## Pointers:-

A pointer is a variable that holds a memory address. This address is the location of another object (typically another variable) in memory. For example, if one variable contains the address of another variable, the first variable is said to point to the second.

If a variable is going to be a pointer, it must be declared as such. A pointer declaration consists of a base type, an *, and the variable name. The general form for declaring a pointer variable is

*datatype *variable-name;*

Where type is the base type of the pointer and may be any valid type. The name of the pointer variable is specified by name.

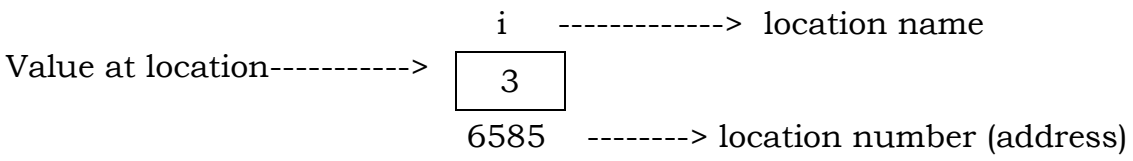
Consider the declaration,

int i=3;

This declaration tells the compiler to:

(a) Reserve space in memory to hold the integer value.
(b) Associates the name with this memory location.
(c) Store the value 3 at this location.

We may represent i's location in the memory as




The operator & is called as 'address of' or 'referencing operator'. The expression &i returns the address of the variable i.

Another pointer operator available in C is ***** is called 'value at address' or 'dereference' or 'indirection' operator. It returns the value at address is also called an indirection operator.

```
/*example program */
main()
{
int i=3;
printf("address of i=%u", &i );
```

```
printf("value of i=%d",i);
printf("value of i=%d",*(&i));
}
```

Output:-

address of i=6855

value of i=3

value of i=3

**Features of pointers:**

- Pointers are of unsigned integer data type, as they are storing addresses and always addresses are unsigned integer, that's why %c used to print addresses.
- Before using a pointer it should be initialized, other wise it contains some garbage address which points to some where that's why an uninitialized pointer is also called as a wild pointer.
- Pointer size is 2 bytes irrespective of the data type, it occupies 2 bytes only as they are holding same unsigned integer called address
- Always the data type of a pointer and the address of the variable it is going to hold should be the same.

  ```
  int a,*b,   float c,*d;
  b=&a,       d=&c
  ```
- A null pointer is a pointer which pointes to nowhere.

  ```
  char *c=null;
  ```
- Array of pointer

  data type *array-name[size]

  int*a[10];
- Pointer to an array

  data type (*array-name)[size];

  int(*a)[5];

## Advantages of pointers:-

→ Makes program works faster as they are directly dealing with addresses
→ Array can be easily accessed using pointers
→ Call by reference can be achieved
→ Dynamic memory allocation can be achieved
→ Complex declarations are possible in pointers.

**Pointer Expression:-**

In the previous example, the expression &i returns the address of i. If we so desire, this address can be collected in a variable by saying,

j=&i;

Here j is not an ordinary variable like any other integer variable. It is a variable, which contains the address of another variable.


We must declare j before using it. It is declared as

int *j;

This declaration tells the compiler that j will be used to store the address of an integer value in other words j points to an integer.

```
/*example program */
main()
{
int i=3;
int *j;
j=&i;
printf("address of i=%u", &i);
printf("address of i=%u",j);
printf("address of j=%u",&j);
printf("value of j=%d",j);
printf("value of i=%d",i);
printf("value of i= %d",*(&i));
printf("value of i=%d", *j);
}
```

Look at the following declarations:

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

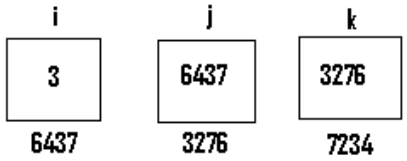
Here alpha, ch and s are declared as pointer variables i.e. variables capable of storing (holding) addresses. That addresses are always going to be whole numbers. Therefore pointers always contain whole numbers. The declaration float *s does not mean that it is going to contain a floating-point value. What it means is, s is going to contain a floating point values address.

Similarly char *ch means that ch is going to contain the address of a char value.

Pointer we know is a variable, which contains address of another variable. Now this variable it self could be another pointer. Thus we now have a pointer, which contains another pointer's address.

```
/*Example Program */
main()
{
        int i=3;
        int *j;
        int **k;
        j=&i;
        k=&j;
        printf("address of i=%u",&i);
        printf("address of i=%u",j);
        printf("address of i=%u",*k);
        printf("address of j=%u",&j);
        printf("address of j=%u",k);
        printf("address of k=%u",&k);
        printf("value of j=%u",j);
        printf("value of k=%u",k);
        printf("value of i=%d",i);
        printf("value of i=%d",*(&i));
        printf("value of i=%d",*j);
        printf("value of i=%d",**k);
}
o/p:-
        address of i=6485
        address of i=6485
        address of i=6485
        address of j=3276
```

address of j=3276
address of k=7234
value of j=6485
value of k=3276
value of k=3276
value of i=3
value of i=3
value of i=3
value of i=3




**Pointer arithmetic:-**

Pointer arithmetic offers a restricted set of arithmetic operators for manipulating the address in pointers.

Pointers and one dimensional arrays:-

If we have an array, *a*, then '*a*' is a constant pointing to the first element and *a*+1 is a constant pointing to the second element of an array. Again, if we have a pointer, p, pointing to the second element of an array, then p-1 is a pointer to the previous (first) element and p+1 is a pointer to the next (third) element. Given *a*, *a*+2 is the address two elements from *a*, and *a*+3 is the address three elements from *a*.

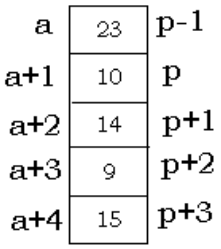



Fig: Pointer arithmetic

But, the meaning at adding or subtracting here is different from normal arithmetic. We can make the access more efficient than the corresponding index notation by using the following calculation.

*Address = pointer +(offset * size of element)*

Ex:- a+n*(size of (one element)

A pointer when incremented always points to an immediately next location of its type. A pointer when decremented always points to the previous location of its type.

**Arithmetic operations on pointers:-**

Arithmetic operations involving pointers are very limited. *Addition* can be used when one operand is a pointer and the other is an integer. *Subtraction* can be used only when both operands are pointers or when the first operand is a pointer and the second operands is an index integer. We can also manipulate a pointer with the postfix and unary increment and decrement operations. All of the following pointer arithmetic operations are valid:

P+5, 5+p, p-5, p1-p2, p++, --p

When one pointer is subtracted from another, the result is an index representing the number of elements between the two pointers**.** However that the result is meaningful only if the two pointers are associated with the same array structure.

The relational operators (<,>, etc) are allowed only if both operands are pointers of the same type.

p1 >= p2 p1! =p2

The most common comparison is a pointer and the null constant,

| Long form | short form |
|---|---|
| if (ptr==null) | if(!ptr) |
| if (ptr!=null) | if(ptr) |

Ex:-/* Print an array forward by adding to a pointer. Then print it backward by subtracting one from a pointer*/

```
#include <stdio.h>
#define max_size 10
int main()
{
    //local declarations
```

```
int ary[20]={1,2,3,4,5,6,7,8,9,10};
int *p1, *p2;
        // print array forward
printf("array forward:\n");
for(p1=ary,p2=ary+max_size; p1<p2; p1++)
printf("%3d",*p1);

        //print array backward
printf("array backward:\n");
for(p1=p2-1; p1>=ary; p1--)
printf("%3d",*p1);
return 0;
}
```

## Pointer to an array:-

Declaration of pointer to an array:-

*data type (*array-name)[size];*

Ex: int (*p)[5];

Where size represents the column size, the space for rows may be dynamically allocated.

/* write a c program that uses pointer to an array */

```
#included<stdio.h>
#include<conio.h>
main()
{
int b[3][2]={{2,3},{7,9},{0,12}};
int (*a)[2];
int i,j,*ptr;
clrsscr();
printf("array elements are \n");
for (i=0;i<3;i++)
{
```

```
a=&b[i];
ptr=(int*)a;
printf("\n");
for (j=o;j<2;j++)
printf("%d",*(ptr+j);
}
getch();
return0;
}
```

Out put:

```
Array elements are:
2   3
7  9
0 12
```

## **Array of Pointers:-**

Declaration:

datatype *array-name[size];

Ex: int *p[5];

There can be an array of integers, an array of floats; similarly, there can be an array of pointers. Since a pointer variable contains an address, an array of pointers would be nothing but collection of addresses.

/*C program that uses array of pointers */

```
#include <stdio.h>
main()
{
    int  *p[4];
    int   a= 3,b=4,c=5,d=6,i;
    clrscr();
    p[o]=&a;
    p[1]=&b;
    p[2]=&c;
    p[3]=&d;
   for(i=0; i<4; i++)
      printf( "%d", *p[i] );
   getch();
```

```
    return 0;
}
```

## Pointer and 2-d arrays:-

Row of 2-d arrays can be thought of as a one-dimensional array.

```
int p[3][2];
```

In the above declaration array p is a two-dimension array of 3 rows and 2 columns. The element p[0] gives the address of the 1st dimensional array. p[1] gives the address of the 2nd dimensional array.

/* write a c program to print 2-d array by using pointers */

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int p[3][2]={{0,2},{1,7},{8,9}};
        int i,j;
        clrscr();
        for(i=0;i<3;i++)
        {
                for(j=0;j<2;j++)
                {
                printf("%d",*(*(p+i)+j));
                }
        }
        getch();
}
```

o/p:-

0 2 1 7 8 9

Note :- p[2][1]

```
        => *(p[2]+1)
        => *(p*(p+2)+1)
```

## Pointers and functions:-

A particularly confusing yet powerful feature of C is the function pointer. A function has a physical location in memory that can be assigned to a pointer.

This address is the entry point of the function and it is the address used when the function is called. Once a pointer points to a function, the function can be called through that pointer. Function pointers also allow functions to be passed as arguments to other functions.

We obtain the address of a function by using the function's name without any parenthesis or arguments.

```
/* write a program to call function by using pointers*/
# include <stdio.h>
# include <conio.h>
int display();
void main()
{
    int(*p) ();
    clrscr();
    p=display;
    (*p)();
    printf("display() functions address is \n");
    printf("%u", display );
    getch();
}
 int display()
 {
    printf (" demo of pointers to functions");
 }
```

Output:

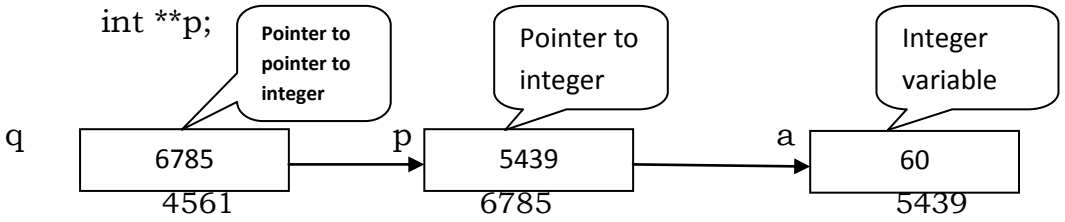
```
demo of pointers to function
display() functions address:   6305
```

## Pointers to pointers:-

So far, all of our pointers have been pointing directly to date. It is possible to use pointers that pointers that point to other pointers. For ex we can have a pointer pointing to a pointer to an integer. This two level indirection is shown below. There is no limit as to how many levels of indirection we can use but, practically we can use more than two.

```
//local declarations
```

```
int a;
int *p;
int **p;
```




Each level of pointer indirection requires a separate indirection operator when it is dereferenced. In the above example to refer to a using pointer p. we have to deference it once. As shown below,

*p

To refer to the variable a using pointer q, we have to deference it twice to get the integer a, because there are two levels of indirection (pointers) involved. Another way to say this is that q is a pointer to a pointer to an integer. The double deference is shown below:

**q;

The following example shows how we can use different pointers with pointers to pointers and pointers to read the value of the same variable. A graphic representation of the variable is shown below.

Ex:-

```
#include<stdio.h>
int main(void)
{
        int a;
        int *p;
        int **q;
        int ***r;
        p=&a;
        q=&p;
        r=&q;
        printf("enter a number");
        scanf("%d",&a);
        printf("the no.is %s",a);          //using a
        printf("enter a number ");
        scanf("%d",p);
        printf("the no.is %d",a);          //using p
        printf("enter a number");          //using q
        scanf("%d",*q);
```

```
    printf("the no is %d",a);
    printf("enter a number");
    scanf("%d",**r);
    printf("the no is %d ", a);        //using r
    return 0;
}
```

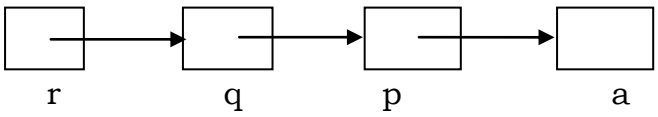



Fig: pointers to pointers

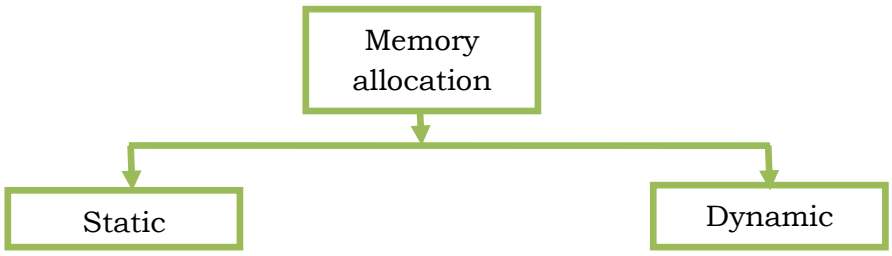
Note: A single pointer can store the address of the variable. A double pointer can store the address of single pointer. A triple pointer can store the address of double pointer.

**Void Pointer:** A void pointer is also called as a generic pointer which can store the address of any variable.

```
Ex:- void *ptr;
     int a, char c, float b;
     ptr=&a
     * (int *) ptr = 10; ptr = &b; *(float *) ptr = 2.4
      ptr = &c,* (char *) ptr= 'a'
     printf ("%d %d %d", a, b, c);
```

**Memory management:-**

C give us two choices when we what to receive memory locations for an object static allocation and dynamic allocation.




**Memory Usage:-**

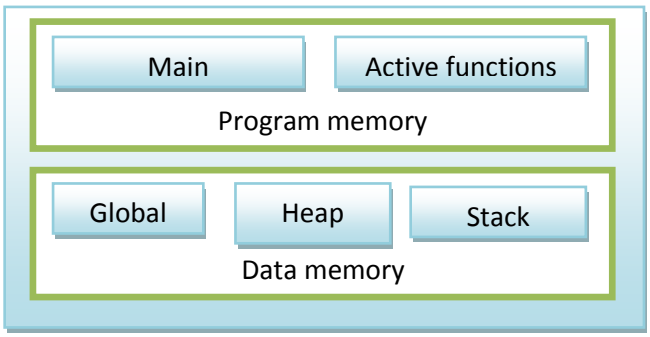
To understand how dynamic memory allocation works, we must study how memory is used. Conceptually memory is divided into program memory and data memory. Program memory consists of the memory used for main and

all called functions. Data memory consists of permanent definitions, such as global data and constants, local declarations and dynamic data memory.

Obviously main must be in memory at all times. Beyond main, each called function mush be in memory only while it or any off its called functions are active as a practical matter, most systems keep all functions in memory while the program is running.

Although the program code for a function may be in memory at all times, the local variables for the functions are available only when it is active. Further more, more then one version of the function can be active at a time. In this case, multiple copies of the local variables are allocated, although only one copy of the function is present. The memory facility for these capabilities is known as the *stack memory*.

In addition to stack a memory allocation know as the *heap* is available. Heap memory is unused memory allocated to the program and available to be assigned during its execution. It is the memory pool from which memory is allocated when requested by the memory allocation functions. This conceptual view of memory is shown in fig below.




**Static memory allocation:**

Static memory allocation requires that the declaration and definition of memory be fully specified in the source program the number of byes reserved cannot be changed during runtime. This is the technique we have used to this point to define variables, pointers, arrays and streams.

**Dynamic memory allocation:**

Dynamic memory allocation uses predefined functions to allocate and release memory for data while the program is running. It affectively postpones the data definitions, but not the data declaration, to run time.

We can refer to memory allocated in the heap only through a pointer. To use dynamic memory allocation, we use either standard data type or derived

type that we have previously declared. Unlike static memory allocation, dynamic memory allocation has no identifier associated with it. It has only an address that must be used to access it. To access data in dynamic memory therefore we must use pointer.

## Memory allocation functions:-

Four memory management functions are used with dynamic memory. Three of them **malloc**, **calloc** and **realloc** are used for memory allocation. The fourth, **free** is used to return memory when it is no longer needed. All the memory management functions are found in the standard library file (stdlib.h). The collection of memory functions is shown in fig below.

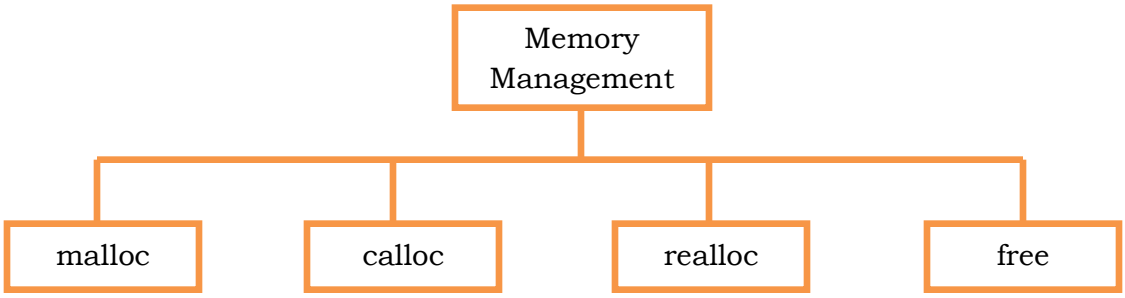



Fig: Memory management functions

### Block memory allocation (malloc):-

The malloc function allocates a block of memory that contains the number of bytes specified in its parameter. It returns a void pointer to the first byte of the allocated memory. The allocated memory is not initialized. We should there fore assume that it will contain unknown values and initialize it as required by our program.

The malloc function declaration is shown below.

void *malloc (size_t size);

The type size_t is defined in several header files including stdio.h. The type is usually an unsigned integer, and by the standard it is guaranteed to be large enough to hold the maximum address of the computer.

For example, if we want to allocate an integer in the heap, we code the call as shown below:

P=malloc (size of (int));

As mentioned above, malloc returns the address of the first byte in the memory space allocated however if it is not successful, malloc returns a null pointer. An attempt to allocate memory from the heap when memory is insufficient is known as overflow. It is up to the program to check for memory overflow.

**Contiguous memory allocation (calloc):-**

Declaration of calloc:

*void *calloc(size_t num_elements, size_t element_size};*

malloc does not initialize memory (to *zero*) in any way. If you wish to initialize memory then use calloc. Calloc there is slightly more computationally expensive but, occasionally, more convenient than malloc. Also note the different syntax between calloc and malloc in that calloc takes the number of desired elements, num_elements, and element_size, element_size, as two individual arguments.

Thus to assign 100 integer elements that are all initially zero you would do:

```
int *ip;
ip = (int *) calloc(100, sizeof(int));
```

**Reallocation of memory (realloc):-**

The realloc function can be highly inefficient and therefore should be used advisedly. When given a pointer to a previously allocated block of memory. realloc changes the size of the block by deleting or extending the memory at the end of the block. If the memory cannot be extended because of other allocations, realloc allocates a completely new block, copies the existing memory allocation to the new allocation, and deletes the old allocation. The programmer must ensure that may other pointer to the data are correctly changed.

Void *realloc (void*ptr,size_t newsize);

**Releasing memory (free):-**

**Void free(void *ptr);**

When memory allocations allocated by malloc calloc or realloc are no longer needed, they should be freed using the predefined function free.

Releasing memory does not change the value on a pointer. It still contains the address in the heap. The program may continue to run, but the

data may be destroyed if the memory area is allocated for another use. Using a pointer after its memory has been released is a common programming error.

The pointer used to free memory must be of the same type as the pointer used to allocate the memory.

**Command line arguments: - (argc, argv)**

All the programs we have written are coded with no parameters. But main is a function, and as a function, it may have parameters. When main has parameters, they are known as command line arguments.

Command line arguments are parameters to main when the program starts. They allow the user to specify additional information when the program is involved. For instance, if we write a program to append row files, rather than specify the names of the files as constants in the code, the user could supply them when the program starts.

int main(int argc,char *argv[])

# Strings:-

Definition: - A string is a collection of characters and ends with a null character '\0' which represents the end of the string.

## Features of strings:

- A string is also called as character array.
- All the string functions are defined in "string.h" header file.
- Each character in the array occupies one byte of memory and the last character is always '\0'.

String declaration format:

*char array-name[size];*

Ex:

```
char s[20];
char st[10];
```

Initialization of strings can be done in four ways:

```
char str[7]={"abcdef"};
char str[7]={'a','b','c','d','e',f','\0'};
char str[]={'a','b','c','d','e','f','\0'};
char str[7]="abcdef";
```

Storage in memory:

```
char str[50]="this is demo";
```

| T | H | I | S | | I | S | | D | E | M | O | \0 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 161 | 162 | 163 | 164 | 165 | 166 | 167 | 168 | 169 | 170 | 171 | 172 | 173 | 174 |

The terminating null ('\0') is important because it is the only way the functions that work with a string can know where the string ends.

## Inputting strings:

-Reading of string – by using gets () and scanf ()

Gets()- ready a string with spaces

Scanf("%s",&array[ ]); - reads a string without spaces

Puts( ) – print the string.

- char a[ ]="hi" \\ character array

- char *s = “hi” \\pointer points to the string

//Program to demonstrate the printing of string.

```
# include<stdio.h>
main()
{
    int i ;
    char a[ ]={”demo string”};
    clrscr();
    i =0;
    while (a[i]!=’\0’)
    {
        printf(“%c”,a[ i]);
        i++;
    }
    return 0;
}
```

// Program to demonstrate the printing of string.

```
# include<stdio.h>

main()
{
        char a[30];
        clrscr();
        printf(“enter a string”);
        gets(a);
        printf(“entered string is \n”);
        puts(a);
        getch();
        return 0;
    }
```

## String handling functions:-

- strlen ():- This function returns the length of the string.
  Ex:-strlen (“SRES”)=4
- strcpy ():- It copies one string to another.
  Ex:- strcpy(str1,str2);
  It copies str2 to string str1.
- strcat():- Appends one string at the end of another string

Ex:-strcat(s1,s2)=”hello world”
S1=”hello”, S2=”world”

- strcmp():-Compares two strings and returns.
  0 – if both strings are exactly equal.
  Positive value – if first string is > second string
  Negative value – if first string is < second string.

- strrev():- Which reverses a string.
  Ex:- char str[ ]=”good”;
  Strrev (str);
  Printf(“%s”,str);
  o/p:-doog
- strupr():- which converts a string to uppercase.
- strlwr():- which converts a string to lowercase.
- strdup():- duplicates a string.
- strchr():- finds first occurrence of a given character in a string.
- strrchr():- finds last occurrence of a given character in a string.
- strstr():- finds first occurrence of a given string in another string.

Most used functions are strlen(), strcpy(), strcat() and strcmp().

**Strlen():-**

This function counts the number of characters in a string.

```
#include<stdio.h>
#include<string.h>
int main()
{
    char str[20]={“i love india”};
    int lengh1,length2;
    length1=strlen(str);
    length2=strlen(“india”);
    printf(“string=%s, length=%d”,str,length);
    printf(“string=%s,length=%d”,”india”,length2);
    getch();
    return 0;
}
```

**Strcpy():-**

This function copies the contents of one string into another. The base address of the source and target strings should be supplied to this function.

```
#include<string.h>
#include<stdio.h>
int main()
{
   char a[20]={“pavithran”};
   char b[ ];
   strcpy (b,a)
   printf(“main string=%s”,a);
   printf(“copied string=%s”,b);
   getch();
   return 0;
}
```

Output:

main string= pavithran
copied string= pavithran

**Strcat():-**

This function appends the source string at the end of the target string.

```
Ex:-   #include<string.h>
        #include<stdio.h>
        void main()
    {
      char p[10]=”great”;
      char q[20]=”india”;
      strcat(p,q);
      printf(“concentrated string is %s”,p);
      getch();
    }
```

**Strcmp():-**

This function compares two strings to find out whether they are identical or different. The two strings are compared character by character until there is a mismatch or end of the one of the strings is reached, whichever occurs first.

If the two strings are identical, strcmp() returns zero.

```
 #include<string.h>
 #include<stdio.h>
main()
 {
   char  a[ ]={“pavithran “};
   char b[ ]={“pavithran”};
   if(strcmp(a,b)==0)
     printf(“strings are equal”);
   else
   printf(“strings are not equal”);
   getch();
 }
```

## Two dimensional array of characters:-

```
 #include<string.h>
int main()
{
  char mainl;ist [5][10]={ “radha”, ”kalpana”, ”ooha”, ”veni”, ”sudha” };
  int i, flag, a;
  char name[10];
  clrscr();
  printf(“enter your name”);
  gets(name);
  flag=0;
  for(“i=0;i<5;i++)
 {
   a=atrcmp(&mainlist[i][0],name);
   if(a==0)
    {
       printf(“name found”);
       flag=1;
      break;
    }
   }
  if (flag==0)
    printf(“name not found\n”);
    return 0;
}
```

**Character operations:-**

isalpha:- if argument is a letter of the alphabet.

isdigit:- if argument is one of the ten decimal digits.

islower:- if argument is a lower case letter of the alphabet.

isupper:- if argument is a upper case letter of the alphabet.

ispunct:- if argument is a punctuation character that is a non control character that is not a space ,a letter of the alphabet, or a digit

isspace:- if argument is a white space character such as a space, a newline, or a tab.

tolower:- it converts its lowercase letter argument.

toupper:-to the uppercase equivalent and returns this equivalent as the value of the call.