***MODULE – 4***

**Strings and Pointers:** Introduction, string taxonomy, operations on strings, miscellaneous string and character functions, arrays of strings.

**Pointers:** Introduction to pointers, declaring pointer variables, Types of pointers, passing arguments to functions using pointers.

**Textbook: Chapter 13.1-13.6, 14 -14.7**

## CHAPTER 1 STRINGS

### 1.1 INTRODUCTION TO STRINGS:

A string is a sequence of characters that is treated as a single data item.

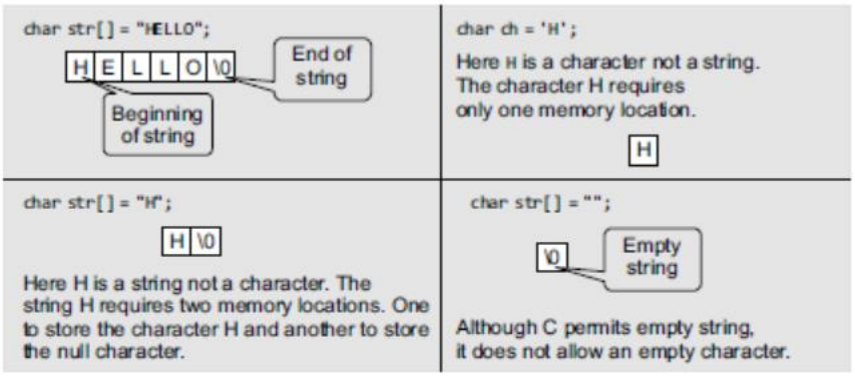
In C language, a string is a null-terminated character array. This means that after the last character, a null character ('\0') is stored to signify the end of the character array.

For example, if we write **char str[] = "HELLO";** then we are declaring an array that has five characters, namely, H, E, L, L, and O.

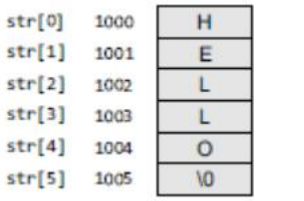
Apart from these characters, a null character ('\0') is stored at the end of the string. So, the internal representation of the string becomes HELLO'\0'.

To store a string of length 5, we need 5 + 1 locations (1 extra for the null character).

**The name of the character array (or the string) is a pointer to the beginning of the string.**




Difference between character storage and string storage




Memory representation of a character array

## DECLARING AND INITIALIZING STRING VARIABLES

C does not support strings as a data type. However, it allows us to represent strings as character arrays in C, therefore, a string variable is any valid C, variable name and is always declared as an array of characters.

**The general form of declaration of a string variable is:**

**char string_name[size];**

The **size** determines the number of characters in the string_ name.

Some examples are:

char city [10] ;

char name[30];

When the compiler assigns a character string to a character array, it automatically supplies a null character ('\0') at the end of the string. Therefore, the size should be equal to the maximum number of characters in the string plus one.

Like numeric arrays, character arrays may be initialized when they are declared.

C permits a character array to be initialized in either of the following two forms:

- **Initialization with string constant**

  char city [9] = "NEW YORK";

- **Initialization with character constant**

  char city [9]={‘N’, ‘E’, ‘W’,’ ‘,’Y', '0', 'R', 'K', '\0'};

  The reason that city had to be 9 elements long is that the string NEW YORK contains 8 character and one element space is provided for the null terminator.

- **Without size**

C also permits us to initialize a character array without specifying the number of elements. In such cases, the size of the array will be determined automatically, based on the number of elements initialized.

For example, the statement char string [ ] = {‘G’, ‘O’,’O’, ‘D’,'\0'}; defines the array string as a five element array.

- **Partial Initialization**

We can also declare the size much larger than the string size in the initializer. That is, the statement. char str[10] = "GOOD"; is permitted. In this case, the computer creates a character array of size 10, places the value "GOOD" in it, terminates with the null character, and initializes all other elements to NULL. The storage will look like:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| G | O | O | D | \0 | \0 | \0 | \0 | \0 | \0 |

However, the following declaration is illegal. char str2[3] = "GOOD"; This will result in a compile time error.

Also note that we cannot separate the initialization from declaration. That is,

char str3[5];

str3 = "GOOD";

is not allowed. Similarly,

char sl[4] = "abc";

char s2[4];

s2 = sl; /* Error */

is not allowed.

### 1.1.1 READING STRINGS

1. Using scanf function

2. gets() function

3.Using getchar(), getch() or getche() function repeatedly.

### 1. Using scanf Function

The input function scanf can be used with %s format specification to read in a string of characters.

Example: char address[10]

scanf("%s", address);

The problem with the scanf function is that it terminates its input on the **first white space** it finds. A white space includes blanks, tabs, new lines etc.,

NEW YORK

then only the string "NEW" will be read into the array address, since the blank space after the word 'NEW' will terminate the reading of string. The scanf function automatically terminates the string that is read with a null character and therefore the character array should be large enough to hold the input string plus the null character.

Unlike previous scanf calls, in the case of character arrays, the **ampersand (&)** is not required before the variable name.

**Write a program to input and output a name**

```
#include<stdio.h>

void main()

{

    char name[20];

    printf(“Enter a name”);

    scanf(“%s”,name);
```

```
    printf("Name is %s",name);
}
```

**Output:**

**Enter a name: bala guru**

**Name is bala**

**READING A LINE OF TEXT**

C supports a format specification known as the %[..] that can be used to read a line containing a variety of characters, including whitespaces.

For example.

```
char line[80];
scanf("%[^\n]",line);
printf("%s",line);
```

will read a line of input from the keyboard and display the same on the screen.

2. **Using getchar(), getch(), getche()**

The getchar function is used to read a single character. It takes the form :

```
char ch;
ch=getchar( ) ;
```

We can use this function repeatedly to read successive single characters from the input and place them into a character array. Thus, an entire line of text can be read and stored in an array.

The reading is terminated when the newline character ("\n") is entered and the null character is then inserted at the end of the string.

**Program :**

**Write a program to read a line of text (use getchar) containing a series of words from the terminal.**

```
#include<stdio.h>
void main( )
  {
     char  line[80], character;
     int i=0;
     do
     {
        character = getchar();
        line[i]   = character;
         i++;
     }
     while(character != '\n');
     line[i] = '\0';
     printf("\n%s\n", line);
  }
```

**Output**

**Programming in C is interesting.**

**Programming in C is interesting.**

**3. Using gets() function**

This is a function with one string parameter and called as under:

```
gets (str);
```

str is a string variable declared properly. It reads characters into str from the keyboard until a new-line character is encountered and then appends a null character to the string. Unlike scanf, it does not skip whitespaces. For example the code segment

```
char 1ine [80] ;
gets (1ine);
printf ("%s", line);
```

reads a line of text from the keyboard and displays it on the screen.

**Write a program to input and output a name**

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

```
void main()
{
    char name[20];
    printf("Enter a name");
    gets(name);
    printf("Name is %s",name);
}
```

**Output:**

**Enter a name:bala guru**

**Name is bala guru**

**Program :**

Write a program to find the number of vowels and consonants in a text string.

```
#include<stdio.h>
#include<string.h>
int main()
{
    int i,vc=0,cc=0;
    char s[100],ch;
    printf("\n Enter the sentence");
    gets(s);
    for(i=0;i<strlen(s);i++)
    {
        if(isalpha(s[i]))
        {
            ch=tolower(s[i]);
            if(ch=='a'||ch=='e'||ch=='i'||ch=='o'||ch=='u')
            vc++;
```

```
                else
                cc++;
            }
        }
        printf("\n Vowel count=%d\n",vc);
        printf("\n Consonant count=%d\n",cc);
        return 0;
    }
```

**Test cases**

| Test No | Input Parameters | Expected Output | Obtained Output |
|---|---|---|---|
| 1 | Hai Hello | VC=4 ,CC=4 | VC=4 ,CC=4 |
| 2 | Umberella | VC=4, CC=5 | VC=4, CC=5 |

**Differences between gets() and scanf()**

| gets() | scanf() |
|---|---|
| It reads characters from the keyboard until a new-line character is encountered and then appends a null character to the string. | It reads characters from the keyboard and terminates its input on the first white space it finds. It appends a null character at the end of the string. |
| It is unformatted input. | It is formatted input. |
| It is used to input only characters and strings | It is used to input any type of data that is integers, characters, floating point numbers and strings |
| **Syntax:** gets(string) **Example:** gets(name); | **Syntax:** scanf("%s",string); **Example:** scanf("%s",name); |

### 1.1.2 WRITING STRINGS

1. Using printf function
2. Using puts() function
3. Using putchar() function repeatedly

**1. Using printf Function**

We have used extensively the printf function with %s format to print strings to the screen. The format %s can be used to display an array of characters that is terminated by the null character.

For example, the statement

```
printf("%s", name);
```

can be used to display the entire contents of the array name.

**2. Using putchar Function**

C supports another character handling function putchar to output the values of character variables. It takes the following form:

```
char ch = 'A';
putchar(ch);
example -printf("%c", ch);
```

We can use this function repeatedly to output a string of characters stored in an array using a loop. Example:

```
int i=0;
char name[6] = "PARIS";
 while(name[i]!='\0')
{
        putchar(name[i]);
        i++;
}
```

**3. Using puts Function**

Another and more convenient way of printing string values is to use the function puts declared in the header file <stdio.h>.

```
puts ( str );
```

where str is a string variable containing a string value. This prints the value of the string variable str and then moves the cursor to the beginning of the next line on the screen. For example, the program segment

```
char 1i ne [80] ;
gets(1ine);
puts(line);
```

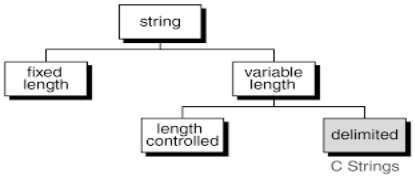
reads a line of text from the keyboard and displays it on the screen.

**Write a program to input and output a name**

```
#include<stdio.h>

void main()

{

        char name[20];

        printf("Enter a name");

        gets(name);

        puts("Name is");

        puts(name);

}
```

## 1.2 STRING TAXONOMY

We can store a string either in fixed-length format or in variable-length format. In C, a string is a variable length of array of characters that is delimited by the null character.




**Fixed-length strings**

When storing a string in a fixed-length format, we need to specify an appropriate size for the string variable.

If the size is too small, then we will not be able to store all the elements in the string.

If the size is large, then unnecessarily memory space is wasted.

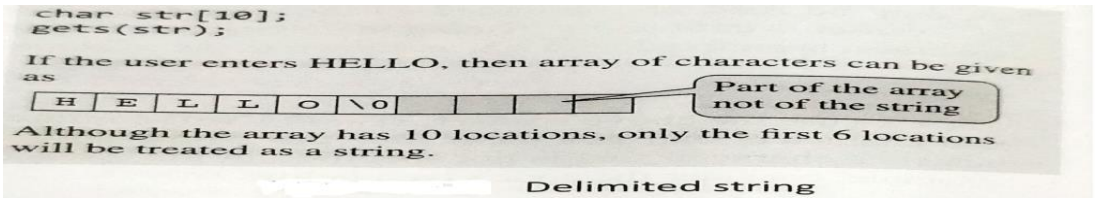
**Variable-length strings**

We can use a variable length format in which the string can be expanded or contracted to accommodate the elements in it.

To use a variable length string format we need a technique to indicate the end of the elements that are a part of the string. This can be done either by using length-controlled string or a delimiter.

**Length-controlled strings** In a length-controlled string, we need to specify the number of characters in the string.

**Delimited strings** In this format, the string is ended with a delimiter. The delimiter is then used to identify the end of the string. For example, in English language every sentence is ended with a full-stop(.). Null character is the most commonly used string delimiter in the C language.




## 1.3 OPERATIONS ON STRINGS

1. **Arithmetic Operations on Characters**
2. **Putting Strings Together (Concatenation)**
3. **Comparison Of Two Strings**
4. **String length**
5. **String Copy**
6. **Reversing a String**

### 1. SOME ARITHMETIC OPERATIONS ON CHARACTERS

C allows us to manipulate characters the same way we do with numbers.

1. To write a character in its integer representation, we may write it as an integer. For example, if the machine uses the ASCII representation, then.

```
x = 'a';
printf("%d\n",x);
```

Will display the number 97 on the screen.

- It is also possible to perform arithmetic operations on the character constants and variables. For example x='z' -1;

is a valid statement. In ASCII, the value of 'x' is 122 and therefore, the statement will assign the value 121 to the variable x.

- We may also use character constants in relations expressions. For example , the expression

  ch >= ‘A’ && ch <=’Z’

  would test whether the character contained in the variable ch is an upper case letter.
- We can convert a character digit to its equivalent integer value using the following relationship:

  x = character - '0';

  where x is defined as an integer variable and character contains the character digit. For example, let us assume that the character contains the digit '7', Then

  x = ASCII value of ‘7’- ASCII value of '0'

  = 55-48

  =7
- The C library supports a function that converts a string of digits into their integer values. The function takes the form

  x = atoi(string);

  x is an integer variable and string is a character array containing a string of digits. Consider the following segment of a program:

  char number[5] = "1988";

  year = atoi(number);

  number is a string variable which is assigned the string constant "1988". The function atoi converts the string "1988" (contained in number) to its numeric equivalent 1988 and assigns it to the integer variable year. String conversion functions are stored in the header file <stdlib.h>

**Program :**

Write a program which would print the alphabet set a to z and A to Z in decimal and character form.

```
void  main()
{
    char  c;
    for( c = 65 ; c <= 122 ; c ++ )
    {
```

```
if( c > 90  &&  c < 97 )
continue;
 printf("%d - %c", c, c);
      }
}
```

Output

```
  65 - A
  66 - B
. . . .
 121 – y
 122 - z
```

### 1. PUTTING STRINGS TOGETHER

Just as we cannot assign one string to another directly, we cannot join two strings together by the simple arithmetic addition.

That is, the statements such as

```
String3 = string1 + string2;
string2 = string1 + "hello";
```

are not valid. The characters from string1 and string2 should be copied into the string3 one after the other. The size of the array string3 should be large enough to hold the total characters.

The process of combining two strings together is called concatenation.

**Program : combine two strings(with space in between) without using library function**

```
#include<stdio.h>
void main()
{
        char s1[20],s2[20],s3[40];
        int i=0,j=0;
        printf("Enter string1: ");
        gets(s1);
        printf("Enter string2: ");
        gets(s2);
        while(s1[i]!='\0')
```

```
        {
                s3[i]=s1[i];
                i++;
        }
        s3[i]=' ';
        while(s2[j]!='\0')
        {
                s3[i+1+j]=s2[j];
                j++;
        }
        s3[i+1+j]='\0';
        printf("Concatenated string is %s",s3);
}
```

**Output:**

**Enter string1: good**

**Enter string2: girl**

**Concatenated string is good girl**

### 2. COMPARISON OF TWO STRINGS

Once again, C does not permit the comparison of two strings directly. That is, the statements such as

```
if(namel == name2)
if(name == "ABC")
```

are not permitted. It is therefore necessary to compare the two strings to be tested, character by character. The comparison is done until there is a mismatch or one of the strings terminates into a null character, whichever occurs first. The following segment of a program illustrates this.

**Program : compare two strings without using library function**

```
#include<stdio.h>
void main()
{
        char s1[20],s2[20];
        int i=0,k;
```

```
        printf("Enter string1: ");
        gets(s1);
        printf("Enter string2: ");
        gets(s2);
        while(s1[i]==s2[i])
        {
                if(s1[i]=='\0')
                        break;
                i++;
        }
        k=s1[i]-s2[i];
        if(k==0)
                printf("Strings are same");
        else
                printf("Strings are different");
}
```

**Output:**

**Enter string1: good**

**Enter string2: good**

**Strings are same**

### 3. STRING LENGTH

The length of the string can be found by counting character by character until the null character.

**Program : Length of the string without a library function**

```
        #include<stdio.h>
        void main()
        {
                char s1[20];
                int i=0;
                printf("Enter a string: ");
                gets(s1);
                while(s1[i]!='\0')
```

```
        {
            i++;
        }
        printf("Length of the string is %d",i);
    }
```

**Output:**

**Enter a string: good**

**Length of the string is 4**

### 4. STRING COPY

We cannot assign one string to another directly, we can copy a string to another character by character.

**Program : String copy without a library function**

```
#include<stdio.h>
void main()
{
    char s1[20],s2[20];
    int i=0;
    printf("Enter a string: ");
    gets(s1);
    while(s1[i]!='\0')
    {
        s2[i]=s1[i];
        i++;
    }
    s2[i]='\0';
    printf("Copied string  is %s",s2);
}
```

**Output:**

**Enter a string: hello girl**

**Copied string is hello girl**

### 5. STRING REVERSING

We cannot assign one string to another directly, we can copy a string to another character by character.

**Program : String copy without a library function**

```
#include <stdio.h>
#include <string.h>
int main()
{
        char s1[50],s2[50];
        int i = 0, j =0;
        printf ("Enter a string: ");
        gets(s1);
        for(j=strlen(s1)-1;j>=0;j--)
        {
        s2[i] = s1[j];
        i++;
        }
        printf ("The reversed string is %s", s2);
        return 0;
}
```

```
#include <stdio.h>
int main()
{
        char s1[50],s2[50];
        int i=0, j=0,n=0;
        printf ("Enter a string: ");
        gets(s1);
        while(s1[n]!='\0')
        {
                n++;
        }
        for(j=n-1;j>=0;j--)
        {
                s2[i] = s1[j];
                i++;
        }
        printf ("The reversed string is %s", s2);
        return 0;
}
```

**Output:**

Enter a string: good

The reversed string is doog

## 1.5 MISCELLANEOUS STRING AND CHARACTER FUNCTIONS [STRING HANDLING FUCTIONS]

### *1.5.1 Character Manipulation Functions*

Some character functions are contained in the file ctype.h and therefore the statement #include <ctype.h> is included in the program.

| Function | Test (c is a character variable) |
|---|---|
| isalnum(c) | Is c an alphanumeric character? |
| isalpha(c) | Is c an alphabetic character? |

| | |
|---|---|
| isdigit(c) | Is c a digit? |
| islower(c) | Is c lower case letter? |
| isprint(c) | Is c a printable character? |
| ispunct(c) | Is c a punctuation mark? |
| isspace(c) | Is c a white space character? |
| isupper(c) | Is c a upper case letter? |
| toupper(c) | converts c to upper case |
| tolower(c) | converts c to lower case |

### *1.5.2 String Manipulation Functions*

Fortunately, the C library supports a large number of string-handling functions that can be used to carry out many of the string manipulations. Following are the most commonly used string handling functions (stored in header file string.h).

| S:No | Function | Action |
|---|---|---|
| 1 | strcat( ) | Concatenates two strings |
| 2 | strncat() | Concatenates first leftmost **n** characters of string2 to string1 |
| 3 | strcmp( ) | Compares two strings |
| 4 | strncmp() | Compares first leftmost **n** characters in both strings |
| 5 | strcpy( ) | Copies one string over another |
| 6 | strncpy() | Copies first leftmost **n** characters of source string to target. |
| 7 | strlen( ) | Finds the length of a string |
| 8 | strrev() | Reverses a string |
| 9 | strstr() | Searches for string2 in string1 |
| 10 | strchr() | Locate the first occurrence of the character |
| 11 | strrchr() | Locate the last occurrence of the character |

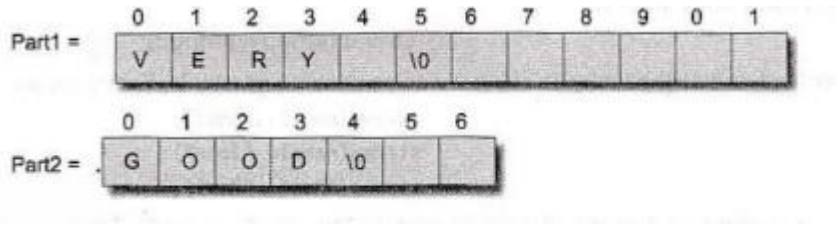
**1. strcat( ) Functions**

The strcat functions joins two strings together. It takes the following form:

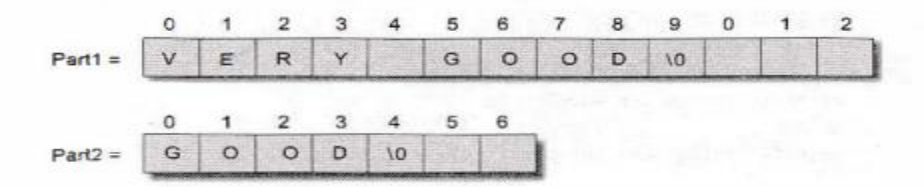
strcat(string1,string2);

string1 and string2 are character arrays. When the function strcat is executed, string2 is appended to string1.It does so by removing the null character at the end of string1 and placing string2 from there.

The string at string2 remains unchanged. For example, consider the following three strings:




Execution of the statement will result in :




We must make sure that the size of string1 (to which string2 is appended) is large enough to accommodate the final string.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[20];
        printf("Enter string1:");
        gets(s1);
        printf("Enter string2:");
        gets(s2);
        strcat(s1,s2);
        printf("concatenated string is %s",s1);
}
```

**Output:**

**Enter string1:good**

**Enter string2: girl**

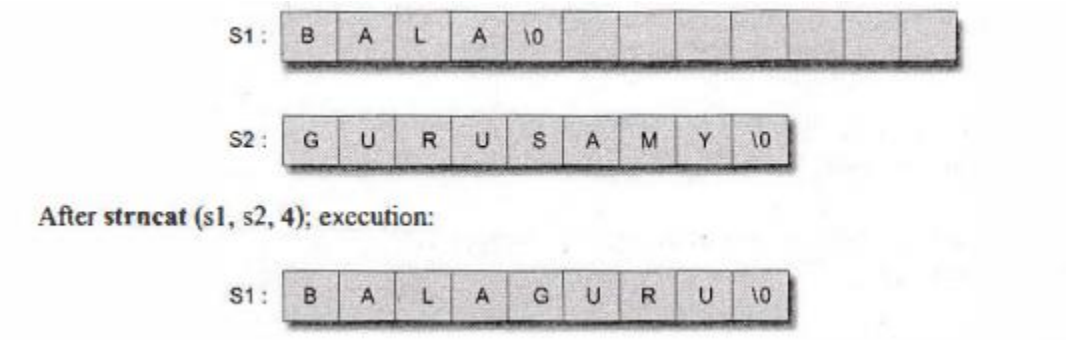
**concatenated string is good girl**

**2. strncat Function**

This is another concatenation function that takes three parameters as shown below:

strncat (sl, s2, n);

This call will concatenate the left-most n characters of s2 to the end of s1. Example:




```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[20];
        printf("Enter string1:");
        gets(s1);
        printf("Enter string2:");
        gets(s2);
        strncat(s1,s2,4);
        printf("concatenated string is %s",s1);
}
```

**Output:**

**Enter string1:good**

**Enter string2:girls**

**concatenated string is goodgirl**

**3. strcmp() Function**

The strcmp function compares two strings and returns value 0 if they are equal. If they are not, it has the numeric difference between the first nonmatching characters in the strings. It takes the form:

strcmp(string1, string2);

string1 and string2 may be string variables or string constants.

Examples are:

```
strcmp(namel, name2);
strcmp(namel, "John");
strcmp("Rom", "Ram");
```

For example, the statement

```
strcmp("their", "there");
```

will return a value of -9 which is the numeric difference between ASCII "i" and ASCII "r". That is, "i" minus "r" in ASCII code is -9. If the value is negative, string1 is alphabetically above string2.

| | |
|---|---|
| returns **0** | If both strings are same |
| returns **negative** | If string1 is alphabetically above string2 (string1<string2) |
| returns **positive** | If string1 is alphabetically below string2 (string1>string2) |

```
#include<stdio.h>
#include<string.h>
void main()
{
    char s1[20],s2[20];
    int k;
    printf("Enter string1:");
    gets(s1);
    printf("Enter string2:");
    gets(s2);
    k=strcmp(s2,s1);
    if(k==0)
        printf("same strings");
    else
        printf("not same");
}
```

**Output:**

**Enter string1:good**

**Enter string2:good**

**same strings**

**4. strncmp Function**

This function has three parameters as illustrated in the function call below:

strncmp (sl, s2, n);

this compares the left-most n characters of s1 to s2 and returns.

(a) 0 if they are equal;

(b) negative number, if s1 sub-string is less than s2**(alphabetically above)**; and

(c) positive number, otherwise.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[20];
        int k;
        printf("Enter string1:");
        gets(s1);
        printf("Enter string2:");
        gets(s2);
        k=strncmp(s2,s1,5);
        if(k==0)
                printf("same strings");
        else
                printf("not same");
}
```

**Output:**

**Enter string1:hello dad**

**Enter string2:hello world**

**Same strings**

### 5. strcpy( ) Function

The strcpy function works almost like a string-assignment operator.

It takes the form:

strcpy(string1, string2);

and assigns the contents of string2 to string1. string2 may be a character array variable or a string constant. Here string1 is target and string2 is source.

For example, the statement

strcpy(city, "DELHI");

will assign the string "DELHI" to the string variable city.

Similarly, the statement

strcpy(cityl, city2);

will assign the contents of the string variable city2 to the string variable city1. The size of the array city1 should be large enough to receive the contents of city2.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[20];
        printf("Enter a string:");
        gets(s1);
        strcpy(s2,s1);
        printf("Copied string is %s",s2);
}
```

**Output:**

**Enter a string:hello girl**

**Copied string is hello girl**

### 6. strncpy Function

**strncpy()** function that copies only the left-most· n characters of the source string to the target string variable. This is a three-parameter function. strncpy(sl, s2, n); where n is a integer.

strncpy(sl, s2, 5);

This statement copies the first 5 characters of the source string s2 into the target string s1.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[20];
        printf("Enter a string:");
        gets(s1);
        strncpy(s2,s1,5);
        printf("Copied string is %s",s2);
}
```

**Output:**

**Enter a string:hello girl**

**Copied string is hello**

**7. strlen( ) Function**

This function counts and returns the number of characters in a string. It takes the form

```
n = strlen(string);
```

Where n is an integer variable, which receives the value of the length of the string. The argument may be a string constant. The counting ends at the first null character. **strlen() gives the length of the string excluding the null character.**

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20];
        int n;
        printf("Enter a string:");
        gets(s1);
        n=strlen(s1);
        printf("Length of the string is %d",n);
}
```

**Output:**

**Enter a string:good**

**Length of the string is 4**

**8. strrev( ) Function**

This function reverses the string. It takes the form: strrev(string); The reversed string is stored in the string itself.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20];
        printf("Enter a string:");
        gets(s1);
        strrev(s1);
        printf("Reversed string is %s",s1);
}
```

**Output:**

**Enter a string:good**

**Reversed string is doog**

**9. strstr Function**

It is a two-parameter function that can be used to locate a sub-string in a string.

This takes the forms: strstr (sl, s2);

strstr (sl, "ABC");

The function strstr searches the string s1 to see whether the string s2 is contained in s1. If yes, the function returns the position of the first occurrence of the sub-string. Otherwise, it returns a NULL.

Example.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20],s2[20];
```

```
        printf("Enter string1: ");
        gets(s1);
        printf("Enter string2: ");
        gets(s2);
        if (strstr (s1, s2) == NULL)
                printf("substring is not found");
        else
                printf("s2 is a substring of s1");
}
```

**Output:**

**Enter string1:good for good**

**Enter string2:for**

**s2 is a substring of s1**

### 10. strchr Function

It is a two-parameter function that can be used to determine the existence of a character in a string.

This takes the form: strchr(s1,'m'); will locate the first occurrence of the char 'm' in the string s1.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20];
        char *pos;
        printf("Enter a string: ");
        gets(s1);
        pos=strchr(s1,'n');
        if(pos)
                printf("First occurrence of the character is  found at %d",pos);
        else
                printf("character is not found");
```

```
}
```

**Output:**

Enter a string: gone

First occurrence of the character is found at 1000

**11. strrchr Function**

It is a two-parameter function that can be used to determine the existence of a character in a string.

This takes the form: strrchr(s1,'m'); will locate the last occurrence of the char 'm' in the string s1.

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[20];
        char *pos;
        printf("Enter a string: ");
        gets(s1);
        pos=strrchr(s1,'n');
        if(pos)
                printf("Last occurence of the character is  found at %d",pos);
        else
                printf("character is not found");
}
```

Output:

Enter a string: gone

Last occurrence of the character is found at 1000

**PROGRAM**

Write a program to check if the input string is palindrome or not.

```
#include<stdio.h>

#include<string.h>
```

```
void main()
{
    char s1[20],s2[20];
    printf("Enter a string:");
    gets(s1);
    strcpy(s2,s1);
    strrev(s2);
    if(strcmp(s1,s2)==0)
        printf("string is a palindrome");
    else
        printf("string is not a palindrome");
}
```

**Output:**

**Enter a string: Malayalam**

**string is a palindrome**

**Arrays of strings – (Table of strings or 2-D Character array)**

List of character strings can be stored in a 2-D character array. We often use list of names of students in a class, list of cities etc.

These are called as array of strings or table of strings

Ex – name[3][10];

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| r | a | m | \0 | | | | | | |
| r | a | v | i | \0 | | | | | |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| r | a | m | y | a | \0 | | | | |

***DECLARATION:***

**char array_name[row_size][col_size];**

rowsize= number of names (number of strings)

colsize=maximum number of characters in a string

***INITIALIZATION***

Arrays of strings may be initialized (given initial values) when they are declared.

2 types of initialization -

1) Compile time initialization

2) Run time initialization

***COMPILE TIME INITIALIZATION***

1) char name[3][10]={“ram”,”ravi”,”ramya”};

   name[0]=”ram”

   name[1]=”ravi”

   name[2]=”ramya”

2) without size

   char name[][10]={“ram”,”ravi”,”ramya”};

To access

- Each name only row index is used , (i.e) to access $i^{th}$ name - name[i] is used.

  Ex- second name is at index 1, so name[1] is used

- Each character both row and column index is used , (i.e) to access $i^{th}$ name $j^{th}$ character name[i][j] is used.

  Ex- Second name (index 1) third character(index 2)- name[1][2] is used

***RUN TIME INITIALIZATION***

```
for(i=0;i<n;i++)
        scanf(“%s”,name[i]);
```

(or)

```
for(i=0;i<n;i++)
        gets(name[i]);
```

**To output we can use printf() or puts()**

```
for(i=0;i<n;i++)
        printf("\n%s",name[i]);
```

**(or)**

```
for(i=0;i<n;i++)
        puts(name[i]);
```

**WAP to input n names and output the same.**

```
void main()
{
        int n,i;
        char name[10][20];
        printf("enter n");
        scanf("%d",&n);
        printf("enter names");
        for(i=0;i<n;i++)
                gets(name[i]);
        printf("names are");
        for(i=0;i<n;i++)
                printf("\n %s",name[i]);
}
```

***Lab Program***

***Write functions to implement string operations such as compare, concatenate, and find string length. Use the parameter passing techniques.***

**Program**

```
#include<stdio.h>
int strlength(char str1[50]);
void strconcat(char str1[50],char str2[50]);
int strcompare(char str1[50],char str2[50]);
int strlength(char str[50])
{
  int i=0;
  while(str[i]!='\0')
        i++;
  return i;
}
void strconcat(char str1[50],char str2[50])
{
int i=0,l;
l=strlength(str1);
while(str2[i]!='\0')
 {
     str1[l+i]=str2[i];
     i++;
 }
str1[l+i]='\0';
}
int strcompare(char str1[50],char str2[50])
{
  int i=0,k;
  while(str1[i]==str2[i])
  {
        if(str1[i]=='\0')
              break;
        i++;
  }
```

```
  k=str1[i]-str2[i];
  return k;
}
int main()
{
char source1[50],source2[50];
int length1,length2,k;
printf("\n Enter the source string 1:");
gets(source1);
printf("\n Enter the source string 2:");
gets(source2);
length1=strlength(source1);
length2=strlength(source2);
printf("\n string length of string 1 is %d",length1);
printf("\n string length of string 2 is %d",length2);
k=strcompare(source1,source2);
if(k==0)
  printf("\n Both string are same");
else
  printf("\n Both string are different");
strconcat(source1,source2);
printf("\n concatenated string is ");
puts(source1);
return 0;
}
```

**Test cases**

| Test No | Input Parameters | Expected Output | Obtained Output |
|---|---|---|---|
| 1 | Enter the source string1 :good<br>Enter the source string 2:good | string length of string 1 is 4<br>string length of string 2 is 4 | string length of string 1 is 4<br>string length of string 2 is 4 |

| | | | |
|---|---|---|---|
| | | Both strings are same<br>concatenated string is goodgood | Both string are same<br>concatenated string is goodgood |
| 2 | Enter the source string1 :good<br>Enter the source string 2:girls | string length of string 1 is 4<br>string length of string 2 is 5<br>strings are different<br>concatenated string is goodgirls | string length of string 1 is 4<br>string length of string 2 is 5<br>strings are different<br>concatenated string is goodgirls |

***MODULE – 4***

**Strings and Pointers:** Introduction, string taxonomy, operations on strings, miscellaneous string and character functions, arrays of strings.

**Pointers:** Introduction to pointers, declaring pointer variables, Types of pointers, passing arguments to functions using pointers.

**Textbook: Chapter 13.1-13.6, 14 -14.7**

# CHAPTER 2 POINTERS

## 2.1 UNDERSTANDING THE COMPUTER'S MEMORY

- The computer's memory is a sequential collection of storage cells as shown in the below Fig.
- Each cell, commonly known as a byte, has a number called address associated with it.
- Typically, the addresses are numbered consecutively, starting from zero.
- The last address depends on the memory size.
- A computer system having 64 K memory will have its last address as 65,535.

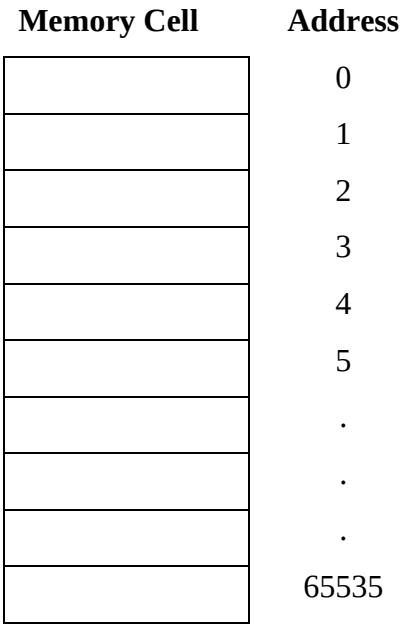



Fig : Memory Organisation

## 2.2 INTRODUCTION TO POINTERS

- A pointer is a derived data type in C.
- A pointer is variable that contains the memory location of another variable.
- Pointers contain memory addresses as their values.
- Every variable in C has a name and a value associated with it.
- When a variable is declared, a specific block of memory within the computer is allocated to hold the value of that variable.
- The size of the allocated block depends on the data type.
- Since a pointer is a variable, its value is also stored in the memory in another location. Suppose, we assign the address of quantity to a variable p.
- The link between the variables p and quantity can be visualized as shown in Fig. The address of p is 5048.

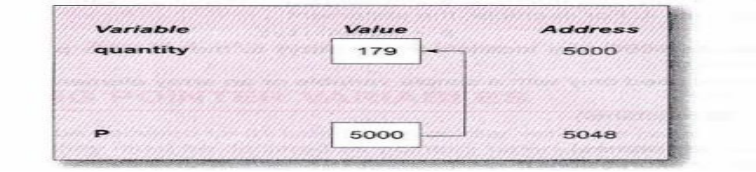



Fig : Pointer Variable

***Finding the size of various data types***

```
int main()
{
    printf("\n size of integer is %d bytes",sizeof(int));
    printf("\n size of floating point number is %d bytes",sizeof(float));
    printf("\n size of double is %d bytes",sizeof(double));
    printf("\n size of character is %d byte",sizeof(char));
    return 0;
}
```

***Output***

```
size of integer is 2 bytes
```

size of floating point number is 4 bytes

size of double is 8 bytes

size of character is 1 byte

Pointers are used frequently in C, as they offer a number of benefits (advantages) to the programmers.

They include (Advantages of pointers):

- Pointers are used to pass information back and forth between functions.
- Pointers enable the programmers to return multiple data items from a function via function arguments.
- Pointers provide an alternate way to access the individual elements of an array.
- Pointers are used to pass arrays and strings as function arguments.
- Pointers are used to create complex data structures, such as trees, linked lists, linked stacks, linked queues, and graphs.
- Pointers are used for the dynamic memory allocation of a variable.

Disadvantages of pointers

- If the pointers are not initialized properly, it causes segmentation fault.
- It is difficult to understand and debug
- It leads to memory leakage, if the pointers are not freed after usage in a dynamic memory management.

## 2.3 DECLARING POINTER VARIABLES

- The general syntax of declaring pointer variables can be given as below.

  **data_type *ptr_name;**

- Here, data_type is the data type of the value that the pointer will point to.
- For example,

  int *pnum;

  char *pch;

  float *pfnum;
- In each of the above statements, a pointer variable is declared to point to a variable of the specified data type.

- The declarations cause the compiler to allocate memory locations for the pointer variables p.
- Since the memory locations have not been assigned any values, these locations may contain some unknown values in them and therefore they point to unknown locations as shown:

  int *p ;


Contains garbage Points to unknown location

***Initialization Of Pointer Variables***

The process of assigning the address of a variable to a pointer variable is known as initialization. Once a pointer variable has been declared we can use the assignment operator **to initialize** the variable.

***Example:***

```
int x=10;
int *ptr;
ptr =&x;
```

We can also combine **the initialization with the declaration**. That is,

```
int *ptr = &x;
```

is allowed. The only requirement here is that the variable x must be declared before the initialization takes place.

- In the above statement, ptr is the name of the pointer variable.
- The * informs the compiler that ptr is a pointer variable and the int specifies that it will store the address of an integer variable.
- An integer pointer variable, therefore, 'points to' an integer variable.
- In the last statement, ptr is assigned the address of x.
- The & operator retrieves the lvalue (address) of x, and copies that to the contents of the pointer ptr.
- Now, since x is an integer variable, it will be allocated 2 bytes.
- Assuming that the compiler assigns it memory locations 1003 and 1004, the address of x (written as &x) is equal to 1003, that is the starting address of x in the memory.

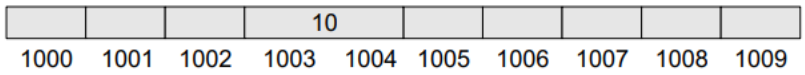



**Figure** Memory representation

- When we write, ptr = &x, then ptr = 1003.
- We can 'dereference' a pointer, i.e., we can refer to the value of the variable to which it points by using the unary * operator as in *ptr.
- That is, *ptr = 10, since 10 is the value of x.
- Look at the following code which shows the use of a pointer variable:

```
#include
int main()
{
        int num, *ptr;
        ptr = &num;
        printf("\n Enter the number : ");
        scanf("%d", &num);
        printf("\n The number that was entered is : %d", *ptr);
        printf("\n The address of number that was entered is : %d", ptr);
        return 0;
}
```

***Output***

```
Enter the number : 10
The number that was entered is : 10
The address of number that was entered is : 6487572
```

What will be the value of *(&num)?

It is equivalent to simply writing num.

We could also define a **pointer variable with an initial value of NULL or 0 (zero).** That is, the following statements are valid.

int *p = NULL;

int *p = 0;

**POINTER FLEXIBILITY** Pointers are flexible.

- We can make the same pointer to point to different data variables in different Statements.

  Example;

  int x, y, z, *p;

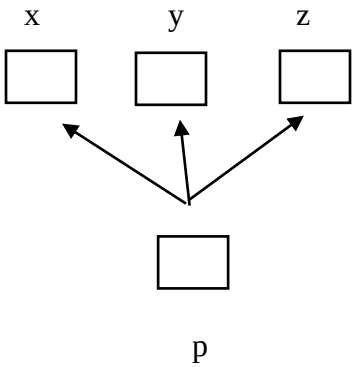
  p = &x;

  …

  p = &y;

  …

  p = &z;



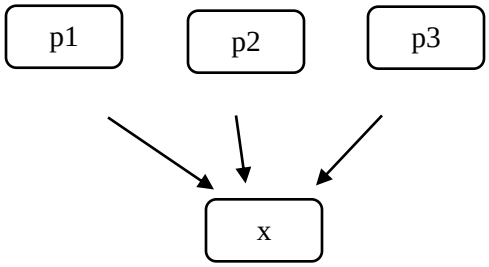

- We can also use different pointers to point to the same data variable. Example;

  int x;

  int *pl = &x;

  int *p2 = &x;

  int *p3 =&x;




With the exception of NULL and 0, no other constant value can be assigned to a pointer variable. For example, the following is wrong:

int *p = 5360; / *absolute address */

**Differences between normal variable and a pointer variable**

| **Normal variable** | **Pointer variable** |
|---|---|
| Stores values | Stores address |
| Not dereferenced to print the value through variable | Dereferenced with the help of * the indirection operator to print the value(content) through pointer |
| Dereferenced with the help of & the address of operator to print the address through variable | Not dereferenced to print the address through pointer |

**Program to add 2 numbers using pointers**

```
void main()
{
        int a,b,c,*p,*q;
        printf("enter 2 numbers");
        scanf("%d%d",&a,&b);
        p=&a;
        q=&b;
        c=*p + *q;
        printf("sum=%d",c);
}
```

## 2.4 TYPES OF POINTERS

***Null Pointers***

- A pointer variable is a pointer to a variable of some data type.
- However, in some cases, we may prefer to have a null pointer which is a special pointer value and does not point to any value.
- This means that a null pointer does not point to any valid memory address.
- To declare a null pointer, we may use the predefined constant NULL
- We can write int *ptr = NULL;
- We can always check whether a given pointer variable stores the address of some variable or contains NULL by writing,

```
if (ptr == NULL)
{
```

```
        Statement block;
    }
```

- We may also initialize a pointer as a null pointer by using the constant 0

  int *ptr;

  ptr = 0; This is a valid statement in C

***Generic Pointers***

- A generic pointer is a pointer variable that has void as its data type.
- The void pointer, or the generic pointer, is a special type of pointer that can point to variables of any data type.
- It is declared like a normal pointer variable but using the void keyword as the pointer's data type.
- For example, void *ptr;
- In C, since we cannot have a variable of type void, the void pointer will therefore not point to any data and, thus, cannot be dereferenced.
- We need to cast a void pointer to another kind of pointer before using it.
- Generic pointers are often used when you want a pointer to point to data of different types at different times.

```
#include<stdio.h>
int main()
{
        int x=10;
        char ch = 'A';
        void *gp;
        gp = &x;
        printf("\n Generic pointer points to the integer value = %d", *(int*)gp);
        gp = &ch;
        printf("\n Generic pointer now points to the character= %c", *(char*)gp);
        return 0;
}
```

***Output***

```
Generic pointer points to the integer value = 10
```

Generic pointer now points to the character = A

***Double Pointers (Chain of Pointers or Pointers to Pointers)***

Double pointers is used to make a pointer to point to another pointer, thus creating a chain of pointers.

```
void main()
{
        int a=10;
        int *p1,**p2;
        p1=&a;
        p2=&p1;
        printf("%d",**p2);
}
```

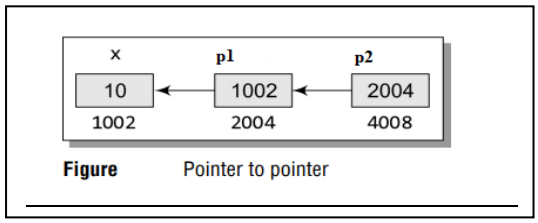



**Figure** Pointer to pointer

**Output**

**10**

Here p2 contains address of p1. This is also known as multiple indirections.

int **p2; tells the compiler that p2 is a pointer to a pointer of int type.

## 2.5 POINTER EXPRESSIONS AND POINTER ARITHMETIC

### POINTER EXPRESSIONS

Consider p1 and p2 are pointer variables:

1. Like any other variable, content at pointer can be used in expression.
   y=*p1 * *p2
2. Pointers can be compared: p1==p2, p1>p2 is valid
3. p1/p2, p1*p2, p1+p2 is illegal
4. p2-p1 is legal; if p1 and p2 are character pointers to same array. Then p2-p1 gives the number of number of elements between p1 and p2.
5. We can add or subtract integers; p1+5, p1-2
6. p1++, p1-- is legal.
7.

### POINTER ARITHMETIC (ADDRESS ARITHMETIC) POINTERS INCREMENT AND SCALE FACTOR

Let us consider starting address as 8000. Then the operation p++ is done to point to next subsequent element:

1. int

   p++ (i.e) p=p+1 address is 8002.
2. char

   p++ (i.e) p=p+1 address is 8001.
3. float

   p++ (i.e) p=p+1 address is 8004.
4. double

   p++ (i.e) p=p+1 address is 8008.

Let us consider starting address as 8008. Then the operation p-- is done to point to previous element:

1. int

   p-- (i.e) p=p-1 address is 8006.
2. char

   p-- (i.e) p=p-1 address is 8007.
3. float

   p-- (i.e) p=p-1 address is 8004.
4. double

   p-- (i.e) p=p-1 address is 8000.

Let us consider starting address as 8000. Then the operation p+5 is done to point to the fifth element from that element (i.e) element at index 5:

1. int

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(int))=8000+5*2

   address is 8010.
2. char

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(char))=8000+5*1

   address is 8005.
3. float

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(float))=8000+5*4

   address is 8020.

4. double

   p=p+5 (i.e) p=p+5 (starting address+index*sizeof(double))=8000+5*8

   address is 8040.

Let us consider starting address as 8040. Then the operation p-5 is done to point to the five elements before that element:

1. int

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(int))=8040-5*2

   address is 8030.

2. char

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(char))=8040-5*1

   address is 8035.

3. float

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(float))=8040-5*4

   address is 8020.

4. double

   p=p-5 (i.e) p=p-5 (starting address-index*sizeof(double))=8040-5*8

   address is 8000.

When we increment a pointer, its value is increased by the length of the data type that its points. This length is called scale factor.

- ➔ Scale factor for **int** in a 16-bit machine is **2 bytes** (i.e) the size of the datatype.
- ➔ Scale factor for **char** in a 16-bit machine is **1 byte** (i.e) the size of the datatype.
- ➔ Scale factor for **float** in a 16-bit machine is **4 bytes** (i.e) the size of the datatype.
- ➔ Scale factor for **double** in a 16-bit machine is **8 bytes** (i.e) the size of the datatype.

## 2.6 POINTERS AND ARRAYS

**int a[5]={10,20,30,40,50};**

Suppose the base address is 8000, each integer requires two bytes.

Here **a** refers to starting address. Also, &a[0] refers to the starting address.

**Initialization**

int a[5];

int *p;

p=a;

(or)

int a[5];

int *p;

p=&a[0];

Now, we can access the next element in the array by using p++.

p=&a[0] 8000

p+1=&a[1] 8002

p+2=&a[2] 8004

p+3=&a[3] 8006

p+4=&a[0] 8008

**Program**

```
void main()
{
    int a[10],n,I,*p;
    printf("enter n");
    scanf("%d",&n);
    printf("enter elements");
    for(i=0;i<n;i++)
        scanf("%d",&a[i]);
    p=a;
    printf("elements are");
    for(i=0;i<n;i++)
        printf("%d\t",*(p+i));
}
```

**OUTPUT**

enter n 5

enter elements 1 2 3 4 5

elements are 1 2 3 4 5

## 2.7 PASSING ARGUMENTS TO FUNCTION USING POINTERS

- When an array is passed to a function as an argument, only the address of the first element of the array is passed. It works like call by reference.

- Similarly, we can pass the address of a normal variable as an argument to function- It works like functions that return multiple values

In above both cases the parameters receiving the address should be pointers- so it works like call by reference. So changes in the formal parameters will affect the actual parameters.

- The function parameters are declared as pointers or arrays.
- Dereferenced pointers are used in function body
- When the function is called, the addresses are passed as actual arguments.

**Example : To swap two numbers**

```
void swap(int  *p,int  *q);
void main()
{
        int a,b;
        printf("enter two numbers");
        scanf("%d%d",&a,&b);
        printf("before swapping");
        printf("a=%d,b=%d",a,b);
        swap(&a,&b);
        printf("after swapping");
        printf("a=%d,b=%d",a,b);
}
void swap(int  *p,int  *q)
{
        int t;
        t=*p;
        *p=*q;
        *q=t;
}
```

**Output:**

```
enter two numbers 5 4
before swapping
a=5,b=4
```

after swapping

a=4,b=5

**program (bubble sort)**

```
#include<stdio.h>
void bubblesort(int a[20],int n);
void main()
{
	intn,a[20],i,j, temp;
	printf("enter the number of elements n");
	scanf("%d",&n);
	printf("enter the array elements");
	for(i=0;i<n;i++)
	{
		scanf("%d",&a[i]);
	}
	bubblesort(a,n)
	printf("\n the sorted elements are\n");
	for(i=0;i<n;i++)
	{
		printf ("%d\t",a[i]);
	}
}
void bubblesort(int a[20],int n)
{
	int  i,j;
	for(i=0;i<n-1;i++)
	{
		for(j=0;j<n-1-i;j++)
		{
			if(a[j]>a[j+1])
			{
```

```
                    temp=a[j];
                    a[j]=a[j+1];
                    a[j+1]=temp;
                }

            }
        }
```

**Output**

enter the number of elements n5

enter the array elements 50 20 40 10 30

the sorted elements are

10 20 30 40 50

## 2.8 Troubles with Pointers (Drawbacks of Pointers)

In most of the cases, compiler may not detect the error and produce unexpected results, when we use pointers. The output does not give us a clue regarding where we went wrong.

Debugging is difficult.

Some possible errors

- Assigning values to uninitialized pointers

```
int m=10,*p;
*p=m;
```

- Assigning a value to a pointer

```
int m=10,*p;
p=m;
```

- Not dereferencing to print the value

```
int m=10,*p;
p=&m;
printf("%d",p);
```

- Assigning the address of uninitialized variable

```
int m,*p;
p=&m;
```

- Comparing pointers that point to different objects

```
char name1[20],name2[20];
char *p1=name1;
char  *p2=name2;
if(p1>p2)
```

***Lab Program 11***

***Develop a program using pointers to compute the sum, mean and standard deviation of all elements stored in an array of N real numbers.***

**Algorithm**

**Step 1: [Initialize]**
Start

**Step 2:[Read the no of elements and array elements]**
Read n
Read a[]

**Step 3: [Set starting address of array to a pointer variable]**
ptr=a

**Step 4:[Iterate using a for loop to find sum using pointers]**
for(i=0;i<n;i++)
sum=sum+*ptr
ptr++
end for

**Step 5:[Calculate mean]**
mean=sum/n

**Step 6: [Set starting address of array to a pointer variable]**
ptr=a

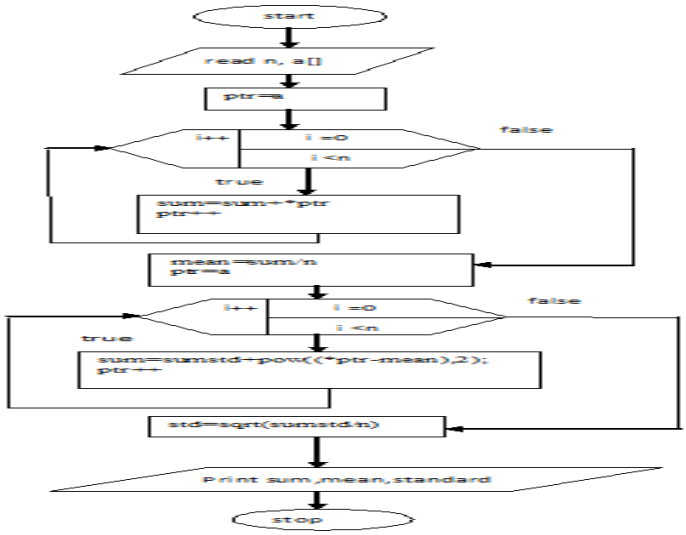
**Step 7:[Iterate using a for loop to find sumstd using pointers]**
for(i=0;i<n;i++
sumstd=sumstd+pow((*ptr-mean),2)
ptr++
end for

**Step 8:[Calculate standard deviation]**
std=sqrt(sumstd/n)

**Step 9:[Display the result]**
Print sum,mean,std

**Step 10:[Finished]**
Stop

**Flow Chart**




**Program**

```
#include<stdio.h>
#include<math.h>
int main()
{
        float a[10],*ptr,mean,std,sum=0,sumstd=0;
        int n,i;
        printf("\n Enter the number of elements");
        scanf("%d",&n);
        printf('\n Enter the array elements");
        for(i=0;i<n;i++)
        {
                scanf("%f",&a[i]);
        }
        ptr=a;
        for(i=0;i<n;i++)
        {
                sum=sum+*ptr;
                ptr++;
        }
        mean=sum/n;
        ptr=a;
        for(i=0;i<n;i++)
        {
                sumstd=sumstd+pow((*ptr-mean),2);
                ptr++;
        }
        std=sqrt(sumstd/n);
        printf("Sum=%f\n",sum);
        printf("Mean=%f\n",mean);
        printf("Standard Deviation=%f\n",std);
```

```
    return 0;
}
```

**Test cases**

| Test No | Input Parameters | Expected Output | Obtained Output |
|---|---|---|---|
| 1 | Enter the number of elements 5<br>Enter the array elements 1 5 9 6 7 | Sum= 28<br>Mean= 5.6<br>Standard Deviation= 2.09 | Sum= 28<br>Mean= 5.6<br>Standard Deviation= 2.09 |
| 2 | Enter the number of elements 4<br>Enter the array elements 2.3 1.1 4.5 2.78 | Sum= 10.68<br>Mean= 2.67<br>Standard Deviation= 0.863 | Sum= 10.68<br>Mean= 2.67<br>Standard Deviation= 0.863 |