UNIT- 4 ARRAYS AND POINTERS

UNIT STRUCTURE

4.1 LEARNING OBJECTIVES

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

- learn the concept of arrays in C programming
- learn to define and declare array
- pass array to a function as argument
- learn about character arrays
- learn about different string library functions and their usefulness
- learn about pointer variable and use them in programs
- pass pointer to a function
- learn about dynamic memory allocation and its implementation

4.2 INTRODUCTION

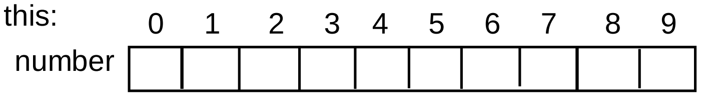
The previous unit gives us the concept of functions in C language. Many applications require the processing of multiple data items that have common characteristics. In such situations it is convenient to place the data items into a linear data structure, where they will all share a common name. The individual data items can be characters, integers, floating-point numbers etc. This unit discusses one of the most important linear data structure called *array*. The unit also introduces pointers and their manipulation. C language uses pointers to represent and manipulate complex data structures. At the end of this unit the concept of dynamic memory allocation is introduced.

4.3 ARRAYS

An array is a series of elements of the same type placed in contiguous memory locations that can be individually referenced by adding an index to a unique identifier. This data structure is a finite and ordered set of homogeneous elements.

Suppose we want to store marks of 100 students. In such a case, we may use two options. One way is to construct 100 variables to store marks of 100 students i.e., each variable containing one student's mark. The other method is to construct just one variable which is capable of holding all hundred students marks. Obviously, this option will be easier and for this we can use array.

Again, let us consider another example to understand array more clearly. Suppose we are to store 10 integer values. For this, we can store 10 values of integer type in an array without having to declare 10 different variables, each one with a different identifier. Instead of that, using an array we can store 10 different values of the same type, ***int*** for example, with a unique identifier. An array to contain 10 integer values of type ***int*** called ***number*** could be represented like this:



where each blank panel represents an element of the array, that in this case are integer values of type ***int***. These elements are numbered from 0 to 9 where 0 indicates the first location. Like a regular variable, an array must be declared before it is used.

4.4 DECLARATION OF ARRAY

An array declaration is very similar to a variable declaration. We can declare an array by specifying its data type, name, and the number of elements the array holds between square brackets immediately following the array name. Here is the syntax:

data_type array_name[size in integer] ;

For example,   int number[5];

In this declaration, 'number' is an interger array of 5 elements which can hold maximum 5 elements. Each array element is referred to by specifying the array name followed by one or more *subscripts*, with each subscript enclosed in square brackets. For the above declaration, the array elements are *number[0], number[1]*, *number[2]*, *number[3]*, *number[4]*.

An array can be initialized at the time of declaration. The general syntax for initializing a one dimensional array at the time of declaration is:

data_type array_name[n] = {element1, element2, ..., element(n-1)};

where, ***n*** is the size of the array and ***element1***, ***element2***,....,***elementn*** are the elements of the array. The total number of elements between braces { } must not be larger than the number of elements that we declare for the array between square brackets[ ]. For example,

```
   int  num[5] = {16,17, 2,3,4};     /* array initialization at the time of
                                        declaration  */
```

In the example, we have declared an array "num", which has 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element.

When an initialization of values is provided for an array, C allows the possibility of leaving the square brackets empty [ ]. In this case, the compiler will assume a size for the array that matches the number of values included between braces { }. For example, we can write the above statement as:

```
int num[  ] = {16,17, 2,3,4};
```

Initilization of two dimensional array during declaration is done by specifying the elements in *row major* order. For example,

```
int number[3][4] = {
                        {1, 3, 5, 7 },
                        {11,13, 17, 19},
                        {23, 27, 29, 31}
                     };
```

The *number* is a two dimensional array of integers with certain initial values. The first subscript can be omitted as shown below:

```
int number[ ][4] = {
                        {1, 3, 5, 7 },
                        {11,13, 17, 19},
                        {23, 27, 29, 31}
                    };
```

The inner braces can also be omitted. This can be written as:

```
int number[ ][4] = {1, 3, 5, 7, 11, 13, 17, 19,  23, 27, 29, 31 };
```

We will discuss two dimensional array again while discussing multi dimensional array.

Row major order:
The elements of the first row in a sequence, followed by those of the second, and so on.

4.5 DEFINING AN ARRAY

Arrays are defined in much the same manner as ordinary variables, except that each array name must be accompanied by a size specification. The simplest form of the array is one dimensional array. For a one dimensional array, the size is specified by a positive integer enclosed in square brackets. For example, int num[100];

Here, ***num*** is an ***one dimensional*** array of size 100 i.e., maximum number of elements in this array will be 100.

In case of a ***two dimensional array***, an element in the array can be accessed using two indices, ***row number*** and ***column number***. Elements can be accessed randomly. For example:

int matrix[20]20];

Here, *matrix* is a two dimensional array with 20 rows and 20 columns. The number of rows or columns is called the range of the dimension.

Representation of one dimensional array in memory is straight forward. Elements from *index 0* to some maximum are stored in some contiguous memory locations. Elements are always stored in **row major** fashion. But in case of two dimensional array there are two methods of representation in memory, which are ***row major*** and ***column major***. In row major representation, the first row of the array occupies the first set of memory locations reserved for the array, the second row occupies the next set and so forth. On the other hand, in column major representation the first column of the array occupies the first set of memory locations reserved for the array, the second column occupies the next set and so forth.

4.6 ACCESSING ARRAY ELEMENTS

We can access an array using its *name* and the *index* of a particular element within square braces. The *array index* indicates the particular element of the array which we want to access. The numbering of elements starts from zero. The smallest index in an array is called the *lower bound* and the highest index is called *upper bound*. In case of C, the lower bound is always 0. If the lower bound is ‘lower’ and the upper bound is ‘upper’, then the number of elements is:

upper- lower + 1

The following statement

```
int num[5]= {22, 24, 26, 28, 30};
```

represents an one dimensional array of 5 integer numbers. Each element of the array can be accessed with the index *num[0]*, *num[1]*, *num[2]*, *num[3]* and *num[4]*. It is assumed that each element of the array occupies two bytes. The general expression for accessing ***one dimensional*** array 'num' is

num[i]

For example, let us consider the following lines of code:

```
int num[5];
printf("\nEnter the numbers into the array:");
for( i = 0; i < 5 ; i++)
{
   scanf("%d", &num[i]);
}
```

Here, the variable ***i*** varies from 0 to 4. The function *scanf()* is called to input the integer values. The address of ***i**th* location is passed to *scanf()* which makes *scanf()* store the integer input into successive locations each time the loop is executed. ***&num[i]*** in the *scanf()* statement refers to the memory location of the integher at the ***i**th* position.

The elements of ***two dimensional*** array can be accessed by the following expression:

marks[i][j];

where *i* and *j* refers to row and column numbers respectively.

4.7 PASSING ARRAY TO FUNCTION

To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within a function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declarations. When declaring a one dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not specified within the formal argument declaration.

Program 1: Program to find the sum of elements of an array where the array is passed as argument to the function.

```
#include<stdio.h>
#include<conio.h>
int  addition (int a, int x[ ] ) ;          //function prototype
void main()
{         int n,i,add;
          int y[20] ;  //array declaration of maximum size 20
          clrscr();
          printf("\nEnter the size of the array:");
          scanf("%d",&n);
          printf("\nEnter the array elements:");
          for(i=0;i<n;i++)
             scanf("%d",&y[i]);
          printf("\nEntered elements are:\n");
          for(i=0;i<n;i++)
              printf("%d\t",y[i]);
          add=addition(n,y) ;   //array size and name is passed
          printf("\nResult is: %d",add);
          getch();
}
int addition (int a, int x[ ])   // function definition , formal argument
{
          int i,sum=0;
          for(i=0;i<a;i++)
               sum=sum+x[i];
          return sum;         //sum is returned to the main function
}
```

EXERCISE

Q. Find the average of *n* numbers using array where *n* is the size of the array.

Q. Write a program to find the summation of 10 even numbers where numbers are entered through the keyboard.

4.8 MULTIDIMENSIONAL ARRAYS

An array with more than one index value is called a ***multidimensional array*** . Multidimensional arrays can be described as "arrays of arrays". The syntax for declaring a multidimensional array isas follows:

data_type array_name[ ][ ][ ];

The number of square brackets specifies the dimension of the array.

Matrix :
A rectangular array of elements (or entries) set out by rows and columns

Initialization of multidimensional arrays:

We have already seen the declation of two dimensional arrays in our previous sections. Like the one dimensional arrays, two dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. For example:

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

The above statement initializes the elements of first row to 1 and second row to 2. The initialization is done row by row. The above statement can be equivalently written as

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

Arrays of three or more dimensions are not used very often because of the memory required to hold them. The computer takes time to generate each index and this can cause the access of multidimensional arrays very slow as compared to a single dimensional array with the same number of elements. Some examples of multidimesional array declation are:

```
int count[3][5][12];
float table[5][4][5][3];
```

Here, ***count*** is a three dimensional array declared to contain 180 integer elements. Similarly, ***table*** is a 4-dimensional array containing 300 elements of floating point type.

Often there is a need to store and manipulate two dimensional data structure such as matrices and tables. In case two dimensional (2D) array, there are two subscripts. One subscript denotes the row and the other denotes the column. We have already used two dimensional array in our previous sections. The declaration of two dimension arrays is as follows:

datatype array_name[row_size][column_size];

For example, int marks[3][4] ;
Here *marks* is declared as a matrix having 3 rows(numbered from 0 to 2) and 4 columns(numbered 0 through 3). The first element of the matrix is marks[0][0] and the last row last column is marks[2][3] .

<u>Elements of two dimensional arrays:</u>

A two dimensional array ***marks[3][5]*** is shown below. The first element is given by ***marks[0][0]*** contains 50 and second element is ***marks [0][1]*** and contains 75 and so on.

marks[0][0] 50	marks[0][1] 75	marks[0][2] 70	marks[0][3] 61
marks[1][0] 51	marks[1][1] 35	marks[1][2] 65	marks[1][3] 78
marks[2][0] 45	marks[2][1] 67	marks[2][2] 28	marks[2][3] 55

To represent a matrix a two dimensional array is required. Suppose there are two matrices A and B having the following elements

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 3 & 8 \\ 6 & 3 & 4 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 3 & 5 \\ 1 & 1 & 1 \\ 1 & 5 & 4 \end{pmatrix}$$

Then the addition matrix will be :

$$C = \begin{pmatrix} 3 & 5 & 8 \\ 6 & 4 & 9 \\ 7 & 8 & 8 \end{pmatrix}$$

One can write the following program for addition of two matrices.

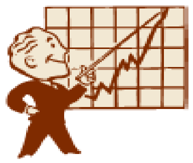
Program 2: Program to add two matrices and store the results in the third matrix.

```
#include<stdio.h>
#include<conio.h>
void main()
{
            int a[10][10],b[10][10],c[10][10],i,j,m,n,p,q;
            clrscr();
            printf("Enter the order of the matrix a\n");
            scanf("%d%d",&p,&q);
            printf("Enter the order of the matrix b\n");
            scanf("%d%d",&m,&n);
            if(m==p && n==q)
            {
                        printf("\nMatrix can be added\n");
            }
            printf("\nEnter the elements of the matrix a:");
            for(i=0;i < m;i++)
                for(j=0;j < n;j++)
                        scanf("%d",&a[i][j]);
            printf("\nEnter the elements of the matrix b:");
            for(i=0;i < p;i++)
               for(j=0;j < q;j++)
                        scanf("%d",&b[i][j]);
            printf("\nThe sum of the matrix a and b is:\n");
            for(i=0;i<m;i++)
            {
                         for(j=0;j<n;j++)
                        {
                                    c[i][j]=a[i][j]+b[i][j];
                                    printf("%d\t",c[i][j]);
                        }
                        printf("\n");
            }
            getch();
}
```

Program 3: Program to find the sum of the diagonal elements of a matrix

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int a[10][10], i, j, n, trace;
  clrscr();
 printf("\nEnter the order of the matrix:");
 scanf("%d", &n);
 printf("\nEnter the elements of the matrix:\n");
 for(i=0;i<n;i++)
    for(j=0;j<n;j++)
           scanf("%d", &a[i][j]);
 trace=0;
 for(i=0;i<n;i++)
   trace= trace+a[i][i];
 printf("\nThe sum of the diagonal elements = %d",trace);
 getch();
}
```

If we execute the program with the order 3 and the lements of the matrix as 1,2 3, 4, 5, 6, 7,8, 9 the the sum of the diagonal elements will be 15.



CHECK YOUR PROGRESS

1. State whether the following statements are true(T) or false(F)
 (i) Arrays are sets of values of the same type which have a single name follwed by an index.
(ii) If N[10] is an array, then N+2 points to the third element of the array.
(iii) The following is a correct array definition char num(30);
(iv) “B” is a string but ‘B’ is a character.
(v) Elements of one dimensionl arrays are always stored in row major fashion.

(vi) Return statement cannot be used to return an array.

2. Fill in the blanks:

(i) An array is a collection of ________ data items.

(ii)The __________________ indicates to the compiler that we are dealing with an array.

(iii) The values used to initialize an array are separated by _____ and surrounded by braces.

(iv) Array elements can be accessed _________.

3. Which of the following declaration is wrong?

(a) int p[10]={1,2,3,4,5}; (b) char ch[ ][3]={{‘a’,’b’,’c’},{‘d’,’e’,’f’}};

(c) int j=0, i=j; (d) All are correct

4. What will be the output of the following program code?

```
#include<stdio.h>
#include<conio.h>
void main() {  int n, result=0, number[ ]={2,5,4,6,1};
          clrscr();
          for ( n=0 ; n<5 ; n++ ) {
                    if(number[n]==2)
                              continue;
                              result= result+ number[n];   }
      printf("%d",result);
      getch(); }
```

5. Find the number of elements of the following array declarations:

(a) int x[2][3]; (b) float p[9];

4.4 STRINGS

A character *(char)* variable can hold a single character. While writing program , sometimes we may have to store a sequence of characters like a person’s name, address etc. We need a way to store these sequence of characters. Although there is no special data type for strings, C handles this type of information with array of characters.

A ***string*** is just an array of characters with the one additional convention that a "*null*" character is stored after the last real character in the array to mark the end of the string. Null character is a character with a numeric value of zero and it is represented by '**\0**' in C. So when we define a string we should be sure to have sufficient space for the null terminator. An array of characters representing a string is defined with the following syntax :

char array_name[size];

In general, each character of a string is stored in one byte, and successive characters of the string are stored in successive bytes. If we want to store the name KKHSOU, then we have to declare an array of *char* of size 6. Although there are five characters in the name but we need an array of six characters. This extra space is to store the "null" character.

String constants :

String constants have double quote marks around them, and can be assigned to char pointers as shown below. Alternatively, we can assign a string constant to a *char* array - either with no size specified, or we can specify a size, but we shouldn't forget to leave a space for the null character.

```
char *text = "Hello";
char text[ ] = "Hello";
char text[6] = "Hello";
```

In the third statement the total numbers of characters in the word "Hello" is 5, but as it is a character array so we have considered the size of array text as 6. i.e., one extra space for null.

Reading and Writing Strings:

One possible way to read in a string is by using ***scanf()***. However, the problem with this, is that if we were to enter a string which contains one or more spaces, scanf() would finish reading when it reaches a space, or if return is pressed. As a result, the string would get cut off. So we could use the ***gets()*** function. A gets takes just one argument - a *char pointer*, or the name of a *char array*, but we have to declare the array or pointer variable first.

A ***puts()*** function is similar to gets() function in the way that it takes one argument - a *char pointer*. This also automatically adds a newline character after printing out the string. Sometimes this can be a disadvantage, so *printf()* could be used instead. The concept of pointers will be covered later in this unit.

4.9.1 Initialization of Strings

C allows to initialize a string at the time of its declaration. Let us consider the following declaration:

char month[ ]={‘A’, ‘p’, ‘r’, ‘i’, ‘l’, ‘\0’};

month is a string which is initialized to *April*. This is a valid statement. But C provides another way to initialize strings which is:

char month[ ]= “April”;

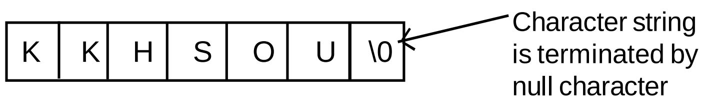
The characters of the string are enclosed within double quotes. The compiler takes care of storing the ASCII (*American Standard Code for Information Interchange*) codes of the characters of the string in memory and also the null terminator in the end.
For example,

```
char name[ ] = {‘K’,’K’,’H’,’S’,’O’,’U’,’\0’};
```

We can also have a simpler choice by giving the following declaration:

```
char name[ ] = “KKHSOU”;
```

Here the string is surrounded by double quotes (“ “). In this method of initialization we do not need to insert the null character ‘\0’ . this will be inserted automatically.



For example let us consider the following two character array definitions. Each includes the assignment of string “KKHSOU” .

```
char university [6] = “KKHSOU” ;    // defined as 6 element array
char university [ ] = “KKHSOU”;      // here size is not specified.
```

The results of these initial assignments are not the same because of the null character "\0", which is automatically added at the end of the second string. Thus the elements of the first array are:

```
university [0] = 'K'
university [1] = 'K'
university [2] = 'H'
university [3] = 'S'
university [4] = 'O'
university [5] = 'U'
```

Whereas the elements of the second array are:

```
university [0] = 'K'
university [1] = 'K'
university [2] = 'H'
university [3] = 'S'
university [4] = 'O'
university [5] = 'U'
university [6] = '\0'
```

The first form is incorrect, since the null character '\0' is not included in the array. So we can define it as:

```
char university [7] = "KKHSOU" ;
```

Program 4: Reading a sting of characters from the keyboard and displaying it.

```
#include<stdio.h>
#include<conio.h>
void main( )
{         char university [7] ;    //declaration of a string of characters
          university[0] = 'K';
          university[1] = 'K';
          university[2] = 'H';
          university[3] = 'S';
          university[4] = 'O';
          university[5] = 'U';
          university[6] = '\0';          // Null character - end of text
          printf("University name: %s\n", university);
          printf("\nOne letter is: %c\n", university [2]);
          printf("\nPart of the name is: %s\n", &university [3]);
```

```
        getch();
}
```

Output :

```
        University name: KKHSOU
        One letter is:H
        Part of the name is: SOU
```

4.9.2 Arrays of Strings

Arrays of strings (arrays of character arrays) can be declared and handled in a similar manner to that described for two dimensional arrays. Let us consider the following example:

Program 5:

```
#include< stdio.h>
#include<conio.h>
void main( )
{
        char names[2][8] = {"KKHSOU", "IDOL"};
        printf("Names = %s, %s\n",names[0],names[1]);
        printf("\nNames = %s\n",names);
        printf("Initials = %c. %c.\n",names[0][0],names[1][0]);
        getch();
}
```

Output :

```
        names = KKHSOU, IDOL
        names = KKHSOU
        Initials = K. I.
```

Here we declare a 2-D character array comprising two "roes" and 8 "columns". We then initialise this array with two character strings, KKHSOU and IDOL.

4.9.3 String Manipulations

C language does not provide any operator which manipulate entire strings at once. Strings are manipulated either via pointers or via special routines available from the standard string library ***string.h***. The file ***string.h*** available in the library of C has several built in functions for string manipulation. Some of them are :

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

To use these funtions we have to include the header file ***string.h*** as shown below:

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

String Length

The strlen() function is used to find the number of characters in a given string including the end-of-string character (null). The syntax is as follows:

len = strlen(ptr);

where *len* is an integer and *ptr* is the array name where the string is stored. The following program determines the length of a string which is entered through the keyboard.

Program 6: Finding the lengh of a string

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
void main()
{
    int len;
    char name[25];
     clrscr();
    printf("\nEnter the name:");
    gets(name);
    len=strlen(name);
   printf("\nLength of the string :%d",len);
    getch();
```

}

If the entered name is *KKHSOU*, then the result will be 6. If we enter *KKHSOU Assam* then strlen() will return 12 counting the blank space as one character.

EXERCISE

Q. Find the lengh of a particular string without using string library function strlen().

String Copy

The strcpy() function copies the contents of one string to another i.e., source string to destination string. The syntax is:

strcpy(str1,str2);

where str1 is the destination string and str2 is the sourse string.

String Concatenation

The strcat() function joins or places together two strings resulting in a single string. It takes two strings as argument and the resultant string is stored in the destination string. The syntax is:

strcat(s1,s2);

where s1 is the destination string and s2 is the source string.

String Compare

The strcpy() function compares two strings, character by character. It accepts two strings as parameters and returns an integer. The syntax is: **strcmp(s1,s2);**

The return value of strcmp() function depends on both the two strings which we compare.

If s2<s1, it returns -1
If s2==s1, it returns 0
If s2>s1, it returns 1

Two strings are equal if their contents are identical.

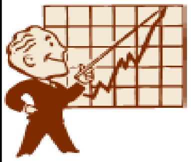
Program 7: Program for checking two string which comes first in the English dictionary.

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{
   int i;
   char s1[20],s2[20];
    clrscr();
   printf("\nEnter a string:");
    gets(s1);
   printf("\nEnter another string to compare:");
    gets(s2);
    i=strcmp(s1,s2);
    if(i==0)
           printf("\nStrings are identical");
    else if(i<0)
           printf("\nFirst string comes first");
    else if(i>0)
           printf("\nSecond string comes first");
    getch();
 }
```

Program 8: Combining two strings

```
#include<stdio.h>
#include<string.h>
#include<conio.h>
void main()
{         char s1[50],s2[20];
          printf("\nEnter your first name:");
          gets(s1);
          printf("\nEnter your last name:");
          gets(s2);
          strcat(s1,s2);
          printf("\nYour full name :%s",s1);
          getch();
```

}



CHECK YOUR PROGRESS

6. Fill in the blanks:

(i) All strings must end with a __________ character.

(ii) String function strrev() belongs to __________ header file.

(iii) __________ function appends a string to another string.

(iv) strcmp() is string library function which_______ two strings.

(v) A string is an array of ___________ .

7. Consider the following code segment:

```
  void main()
  {
          char s[100];
           scanf("%s", s);
          printf("%3s",s);
  }
```

If ***Programming Language*** is entered upon the execution of the program for s, then what will be the output? Options are given below:

(a) Pro

(b) Programming Language

(c) Programming

(d) none of these

8. State whether the following statements are true(T) or false(F).

(i) Two strings are equal if their contents are identical.

(ii) gets() function belongs to the header file *stdio.h* .

(iii) strlen() reverses a string.

(iv) If blank space exists in a string, gets() function reads the string including blank space.

(v) strcat() function takes only one string as argument.

4.10 POINTERS

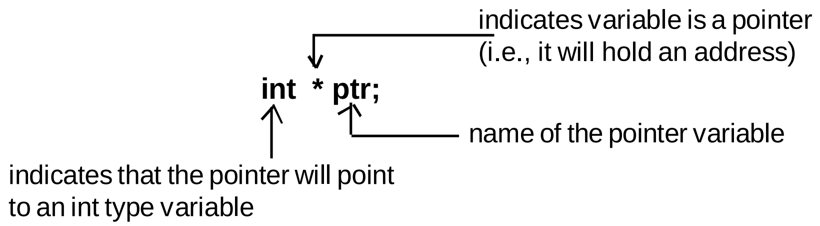
A pointer is a variable that holds the memory address of another variable. We can have a pointer to any variable type. The *unary* operator **&** gives the "address of a variable". The *indirection* or *dereference operator* * gives the "contents of a variabel pointed to by a pointer ".

4.10.1 Declaration of Pointer

Pointer variables must be declared before they may be used in C program. When a pointer variable is declared, the variable name must be preceded by an asterisk " * ". The data type that appears in the declaration refers to the object of the pointer. The general syntax of pointer declaration is:

data_type *variable;

For example, the following is a pointer declaration statement.


indicates variable is a pointer (i.e., it will hold an address)
int * ptr;
name of the pointer variable
indicates that the pointer will point to an int type variable

where *ptr* is the name of the pointer variable. The following program illustrates the use of pointer.

Program 9:

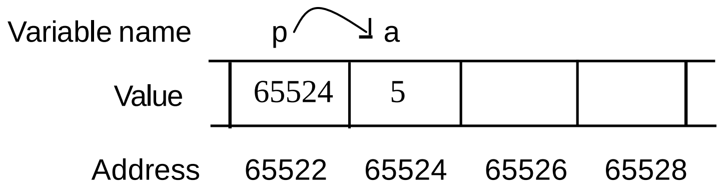
```
#include<stdio.h>
#include<conio.h>
void main( )
{
        int age,*ptr1,*ptr2;
        age = 50;                /* any numerical value */
        ptr1 = &age;             /* the address of age variable */
        ptr2 = ptr1;
        printf("The value is %d %d %d\n", age,*ptr1,*ptr2);
        *ptr1 = 29;              /* this changes the value of age */
        printf("The value is %d %d %d\n", age,*ptr1,*ptr2);
        getch();
}
```

Here "ptr1" and "ptr2" are two pointer variables. So they do not contain a variable value but an address of a variable and can be used to point to a variable. Line 7 of the above program assigns the address of "age" variable to the pointer "ptr1". Since we have a pointer to "age", we can manipulate the value of "age" by using either the variable name itself, or the pointer. Line 10 modifies the value using the pointer "ptr1". Since the pointer "ptr1" points to the variable "age", putting a star in front of the pointer name refers to the memory location to which it is pointing. Line 10 therefore assigns the value 29 to "age". Any place in the program where it is permissible to use the variable name "age", it is also permissible to use the name "*ptr1" since they are identical in meaning until the pointer is reassigned to some other variable.

Program 10: Program to demonstrate the relationships among * and & operators.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a=5;
    int *p;
    p=&a;
    clrscr();
    printf("\nAddress of a=%u", &a);
    printf("\nAddress of a=%u", p);
    printf("\nAddress of p=%u", &p);
    printf("\nValue of p=%u", p);
    printf("\nValue of a=%d", a);
    printf("\nValue of a=%d", *(&a));
    printf("\nValue of a=%d", *p);
    getch();
}
```

Output :
```
65524
65524
65522
65524
5
5
5
```
Memory address may be different with different computer. In our case if the memory address of variable **a** is 65524 then the diagrametic representation will be like this:



Pointer expressions and pointer arithmetic

Like other variables pointer variables can be used in expressions. For example if p1 and p2 are properly declared and initialized pointers, then the following statements are valid.

```
x = *p1**p2;
sum  =sum+*p1;
y = 5* - *p2/p1;
*p2 = *p2 + 10;
```

C language allows us to add integers to or subtract integers from pointers as well as to subtract one pointer from the other. We can also use short hand operators with the pointers p1+=; sum+=*p2; etc. We can also compare pointers by using relational operators the expressions such as p1 >p2 , p1==p2 and p1!=p2 are allowed.

4.10.2 Passing Pointer to a Function

Pointers are often passed to a function as arguments. This allows data type within the calling portion of the program to be accessed by the function, altered within the function, and then returned to the calling portion of the program in altered form. This is called passing arguments by reference or by address.

Here is a simple C program that illustrates the difference between ordinary arguments which are passed by value, and pointer arguments which are passed by reference.

Program 11: Arguments are passed by value

```
#include<stdio.h>
#include<conio.h>
void function1(int, int);
void main()
{
          int a =1;
          int b =2;
          printf("Before calling function1 :  a=%d      b=%d" , a  , b );
          function1(a,b);        //passed by value
          printf("After calling function1 :  a=%d      b=%d" , a  , b );
}
void function1(int a , int b )   //function definition
{
    a=5;
    b=5;
    printf("Within the  function1 :  a=%d      b=%d" , a  , b );
    return;
}
```

Output :

```
          Before calling function1 : a=1 b=2
          Within function1 :           a=5, b=5
          After calling function1 :     a=1, b=2
```

When an argument is passed by value, the data item is copied to the function. Thus, any alteration made to the data item within the function is not carried over in to the calling routine.

When an argument is passed by reference, the address of the data item is passed to the function. Here, the above example is considered again to see the differences in output. In this case we observe that original value is changed after execution of function2.

Program 12: Arguments are passed by reference

```
#include<stdio.h>
#include<conio.h>
void function2(int *, int *);
void main()
{         int a =1;
          int b =2;
          printf("Before calling function2 :  a=%d      b=%d" , a  , b );
          function2(&a,&b);       passed by reference
          printf("After calling function2 :  a=%d      b=%d" , a  , b );
}
void function2(int *pa , int  *pb )          //function
{
          *pa=5;
          *pb=5;
  printf("Within the  function2 :  *pa=%d      *pb=%d" , *pa  , *pb );
          return;
}
```

Output :

```
          Before calling function2 :  a=1  ,  b=2
          Within function2:             *pa=5,  *pb=5
          After calling function2 :      a=5,     b=5
```

4.10.3 Pointers and One-Dimensional Arrays

An array in C is declared as:

int X[5]={2,1,6,9,5};

X is an array of integers and it has five elements. If X is a one dimensional array, then the address of the first element can be expressed as either &X[0] or X. Moreover, the address of the second array element can be written as either &X[1] or as X+1 and so on. In general, the address of array element (i+1)can be expressed as either &X[i] or X+i . Since &X[i] and X+i both represent the address of the i^{th} element of X , it would seem reasonable that X[i] and *(X+i) both represent the contents of that address. i.e , the value of i^{th} element of X.

An array is actually a pointer to the 0th element of the array. Dereferencing the array name will give the 0th element. This gives us a range of equivalent notations for array access. In the following examples, ARR is an array.

Array access	**Pointer equivalent**
ARR[0]	*ARR
ARR[2]	*(ARR+2)
ARR[n]	*(ARR+n)

There are some differences between arrays and pointers. The array is treated as a constant in the function where it is declared. This means that we can modify the values in the array, but not the array itself, so statements like ARR ++ are illegal, but ARR[n] ++ is legal.
Let us consider an **int** variable called **i**. Its address could be represented by the symbol **&i**. If the pointer is to be stored as a variable, it should be stored like this:

```
int *p = &i;
```

int * is the notation for a pointer to an **int**. The operator & returns the address of its argument.

The other operator which gives the value at the end of the pointer is *. For example: i = *p;

4.11 DYNAMIC MEMORY ALLOCATION

When an array is declared as above, memory is allocated for the elements of the array when the program starts, and this memory remains allocated during the lifetime of the program. This is known as **static** array allocation.

Until this point, the memory allocation for our program has been handled automatically when compiling. However, sometimes the computer doesn't know how much memory to set aside (for example, when you have an unsized array). It may happen that you don't know how large an array you will need (or how many arrays). In this case it is convenient to allocate an array while the program is running. This is known as **dynamic memory allocation**. Dynamic data structures provide flexibility in adding , deleting or rearranging data items at run time . Dynamic memory management permit us to allocate additional memory space or to release unwanted space at run time.

malloc and free :

A block of memory may be allocated using the function **malloc**. The **malloc()** function reserves a block of memory of specified size and returns a pointer of type void. Syntax is as follows:

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

For example, x =(int *)malloc (100 * sizeof(int));

Here a memory space equivalent to 100 times the size of an integer byte is reserved and the address of the first byte of the memory allocated is assigned to the pointer **x** of type integer.

malloc requires one argument - the number of bytes which we want to allocate dynamically. If the memory allocation was successful, malloc will return a void pointer. We can assign this to a pointer variable, which will store the address of the allocated memory. If memory allocation failed (for example, if you're out of memory), malloc will return a NULL pointer. Passing the pointer into free will release the allocated memory - it is good practice to free memory when you've finished with it. The general format of free() function is:

```
int free(pointer);

char *pointer;
```

The following program will ask you how many integers you would like to store in an array. It will then allocate the memory dynamically using malloc and store a certain number of integers, print them out, then releases the used memory using free.

Program 13:

```
#include <stdio.h>
#include <stdlib.h>  /* required for the malloc and free functions */
#include<conio.h>
void main()
```

```
{
        int number;
        int *ptr,i;
        clrscr();
        printf("How many ints would you like to store? ");
        scanf("%d", &number);
        ptr =(int *)malloc(number*sizeof(int));  // allocate memory
        if(ptr!=NULL)
        {
                for(i=0 ; i<number ; i++)
                {
                        *(ptr+i) = i;
                }
                for(i=number ; i>0 ; i--)
                {
                        printf("%d\n", *(ptr+(i-1)));  /* print out in
                                                reverse order */
                }
        free(ptr);     // free allocated memory
    }
    else
    {
          printf("\nMemory allocation failed - not enough memory");
    }   //end bracket of if-else
}// end of main
```

If we enter 4

Output : How many ints would you like store? 4

```
3
2
1
0
```

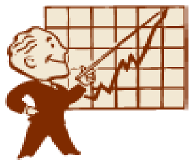
<u>calloc</u> :

calloc is similar to malloc, but the main difference between the two is that in case of callocc the values stored in the allocated memory space is zero by default. With malloc, the allocated memory could have any value. Calloc requires two arguments. The first is the number of variables which we like to allocate memory for. The second is the size of each variable. Like malloc, calloc will return a void pointer if the memory allocation was successful, else it'll return a NULL pointer. Syntax of calloc is as follows:

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

realloc :

The realloc() function is used to change the size of previously allocated block. Suppose, we have allocated a certain number of bytes for an array but later find that we want to add values to it. We could copy everything into a larger array, which is inefficient, or we can allocate more bytes using realloc, without losing our data. *realloc()* takes two arguments. The first is the pointer referencing the memory. The second is the total number of bytes you want to reallocate. Passing zero as the second argument is the equivalent of calling free. Once again, realloc returns a void pointer if successful, else a NULL pointer is returned.



CHECK YOUR PROGRESS

9. State whether the following statements are true(T) or false(F).

(i) The statement *p++ ; increments the content of the memory location pointed by p.

(ii) The address operator is obtained by *.

(iii) *p++ ; increments the integer pointed by p.

(iv) The address operator (&) is the inverse of the de-referncing operator (*).

(v) Arrays cannot be returned by functions, however pointer to array can be returned.

10. Choose the correct option:

(i) Which is the correct way to declare an integer pointer ?

(a) int_ptr x; (b) int *ptr; (c)*int ptr; (d)*ptr;

(ii) In the expression **float *p;** which is represented as type float?

(a)The address of p (b)The variable p

(c) The variable pointed to by p (d) None of the above

(iii) A pointer is

(a) Address of variable

(b)A variable for storing address

(c) An indirection of the variable to be accessed next.

(d)None of the above.

(iv) Assuming that **int num[ ]** is an one-dimensional array of type int, which of the following refers to the third element in the array?

(a) *(num+4); (b)*(num+2);

(c)num+2; (d)p=&a[3];

(v) Consider the following two definitions int a[50]; int *p; which of the following statement is incorrect?

(a) p=a+3; (b) a=p;

(c) p=&a[3]; (d)None of these

11. How does the use of pointers economize memory space?

4.12 LET US SUM UP

- Array by definition is a variable that hold multiple elements which has the same data type.
- Array elements are stored in contiguous memory locations and so they can be accessed using pointers.
- A string is nothing but an array of characters terminated by null character "\0".
- The header file of string library function is *string.h*
- *strlen()* returns the number of characters in the string, not including the *null character*
- *strcmp()* takes two strings and compares them. If the strings are equal, it returns 0. If the first is greater than the 2nd, then it returns *some* value greater than 0. If the first is less than the second, then it returns *some* value less than 0.
- *strrev()* reverses a string.
- *strcat()* joins two strings.
- A pointer variable can be assigned the address of an ordinary variable (Eg, PV=&V).
- A pointer variable can be assigned the value of another pointer variable (Eg, PV=PX) provided both pointers point to

object of the same data type.

- A pointer variable cannot be multiplied by a constant; two pointer variables cannot be added.
- On incrementing a pointer it points to the next location of its type.

4.13 FURTHER READINGS

1. Balagurusamy, E: *Programming in ANSI C*, Tata McGraw-Hill publication.
2. Gottfried Byron S: *Programming with C*, Tata McGraw-Hill publication.
3. Venugopal, K.R, Prasad, S.R: *Mastering C,* Tata McGraw-Hill publication.

6.14 ANSWERS TO CHECK YOUR PROGRESS

1. (i) True (ii) False (iii) False (iv) True (v)True (vi) True
2. (i) Homogeneous (ii) square bracket [ ]
   (iii) commas (iv) randomly
3. (d) All are correct
4. 16
5. (a) 6 (b) 9
6. (i) null (ii) string.h (iii) strcat() (iv) compares (v) characters
7. (c) Programming
8. (i) True (ii) True (iii) False (iv) True (v) False
9. (i) False (ii) False (iii) True (iv) False (v)True
10. (i) (b) int *ptr;
    (ii) (c)The variable pointed to by p
    (iii) (b) A variable for storing address
    (iv) (b)*(num+2);
    (v) (b) a=p;
11. Pointers are variables which hold the addresses of other variables. A compiler allocates an address at runtime for each variable and retains this till program execution is completed. Thus, entire