nSum  is [%d] \n", sum);
             }
   return;
}
```

The above code works like this:
When **func()** was called through **main()**, 'sum' was initially zero. For every call to **func()**, the value of 'sum' is incremented with 'count' which is 1 initially, which itself gets incremented with every call. The terminating condition of this recursion is when value of 'count' exceeds 9. When 'count' exceeds 9, the value of 'sum' will be the final result of this addition.

Here is one example where recursion can be used to calculate factorial of a given number. In Mathematics, we can define the factorial of a positive integer *n,* written as *n! ,* as the product of all positive integers less than or equal to *n*. Thus we have,

1! = 1
2! = 1 * 2 = 2
3! = 1 * 2 * 3 = 6
4! = 1 * 2 * 3 * 4 = 24
.....
.....

We can also write, for example,

4! = 1 * 2 * 3 * 4 = (1 * 2 * 3) * 4 = 3! * 4

Thus, in general,

n! = (n - 1)! * n

This leads to the following recursive definition:

The factorial of a positive integer n, is

n!= 1, if n = 1 , or n!= (n - 1)! * n , if n > 1.

If we write this definition recursively in C programming, it will look like this:

```
int factorial (int n)
{
        if (n == 1)
            return 1;
        else
            return  factorial (n - 1) * n;
 }
```

Let us discuss *three* methods for finding the factorial of a given number. In the *first* method we use the 'for' loop, in the *second* method we use user-defined function and in the *third* program we use recursion for finding factorial.

// Program 5.4: Finding factorial of a number using loop

```
#include<stdio.h>
#include<conio.h>
void main()
{
   int i,n,fact = 1;
    clrscr();
```

```
    printf("Enter a number to calculate factorial:");
    scanf("%d", &n);

     for(i=1;i<=n;i++)
           fact=fact * i;

    printf("Factorial of %d is %d\n", n, fact);
     getch();
}
```

Now let us calculate factorial using a user-define function.

// Program 5.5: Finding factorial of a number using function

```
#include<stdio.h>
#include<conio.h>
long factorial(int);
void main()
{
  int number;
  long fact;
   clrscr();
  printf("Enter a number to calculate factorial:");
  scanf("%d", &number);
  printf("%d! = %ld\n", number, factorial(number));  //function call
  getch();
}
long factorial(int n)  //function definition
{
   int i;
   long result = 1;
   for (i = 1; i <= n; i++)
         result = result * i;
   return result;
}
```

If we enter the number 6, then the result will be like this:

```
Enter a number to calculate factorial:6
6! = 720
```

//Program 5.6: Finding factorial of a number recursively

```
#include<stdio.h>
#include<conio.h>
long factorial(int);  //function prototype
void main()
{
          int n;
          long f;
         printf("Enter an integer to find factorial:");
         scanf("%d", &n);
         if (n < 0)
             printf("Negative integers are not allowed.");
        else
        {
              f = factorial(n);              //function call
              printf("%d! = %ld\n", n, f);
        }
 getch();
}

long factorial(int n)   //function definition
{
      if (n == 0)
           return 1;
      else
            return(n * factorial(n-1));    //recursive call
}
```

The output of Program 5.6 will give the factorial of different numbers entered through the keyboard. This is shown below:

```
Enter an integer to find factorial:0
0! = 1
Enter an integer to find factorial:1
1! = 1
Enter an integer to find factorial:2
2! = 2
Enter an integer to find factorial:f5
Negative integers are not allowed.Enter an integer to find factorial:5
5! = 120
Enter an integer to find factorial:10
10! = 3628800
```

Here, the number, whose factorial is to be found, is stored in the variable **n**. A recursive function **factorial(n)** calculates the factorial of the number and will store the result in the variable **f** as factorial function calculates the factorial by recursively multiplying **n** with factorial of **(n-1)**. Finally, when **n = 0**, it returns **1** because **0! = 1**.

The following example generates the Fibonacci series for a given number using a recursive function -

//Program 5.7: Fibonacci Series

```
#include<stdio.h>
#include<conio.h>
int fibonacci(int n)
{
    if(n == 0)
   {
        return 0;
    }
    if(n == 1)
    {
       return 1;
    }
    return fibonacci(n-1) + fibonacci(n-2);
}
void main()
{
  int n;
   clrscr();
  for(n = 0; n < 10; n++)
  {
    printf("%d\t\n", fibonacci(n));
  }
 // return 0;
 getch();
}
```

When the above code is compiled and executed, it produces the following result -

0 1 1 2 3 5 8 13 21 34

Program 5.8: Sum of natural numbers recursively

```
#include <stdio.h>
#include<conio.h>
int sum(int n);
void main()
{
     int num, result;
      clrscr();
     printf("Enter a positive integer: ");
      scanf("%d", &num);
      result = sum(num);   //sum() is called
      printf("sum=%d", result);
     getch();
}

int sum(int n)
{
   if (n!=0)
      return n + sum(n-1);     //sum() function calling itself
   else
      return n;
}
```

Initially, the sum() is called from the main() function with number passed as an argument.

Suppose, the value of **n** is **5** initially. During next function call, **4** is passed to the **sum()** function. In next function call, **3** is passed to the function and this process continues until **n** is equal to **0**.

When **n** is equal to **0**, there is no recursive call and the sum of integers is returned to the **main()** function. The output of **Program 5.8** will be 15 if we enter 5 through keyboard.

5.5 ITERATION VERSUS RECURSION

Generally we can say that recursion and iteration perform the same kinds of tasks. These two approaches solve a complicated task one piece at a time and combine the results. We can compare these two techniques in the following manner:

i) Recursion is not suitable for simple programs. Iterative approach is more efficient in terms of memory utilization and speed of execution.

For complex problem, it is always better to use recursion as it reduces the complexity and keeps code readable as compared to iteration. There are some problems which can be efficiently solved using recursion, such as

- Travesals in case of Tree, Graph data structure.
- Sorting algorithms (Merge Sort, Quicksort etc.)

ii) Iteration means repetition of the process until the condition fails. For example, when we use loops in our program, if the specified condition becomes false, the iteration will terminate.

On the other hand, in recursion, function call itself until the terminating condition is reached.

iii) In iterative approach, codes become longer where recursion keeps our code short and simple

iv) Iterative approach involves four steps: initialization , condition, execution of statements and updation.

In case of recursive function, only terminating condition (also termed as base condition) is specified.

v) Iteration is a faster process. But recursion is slower than iteration due to overhead of maintaining stack.

vi) Iteration takes less memory as compared to recursion. Recursion takes more memory due to overhead of maintaining stack

vii) In recursion, if the terminating condition is not specified, then it creates stack overflow (where our system may run out of memory).

viii) We can not solve all iterative problems using recursion. But any recursive problem can be solved iteratively .

The function for finding the factorial of a number can be written iteratively and recursively in the following ways:

Iterative approach :

```
int factorial (int n)
{
        int fact = 1;
        for (int i = 2; i <= n; i++)
        {
                fact = fact * i;
        }
        return fact;
}
```

Recursive approach :

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

5.6 RULES FOR RECURSIVE FUNCTION

While writing recursive function , we have mainly followed the following two rules:

1. Every recursive function must have atleast one terminating condi -tion to prevent infinite recursion.

2. Every recursive function must make progress toward the base case to prevent infinite recursion.

5.7 ADVANTAGES AND DISADVANTAGES

Advantages of Recursion

- Recursion is useful when the same kind of job has to be continued for a finite number of input or time.
- Recursion makes program elegant and cleaner.
- Recursion reduces the size of the code. It can be used as a substitute for iteration where the iterative solution is very lengthy and complex.

Disadvantages of Recursion

- Recursive programs are generally slower than non recursive programs because it needs to make a function call, so the program must save all its current state and retrieve them again later. This consumes more time and thus recursive programs are slower.
- Recursion requires extra storage space. The recursive calls and automatic variables are stored on the stack. For every recursive call separate memory is allocated to automatic variables with the same name. We will discuss the stack data structure later in this block.
- Recursion uses more processor time.
- In order to prevent infinite recursive call, we need to define proper terminating condition (exit condition) in a recursive function. If the recursive function is called for infinite number of times and memory constrains are not taken care of, then stack overflow may occur. So the system may crash.
- In some cases, recursive functions are difficult to debug and understand the code.



CHECK YOUR PROGRESS

1. Fill in the blanks:

i) Iteration takes _____ memory as compared to recursion

ii) Recursion is _____ than iteration due to overhead of maintaining stack.

iii) _____ approach is more efficient in terms of memory utilization and speed of execution.

iv) Every recursive function must have atleast one _____ to prevent infinite recursion.

v) Program execution will always begin by carrying out the instructions in _____function.



5.8 LET US SUM UP

In this unit we have tried to get you acquainted with the concept of recursion in programming. What we have learned in this unit is summed up as follows:

- **Recursion** is used to solve various mathematical problems by dividing it into smaller problems.
- In programming, the process of calling a function by itself is called *recursion* and the function which calls itself is called *recursive function*.
- While using recursion, programmers need to be careful to define a terminating (or, base or, exit) condition from the function, otherwise it will go into an infinite loop.
- Iteration and recursion each requires a terminating condition. Iteration terminates when the loop-continuation condition fails. Recursion terminates when a terminating (base/exit) condition is satisfied.

- Iteration and recursion can occur infinitely. An infinite loop occurs with iteration if the loop-continuation test never becomes false. Infinite recursion occurs if the recursion step does not reduce the problem in a manner that converges on the base condition.

- Using recursion, the length of the program can be reduced.

- The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
- Recursions use more memory. This is because recursive functions require more memory to hold intermediate states in a stack.

- If the programmer forgets to specify the exit condition in the recursive function, the program will execute out of memory.

- The recursion function is not efficient in execution speed and time.



5.9 ANSWERS TO CHECK YOUR PROGRESS

1. (i) less ii) slower iii) Iterative iv)terminating condition v) main()



5.10 FURTHER READING

1. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.
2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.
3. Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..



5.11 MODEL QUESTIONS

Q1. Define recursion and iteration.

Q2. Define recursive function. What are the advantages of using recursive function?

Q3. Compare recursive and iterative function.

Q4. What are the disadvantages of recursive function?

Q5. What are the disadvantages of iterative function?

*****

UNIT 6 : LINKED LIST

UNIT STRUCTURE

6.1 Learning Objectives

6.2 Introduction

6.3 Linked List

6.4 Types of Linked List

- 6.4.1 Singly Linked List
- 6.4.2 Doubly Linked List
- 6.4.3 Circular Linked List

6.5 Advantages of Linked List Over Array

6.6 Let Us Sum Up

6.7 Answers to Check Your Progress

6.8 Further Reading

6.9 Model Questions

6.1 LEARNING OBJECTIVES

After going through this unit, you will able to :

- describe a linked list
- learn the types of linked list
- describe the advantages of linked list over arrays

6.2 INTRODUCTION

We are already familiar with the array data structure. Array is a linear and homogenous data structure. It is very simple to represent a list of elements by using the array but it has some limitations. Once the size of the array is declared, it cannot be changed during program execution. The unused portion of an array also occupies the memory space. Insertion and deletion operation in an array is time consuming and it needs shifting of elements to rearrange the numbers in order. To

overcome such types of difficulties, we will introduce another data structure called *linked list* in this unit. The different types of linked lists are also discussed in this unit.

6.3 LINKED LIST

In general, a series of linearly arranged finite elements (or numbers) having the same type is called a List. The data elements are called nodes. An array is an example of List. In array, by the index the sequential pattern is maintained. We use the index for accessing and manipulating array elements.

Like the arrays, a linked list is also a linear data structure consisting of nodes where each node has two parts:

i) data part

ii) next part (link part)

The data part can hold data of any type (i.e. int, char, float...etc) and the next part holds the address of the next node. This type of list is called a linked list because it is a list whose order is given by links from one item to the next.

An individual node is shown in the following figure.

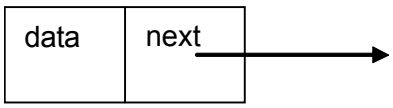


Fig. 6.1 : A node strucutre

A linked list is represented by a pointer (named 'START') to the first node and the *next* part of the last node is assigned the value 'NULL'. Diagrammatic representation of a linked list is shown below :

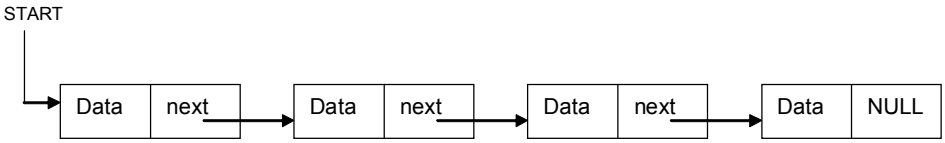


Fig. 6.2 : A linked list

In the above figure, the linked list contains a pointer variable called START, which contains the address of the first node in the list. Each node has two parts, a data part and a next pointer. The data part

contains information which may be an integer, a character, a string etc., and *next* contains the information of pointers which holds the address of the next node. If the *next* has the value NULL, then it is the last node of the list. So, the last node can be identified by the presence of NULL.

A linked list is therefore a collection of structures ordered not by their physical placement in memory (like an array) but by logical links that are stored as part of the data in the structure itself. The link is in the form of a pointer to another structure of the same type. Such a structure is represented as follows :

```
struct node
    {
        int data;
        struct node *next;
    };
```

The first member is an integer data (it can be of any type) and the second member is a pointer to the next node. The next part or link part of a node actually contains memory address, where the data next to the current one is stored. It is also called a pointer to the next data. The structure, which contains a member field that points to the same structure type is called a *self-referential structure*, which we have learned in the first unit.

6.4 TYPES OF LINKED LIST

There are three types of linked list. These are:

- Singly linked list
- Doubly linked list
- Circular linked list

6.4.1 Singly Linked List

Singly linked list is a linked list in which two successive nodes of the list are linked with each other in sequential linear

manner. The following figure depicts a singly linked list.

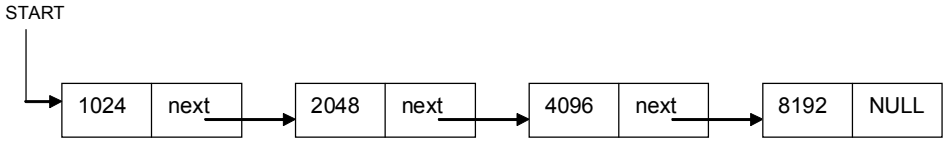


Fig. 6.3 : A singly linked list

In the figure, the first node is pointed by a pointer named 'START'. The NULL in the last node denotes the end of the list. 1024, 2048, 4096, 8192 are the data items of the list.



NOTE

In a list of elements, for any location n, the element at location (n–1) (i.e.left element) is called *predecessor* and the element at position (n+1) i.e. (right element) is called a *successor*.

6.4.2 Doubly Linked List

Doubly linked list is another type of linked list, in which each node has double set of pointers along with the data part. One pointer is pointing to the next node (successor node) and the other pointer points to the preceding node (predecessor node). It means, in a doubly linked list addresses of next as well as preceding elements are linked with current node. This allows us to traverse the list either in forward direction or in backward direction. But in a singly linked list traversal is possible only in one direction. The following figure depicts a doubly linked list.

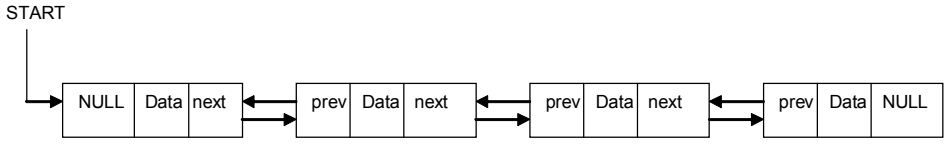


Fig. 6.3 : A doubly linked list

In the figure, each node has two pointers one representing the address of the successor node and the other representing the address of the predecessor node. Here, in the diagram, we have seen that there are two NULL pointers, one for the first node and the other for the last node. When traversing the list from left to right, it identifies the end of the list when *node* → *next* is NULL. Similarly, when traversing the list from right to left, the end of the list is indicated when *node* → *prev* is NULL. The declaration of a doubly linked list is shown below:

```
struct node
{   struct node *prev;
    int data;
    struct node *next;
};
```

6.4.3 Circular Linked List

Circular linked lists are the linked lists which are obtained by linking the last node of the linked list to the first node of the list. So, in a circular linked list, the last node does not contain the NULL pointer; instead it contains the address of the first node. The following figure depicts a circular linked list.

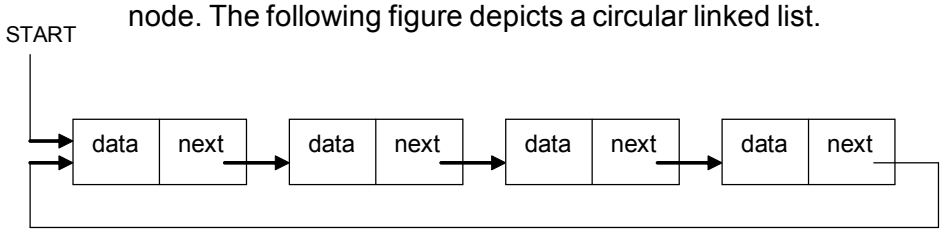


Fig. 6.4 : A circular linked list

One of the shortcomings of a singly linked list is that after reaching the last node in the list, we cannot go back to the previous node unless the address is available. Also, to reach the first node after reaching the end, there will be a need to traverse the whole list backward if pointers(i.e. addresses) are available. But in a circular linked list, each node can be accessed from any node. It means that the circular list can be traversed starting from any node.

6.5 ADVANTAGES OF LINKED LIST OVER ARRAY

Let us discuss the benefits of using a linked list in comparision to using an array.

1) An array, which holds the same type of elements, is static in nature. It means its size is fixed and remains same throughout a program. On the other hand, the linked list uses pointers through which memory can be allocated and freed during

execution of programs. The linked list is more efficient in memory management. The size of the linked list can be expanded or shrunk as per requirements.

2) The memory allocated for an array may be insufficient or may be more than requirement. The user cannot detemine the exact amount of memory required in advance. Due to this, the program execution may not be safe. The pointer has intrinsic flexibility to enlarge or reduce the capacity of storage. We have come to know that, in a node the *next* part of the node points to the successor node in the linked list. That successor node may not be at adjacent memory location or position to that node. It means that the nodes in a linked list are logically contiguous due to linear organization but they may not be physically contiguous in memory. This type of memory allocation is called a linked allocation.

3) Array elements are stored in successive memory locations. This is not the case in the linked list. Insertion and deletion with static list are time consuming and shifting of elements is done to rearrange the numbers in order. In the linked list, insertion and deletion operations are performed very easily. The element can be deleted by de-allocating the memory of that particular element. The insertion can be done by allocationg space at the specified position.



CHECK YOUR PROGRESS

Q.1. Select the appropriate answer from the folowing :

a) The elements of linked list are stored in

i) successive memory locations

ii) alternate memory locations

iii) random memory locations

iv) all of the above

b) The linked list where first and last node cannot be identified is

i) doubly linked list ii) singly linked list

iii) circular linked list iv) all the above

c) The linked list in which the first node have the NULL values is

i) doubly linked list ii) singly linked list

iii) circular linked list iv) all the above

d) The function used for memory allocation in linked list is

i) malloc( ) ii) calloc( )

iii) realloc( ) iv) free( )



6.6 LET US SUM UP

- A linked list is a linear data structure consisting of nodes where each node has a data part and a next part (or link part). The data part can hold data of any type and the next part holds the address of the next node.
- There are three types of linked list.These are: Singly linked list, Doubly linked list, Circular linked list.
- Singly linked list is a linked list in which two successive nodes of the list are linked with each other in sequential linear manner.
- In a doubly linked list each node has two set of pointers along with the data part. One pointer is pointing to the successor node and the other pointer points to the predecessor node.
- Circular linked lists are the linked lists which are obtained by linking the last node of the linked list to the first node of the list.



6.7 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : a) i)successive memory locations

b) iii) Circular linked list

c) i)Doubly linked list

d) i) malloc( )



6.8 FURTHER READING

1. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.

2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.

3. Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..



6.9 MODEL QUESTIONS

Q.1. Why is a linked list called a dynamic data structure?

Q.2. What are the advantges of using a linked list over arrays?

Q.3. Explain types of linked lists. On which basis is the linked list classified?

Q.4. List the advantages of doubly linked list over singly linked list.

Q.5. Compare circular linked list with doubly linked list and list their relative merits and demerits.

*****

UNIT 7: OPERATION ON LINKED LIST

UNIT STRUCTURE

7.1 Learning Objectives
7.2 Introduction
7.3 Traversing a Linked List
7.4 Searching a Linked List
7.5 Insertion into a Linked List
7.6 Deletion from a Linked List
7.7 Reversing a Linked List
7.8 Let Us Sum Up
7.9 Answers to Check Your Progress
7.10 Further Reading
7.11 Model Questions

7.1 LEARNING OBJECTIVES

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

- implement a linked list
- describe traversing a linked list
- search for an element in a linked list
- insert an element at a specified location in a linked list
- delete an element from a linked list
- implement reversing a linked list

7.2 INTRODUCTION

In the previous unit, we have discussed linked list which is a dynamic data structure and is an ideal technique to store data when the user is not aware of the number of elements to be stored. In this unit, we will concentrate on the basic operations that are performed on a linked list. We shall discuss the traversing and searching procedure in a single linked list. In addition,

insertion and deletion techniques will also be discussed along with reversing a linked list.

7.3 TRAVERSING A LINKED LIST

We have already learnt that linked list is a collection of elements called *node*. Each node has two components: the first component contains the *data* (or *information)* of the element and the second component called the *next field* (or *linked field*) which holds the address of the next node. To indicate the end of the list, we generally assign a NULL value. A node in a single linked list has the following structure:

```
struct node
    {
        int data;
        struct node *next;
    };
```

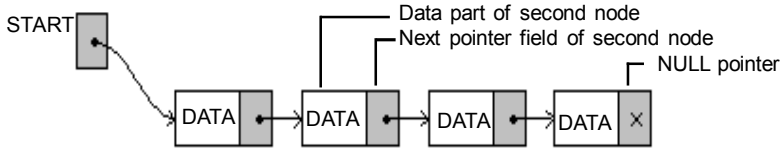


Fig. 7.1 : Linked List with four nodes

Here, member of the structure, struct node *link points to the structure itself.

Traversing a linked list : Traversing a linked list means to visit every node or element in the list starting from the first to the last element. For traversing a linked list we consider ***ptr*** as a pointer variable. The linked list also contains a pointer variable called ***start*** which contains the address of the first node in the list. ***NULL*** pointer denoted by x in the figure indicates the end of the list. Initially we assign the value of ***start*** to ***ptr***. Thus, ***ptr*** will point to the first element of the list. For processing the next element we assign the address of the next element to the ***ptr*** as

```
ptr = ptr -> next;
```

Now, ***ptr*** holds the address of the next node of the list. We can traverse each element of the linked list through this assignment untill ptr has ***NULL*** address (which denotes the last node of the list). Thus, the the traversal of a linked list can be implemented with the following statements:

```
while(ptr!=NULL)
    ptr = ptr -> next;
```

7.4 SEARCHING A LINKED LIST

Searching refers to the finding of an item or data from a given list of elements. So in case of a linked list searching means finding a specific node from a specific list. In searching operation we compare the KEY (item to be searched) value with each node value in the list. It can be implemented as:

```
while(ptr!=NULL)
{
    if(ptr -> data == key)     // Key is the data be searched
        printf("Item found in the list");
    else
        ptr =ptr -> next;
}
```

7.5 INSERTION INTO A LINKED LIST

There may be three possibilities for inserting a node depending upon the position of insertion. The three ways are:

- Insertion at the beginning of the list
- Insertion at the end of the list
- Insertion in between

Insertion at the beginning : Let us consider a pointer ***temp*** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer as :

```
temp->data=d;
```

Again, we know that the start pointer points to the first node of the linked list. Therefore, for inserting at the beginning we will have to assign

start to the next part of temp i.e.,

temp->next=start;

so that next part of the inserted node will point to the node which was originally in first position.

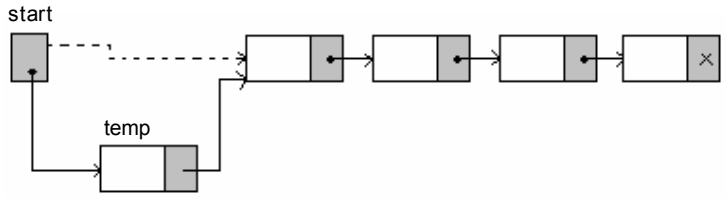


Fig. 7.2 : Insertion of node at the beginning

Now, inserted node points to the next node which was the first node of the linked list. So the inserted node is the first node of the linked list and start pointer is reassigned as

start = temp;

so that start points to the newly inserted node.

Insertion at the end : Let us again consider a pointer ***temp*** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer:

temp->data=d;

We know that to add the new node at the end position we would need to traverse the whole list to reach the last node. Once we traverse the entire linked list and reach the last node the null value of the last node will be replaced by the temp so that it points to the node to be inserted.

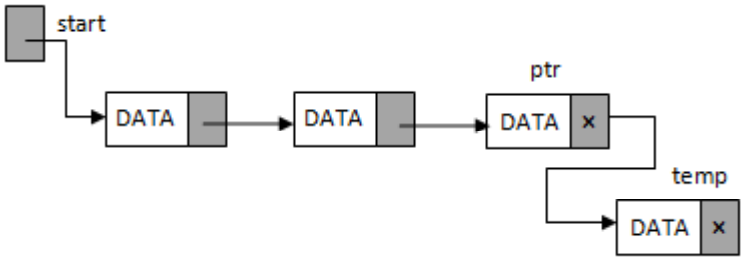


Fig. 7.3 : Insertion of node at the end

```
while   (ptr!=NULL)
ptr = ptr -> next;
```

ptr->next = temp;

Insertion in between : For inserting a node at any position of the linked list we will have to first traverse the linked list to reach the node after which we want to insert the new node. Let us consider a pointer ***temp*** which points to the node that has to be inserted. The data part of the new node is assigned to the data (i.e., d) through the temp pointer as :

temp->data=d;

Again we consider a pointer ***q*** which points to the node after which we have to insert the new node. For inserting the element after the node, we assign the next part of the node to the next part of new inserted node as: temp->next=q->next;

Then the address part of the new inserted node is assigned to the next part of the previous node.

q->next=temp;

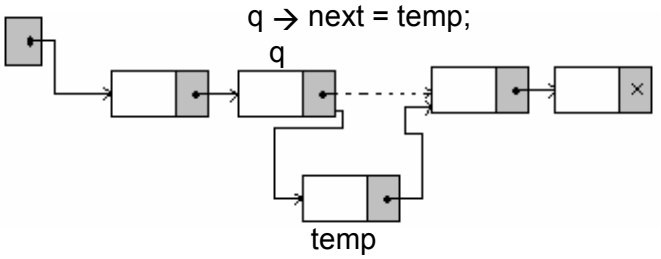


Fig. 7.4 : Insertion of node in between the linked list

7.6 DELETION FROM A LINKED LIST

An element that is to be deleted from a linked list is to be searched first. For this, the traversing operation must be carried out thoroughly on the list. After finding the element there may be three cases for deletion:

- Deletion at beginning
- Deletion at end
- Deletion in between

Deletion at beginning : We know that ***start*** pointer points to the first element of a linked list. If the element to be deleted is the first element of linked list then we assign the value of start to temp as:

temp=start ;

So now temp points to first node which has to be deleted. Now we assign the next part of the deleted node to start as:

start=start->next;

Since start points to the first element of the linked list, so start->next will point to the second element of the linked list. Now, we should free the element to be deleted which is pointed by temp with the following statement:

free(temp);

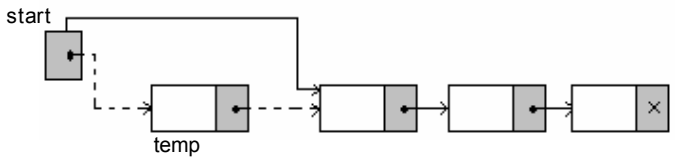


Fig. 7.5 : Deletion of node at the beginning

Deletion at end : If the element to be deleted is the last element of the linked list then we need to make the second last node as the last node of the linked list. We traverse the linked list using a pointer “temp” that will point to the last node and another pointer “prev” that points to the second last node. To make the second last node, the last node of the linked list we assigh a value of NULL to the next part of prev and delete the element pointed by temp using the following statements:

```
temp=start;
while (temp!=NULL)
{
        prev =  temp;
        temp=temp->next;
}
prev->next = NULL;
free(temp);
```

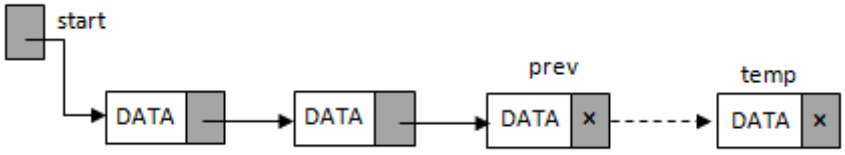


Fig. 7.6 : Deletion of node at the end of the linked list

Deletion in between : If the element is other than the first element of the linked list then we assigh the next part of the deleted node to the next part of the previous node. This can be written as:

```
temp=q->next;
q->next=temp->next;
free(temp);
```

Here, the pointer ***q*** is pointing to the previous node of the node to be deleted. After the first statement ***temp*** will point to the node to be deleted. After second statement, next part of previous node 'q' will point to the next node of the node to be deleted.

If the node to be deleted is the last node of the linked list then the second statement will be as:

```
q->next=NULL;
```

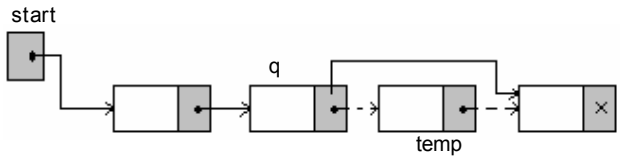


Fig. 7.7 : Deletion of node in between the linked list

The basic operation performed on a single linked list is implemented in the following C program.

Program 7.1 Program for operation on single linked list

```
# include<stdio.h>
# include<conio.h>
# include<malloc.h>
struct node
{
int data;
struct node *next;
};
struct node *start;
struct node* create(int ele);
void display(struct node *);
struct node* add_beg(int ele, struct node* start);
```

```
struct node* add_end(int ele, struct node* start);
struct node* add_mid(int ele, int pos, struct node* start);
struct node* del_beg(struct node* start);
struct node* del_end(struct node* start);
struct node* del_mid(int pos, struct node* start);
void search(int d);

int main()
{
int ch,num,pos;
clrscr();

do
{
printf("Enter \n 1 to create\n 2 to display \n 3 to insert at beginning\n 4 to
insert at end\n 5 to insert after a postion\n 6 to delete at beginning\n 7 to
delete at end \n 8 to delete at position \n 9 to search \n 10 to exit\n");
scanf("%d",&ch);
        switch(ch)
        {
                case 1: printf("Creation\n Enter number: ");
                        scanf("%d",&num);
                        start = create(num);
                        break;
                case 2: printf("Displaying list\n");
                        display(start);
                        break;
                case 3: printf("Insertion at the beginning.\nEnter the
                                                                element:");
                        scanf("%d",&num);
                        start = add_beg(num,start);
                        break;
```

```
                case 4: printf("Insertion at the end.\nEnter the element:");
                        scanf("%d",&num);
                        start = add_end(num,start);
                        break;
                case 5: printf("Insertion at the middle.\nEnter the element:");
                        scanf("%d",&num);
                        printf("Enter the position of the element after which
                        element will be added:\n");
                        scanf("%d",&pos);
                        start = add_mid(num,pos,start);
                        break;
                case 6: printf("Delete at beginning\n");
                        start = del_beg(start);
                        break;
                case 7: printf("Delete at end\n");
                        start=del_end(start);
                        break;
                case 8: printf("Enter the position at which to delete:\n");
                        scanf("%d",&pos);
                        start=del_mid(pos,start);
                        break;
                case 9: printf("Enter the element you want to search:");
                        scanf("%d",&num);
                        search(num);
                        break;
                case 10: exit();
        }
} while(ch != 10);

getch();
return 0;
}
```

```
struct node* create(int ele)
{
struct node *cont;
cont = (struct node *)malloc(sizeof(struct node));

cont->data = ele;
cont->next = NULL;

start= cont;
return start;
}

void display(struct node *temp)
{
temp = start;
if(temp!=NULL)
{
        while(temp->next != NULL)
        {
                printf("%d\n",temp->data);
                temp = temp->next;
        }
        printf("%d\n",temp->data);
        }
else
{
        printf("\nThe linked list is empty\n");
}
}

struct node * add_beg(int ele,struct node *start)
{
```

```
struct node* cont;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
cont->next = start;

start=cont;
return start;
}

struct node * add_end(int ele,struct node *start)
{
struct node* temp;
struct node* cont;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
cont->next = NULL;
temp =  start;

while(temp->next != NULL)
{
        temp = temp->next;
}
temp->next = cont;

return start;
}

struct node * add_mid(int ele,int pos, struct node *start)
{
struct node* temp1;
```

```
struct node* temp2;
struct node* cont;
int count=1;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
temp1 =  start;

while(count < pos)
{
        temp1 = temp1->next;
        count= count+1;
}

temp2=temp1->next;
temp1->next=cont;
cont->next=temp2;

return start;
}

struct node* del_beg(struct node* start)
{
struct node *temp=start;
if(temp->next==NULL)
        {
                start = NULL;
                free(temp);
        }
else
        {
             start =  start->next;
             free(temp);
```

```
        }
        return start;
}

struct node* del_end(struct node* start)
{
struct node *temp=start;
struct node *prev=NULL;

                if(start->next == NULL)
                {
                    start = NULL;
                }
                else
                {
                        while (temp->next!=NULL)
                                {
                                        prev =  temp;
                                        temp=temp->next;
                                }
                prev->next = NULL;
                free(temp);
                }
                return start;
}


struct node* del_mid(int pos, struct node* start)
{
struct node *temp1=start,*temp2,*prev;
int count=1;

if(pos==1)
```

```
{
  start=start->next;
}
else
{
        while (count < pos)
        {
                prev=temp1;
                temp1=temp1->next;
                count = count + 1;
        }

        temp2=temp1->next;
        prev->next = temp2;
        free(temp1);
}
return start;
}

void search(int d)
{
        struct node *ptr = start;
        int pos = 1;
        while(ptr!=NULL)
        {
                if(ptr->data==d)
                {
                        printf("Item %d found at position %d\n",d,pos);
                        return;
                }
                ptr = ptr->next;
                pos++;
        }
```

```
        if(ptr == NULL)
                printf("Item %d not found in list\n",d);
}
```



CHECK YOUR PROGRESS

Q.1. Visiting all the nodes in the linked list is known as

a) Searching b) Traversing

c) Insertion d) none of these

Q.2. A ..... identifies the last logical node in a linked list.

Q.3. Write the functions in C language for the following :

a) Creating a linked list.

b) Inserting a node at the beginning of a list.

7.7 REVERSING A LINKED LIST

Reversing a linked list is to simply change the order of the elements in the linked list. The first element in the linked list becames the last element, the second element becomes the second last element and so on till the last element of the previous linked list becomes the first element of the new linked list.

Program 7.2 Program to reverse a linked list

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

struct node
{
    int data;
    struct node *next;
};

struct node *start;
struct node* create(int ele);
void display(struct node *);
struct node* add_end(int ele, struct node* start);
```

```
struct node *reverse(struct node* start);

void main()
{
    int num;
    clrscr();
    start=NULL;

    printf("Creation\n Enter first elemnt: ");
    scanf("%d",&num);
    start = create(num);

    printf("Insertion at the end.\nEnter the 2nd element:");
    scanf("%d",&num);
    start = add_end(num,start);

    printf("Insertion at the end.\nEnter the 3rd element:");
    scanf("%d",&num);
    start = add_end(num,start);

    printf("Insertion at the end.\nEnter the 4th element:");
    scanf("%d",&num);
    start = add_end(num,start);

    printf("Displaying linked list\n");
    display(start);

    printf("Reversing linked list \n");
    start = reverse(start);
    printf("Displaying reversed list\n");
    display(start);

getch();
}

struct node * create(int d)
{
    struct node *temp;
    temp= malloc(sizeof(struct node));
```

```
    temp->data=d;
    temp->next=NULL;

    if(start==NULL) /*If list is empty */
        start=temp;
    else
    printf("Linked list already created");

    return start;
}

void display(struct node *temp)
{
temp = start;
if(temp!=NULL)
{
    while(temp->next != NULL)
    {
        printf("%d\n",temp->data);
        temp = temp->next;
    }
    printf("%d\n",temp->data);
    }
else
{
    printf("\nThe linked list is empty\n");
}
}


struct node * add_end(int ele,struct node *start)
{
struct node* temp;
struct node* cont;

cont = (struct node *)malloc(sizeof(struct node));
cont->data = ele;
cont->next = NULL;
temp =  start;
```

```
while(temp->next != NULL)
{
     temp = temp->next;
}
temp->next = cont;

return start;
}


struct node* reverse(struct node* root) {
 struct node* new_root = NULL;

 while (root) {
   struct node* temp = root->next;
   root->next = new_root;
   new_root = root;
   root = temp;
 }
 return new_root;
}
```



7.8 LET US SUM UP

- A **linked list** is a dynamic data structure that consists of a sequence of nodes such that in each node there are two fields: one field holds data items and the other field contains a reference (i.e., a *link*) to the next record in the sequence. Linked lists are among the simplest and most common data structures, and are used to implement many important abstract data structures, such as stacks, queues, hash tables etc.
- The following are the primitive operations that can be performed with linked list : traversing, searching, insertion, and deletion
- Traversing is the procedure of passing through (visiting) all the

nodes of the linked list from starting node to the ending node. When any given record is to be searched in the linked list, traversing is applied. When the given element is found, the operation can be terminated or continued for next search.

- Searching is a process in which a given element is compared with all the linked list elements.
- Insertion is the procedure through which we can add new node into a linked list at any position i.e., at the begining of the list or at any position of the list.
- Deletion is a process in which we can delete or remove elements from a linked list i.e. deleting the first node, deleting the last node as well as deleting a particular node from the linked list.
- Reversing a linked list is to simply change the order the order of elements in the linked list.



7.9 ANSWERS TO CHECK YOUR PROGRESS

Ans. to Q. No. 1 : b) Traversing

Ans. to Q. No. 2 : NULL

Ans. to Q. No. 3 : a)

```
create(int d)
{
    struct node *q,*temp;
    temp= malloc(sizeof(struct node));
    temp->data=d;
    temp->next=NULL;
    if(start==NULL) /*If list is empty */
        start=temp;
    else
    {   /*Element inserted at the end */
      q=start;
        while(q->next!=NULL)
            q=q->next;
        q->next=temp;
```

```
        }
    }

b)  add_beg(int d)
    {
        struct node *temp;
        temp=malloc(sizeof(struct node));
        temp->data=d;
        temp->next=start;
        start=temp;
    }/*End of add_beg()*/
```



7.10 FURTHER READING

1. Langsam, Y., Augenstein, M., & Tenenbaum, A. M. (1996). *Data Structures using C and C++* (Vol. 2). New Jersey: Prentice Hall.

2. Kamthane, A. (2009). *Programming with ANSI and Turbo C*. Pearson Education India.

3. Lipschutz, S. (1987). *Schaum's Outline of Data Structure*. McGraw-Hill, Inc..



7.11 MODEL QUESTIONS

Q.1. Write a C program to remove first and last node of the linked list.

Q.2. Write a C program to create and display the elements of a single linked list.

Q.3. Write a C program to insert a node at the first location of a linked list.

Q.4. Describe the insertion procedure in a single linked list.

Q.5. Describe the delete operation in a single linked list.

*****