# CHAPTER 15 POINTERS

## 15.1 INTRODUCTION

C has a rich set of data types and operators. Pointer is an example of derived data type in C. A pointer is a variable which holds address of another variable/object as its content. The memory required for that variable/object can be allocated dynamically using a memory allocation function such as malloc. The object may be any C data type such as integer, character, string, or structure. Pointers are frequently used in C programming. Pointers are one of the most distinct and exciting features of C language. It provides power and flexibility to the language. Although pointer may appear little confusing and complicated in the beginning, but it is a powerful tool and handy to use once it is mastered.

**Advantages of pointers in C:**

- ❖ Pointers are closer to the hardware. They provide direct access to physical memory.
- ❖ A pointer returns more than one value from the functions.
- ❖ Reduces the storage space and complexity of the program.
- ❖ Reduces the execution time of the program.
- ❖ Provides an alternate way to access array elements.
- ❖ Pointers can be used to pass information back and forth between the calling function and called function.
- ❖ Pointers allow us to perform dynamic memory allocation and deallocation.
- ❖ Pointers helps to build complex data structures like linked list, stack, queues, trees, graphs etc.
- ❖ Pointers allow us to resize the dynamically allocated memory block.
- ❖ Addresses of objects can be extracted using pointers.

**Drawbacks of pointers in C:**

- ❖ Uninitialized pointers might cause segmentation fault.
- ❖ Dynamically allocated block needs to be freed explicitly. Otherwise, it would lead to memory leak.
- ❖ If pointers are updated with incorrect values, it might lead to memory corruption.

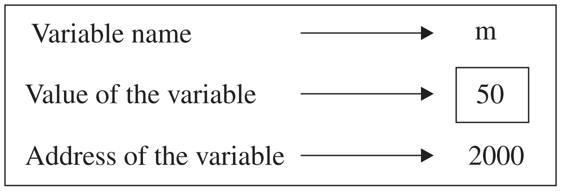
## 15.2 CONCEPT OF POINTER

In C program, when a variable is declared, the C compiler allocates proper bytes of memory to store the value of that variable. In a computer memory, every memory has a unique address.

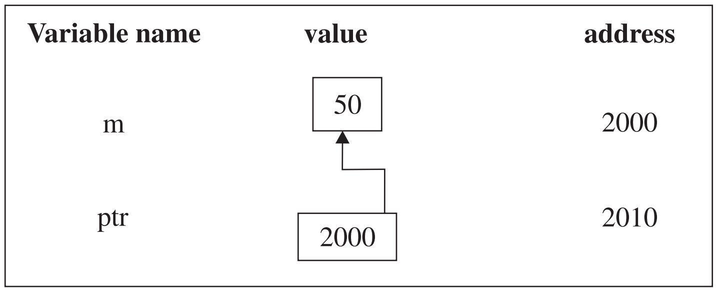
**Example**

```
int m = 50;
```

Here ‘m’ is an integer which requires a 2 byte memory for its storage. Let us assume the system has chosen the address location 2000 for m. It stores the value 50 in location 2000.



The value 50 which is represented by m can be accessed by either using the variable name m or the address 2000. Since the memory addresses unique numbers, they can be assigned to some other variable. The variable that holds memory address is called **pointer variables**. A **pointer** variable is therefore a variable that contains an address which is a location of another variable. Value of **pointer variable** will be stored in another memory location.



**Pointer variable**

The value of the variable ptr is the address of m. The value 50 can be referred either by m or through pointer variable ptr. The variable ptr points to m. Thus ptr gets the name pointer.

### Accessing the Address of a Variable

While executing the declaration statement, C compiler allocates a memory for a variable which is system dependent and is not known for the users. But the allocated address is extracted through the use of pointers. This can be done with the help of ampersand (&) operator in C. Any variable proceeds with the ‘&’ symbol will return the address of the variable.

**Example**

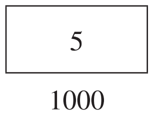
```
P = &m; // now P will return the address of m.
```

Another pointer operator available in C is '*' called "value at operator". It gives the value stored at a particular address. This operator is also called "indirection operator".

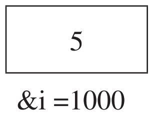
**Example**

```
main()
{
int i = 5;
printf("\n Address of i : = %u",&i);     // Returns the address
printf("\n value i: = %d",*(&i));        // Returns the value at address of i
}
```

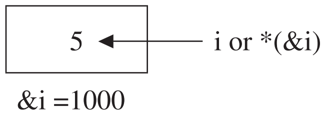
***Explanation:*** While executing the declaration statement int i = 5, the compiler will allocate a 2 byte memory for i and stores the 5 in that location. Let us assume the location is 1000 & 1001.



&i will return the address 1000.



*(&i) will return the value of i at this address, i.e., 5



The & operator can be used only with a simple variable or an array element. The following are the illegal use of address operator:

1. &100 // pointing at constant
2. int x[5];
   &x; // pointing at array name
3. &(x+y) // pointing at expression.

If x is an array then &x[2]+&x[8] are valid expressions which points to 3rd and 7th elements of x.

**Write a program to display the value of the variable along with the address.**

**PROGRAM**

```
main()
{
      char a;
      int m ;
      float x;
      a = 'Z';
      m = 100;
      x = 10.25;
printf("\n %c is stored at address %u \n",a,&a;);
printf("\n %d is stored at address %u \n",m,&m;);
printf("\n %f is stored at address %u \n",x,&x;);
}
```

**Output**

```
Z is stored at address 4450
100 is stored at address 4454
10.25000 is stored at address 4460
```

## 15.3 DECLARING POINTER VARIABLE

Like other C variables, pointer variable also needs to be declared before its first usage, since pointer variable holds the address of another variable of the same type. Pointer variable declaration is as follows:

**data_type *ptr_name;**

The above statement provides the following information to the compiler:

1. The asterisk (*) tells that variable ptr_name is a pointer variable.
2. ptr_name needs memory location, the content of that location holds the address of another variable.

**Example**

int *p; // declaration of a pointer variable p

Here p is a pointer variable which holds an address of another integer variable as its content. Up to this, only declaration part of pointer variable is created. It does not have any value, it means that we haven't store any address in the above declaration. This process can be done by the following initialing statement.

## 15.4 INITIALIZING POINTER VARIABLE

Without assignment, pointer variables are not useful. The process of assigning the address of a variable to a pointer variable is known as initialization of pointer variable. While assigning address of a variable to a pointer variable, we should be careful enough in such a way that, address of a variable of any data type will be assigned to a pointer variable of the same type. Example, address of an integer variable will be assigned to an integer pointer, and address of a floating point variable will be assigned to a floating pointer and so on.

Pointer and normal variable declaration statements:

int i, j, *p; // integer variables and integer pointer declaration

float x, y, *q; // float variables and float pointer declaration
char c, d, *r; // char variables and char pointer declaration

### Initializing pointer variables

p = &i; // valid initialization. Integer variable address is assigned to integer pointer variable
q = & x; // valid initialization. float variable address is assigned to float pointer variable
r = &c; // valid initialization. char variable address assigned to char pointer variable

Thus a pointer variable can be assigned the address of an ordinary variable or it can be assigned the value of other pointer variables provided both pointer variables are of the same type.

int i,*a,*b;
a = &i; // the address of int i is assigned to int pointer a;
b = a; // the value of pointer a (address of i) is assigned to int pointer b.

In the last example, both the pointers (a and b) belongs to same data type of int. A pointer can also be initialized to NULL value. In ANSI C, the value of NULL = 0.

int *ptr = NULL

## 15.5 GENERIC POINTERS

There is an exception in ANSI C for pointer assignment. It supports generic pointer creation. Instead of creating pointer variable of one type and assigning address of a variable of same type to pointers, we can create generic pointers. Generic pointers are pointer variables which do not depend on any specific data type. So it can hold address of any variable as its reference. The data type belongs to generic pointer is void.

**Example**

```
void *Gp; // Gp is a generic pointer with data type as void.
int a;
float b;
char c;
Gp = &a; // Gp hold the address of int variable a;
------
Gp = &b; // after some time Gp is reinitialized to hold the address of float variable b.
------
Gp = &c; // after some time Gp is reinitialized to hold the address of char variable c.
```

All are valid assignments.

## 15.6 ACCESSING A VARIABLE THROUGH ITS POINTER

Once pointer has been created and successfully assigned an address of a variable, the next step is to access the variable through its pointers. Accessing the variable through pointer is fast, because pointers are closer to the hardware. Accessing the variable can be done using another operator '*' (asterisk) called indirection operator. Another name for this operator is dereferencing operator.

**Example**

```
int qms, *a, b ; // declaration of qms & b as normal int variables , a as pointer to an int.
qms = 200; // assigning the value 200 to qms
a = &qms; // the address of qms is assigned to an int pointer a;
b = *a; // content of the variable a is assigned to b;
```

When the operator * is proceeded with a pointer variable, it returns the value of the address location it holds. In this example, initially qms is assigned a numeric value 200 using assignment statement. Then, the address of qms is assigned to a pointer variable a. Now the representation of *a will return actual value stored in the address location of qms. This value will be assigned to a pointer b; so indirectly the value 200 is assigned to b;

### Example program using pointers

```
#include<stdio.h>
void main()
{
      int balance;
      int &addr;
      int val;
      balance = 5000;
      addr = &balance;
      val = *addr;
      printf("\n Balance is = % d",val);
}
```

**Output**

```
Balance = 5000
```

### Simple program illustrating pointers

```
#include<stdio.h>
void main()
{
int i = 5;
int k = 10;
int *ptr = &k;
printf("\n i has the value %d and is stored at %u ", i, &i);
printf("\n k has the value %d and is stored at %u ", k, &k);
printf("\n ptr has the value %u and is stored at %u ", ptr, &ptr);
printf("\n The value of the integer pointed by ptr is %d ", *ptr);
}
```

**Output**

```
i has the value 5 and is stored at 4002
k has the value 10 and is stored at 4004
ptr has the value 4004 and is stored at 4008
The value of the integer pointed by ptr is 10
```

## 15.7 CHARACTER POINTER

Text strings are represented in C by arrays of characters. Since arrays are very often manipulated via pointers, character pointers are probably the most common pointers in C. Strings are actually one dimensional array of characters terminated by a **null** character '\0'. Thus, a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greet[6] = { 'H' , 'e', 'l', 'l', ' o ' ,'\0'};

### Example program to demonstrate pointer to character string

```
#include<stdio.h>
#include<stdlib.h>
void main()
{
int i;
char *cp;
cp = " A String" ;
    while( *cp != 0)
    {
       putchar(*cp);
       cp++;
    }
}
```

**Output**

```
A String
```

In the above example, the loop sets a pointer to the start of the string and then walks along the string until it find the zero at the end. In C, each string is terminated by a NULL to represent the end of the string. The number of iterations of the loop depends on the length of the string. The zero at the end represents the end of the string.

## 15.8 POINTER EXPRESSIONS

C language allows us to perform arithmetic operations on pointers. The arithmetic operations on pointers can affect the memory location pointed by them or the data stored at the pointed location. Like other variables, pointer variables are also used in expressions. The expressions involving pointers have the same rule as other expressions. The following are few examples of pointer expressions.

If ptr1 and ptr2 are properly declared and assigned pointers, then the following statements are valid:

x = *ptr1 * *ptr2; // same as (*ptr1) * (*ptr2)

total = total + *ptr1;

y = 10 * - *ptr2 / *ptr1; // same as ( 5 * ( -(*ptr2)))/(*ptr1)
*ptr2 = *ptr2 + 10;

## Example Program

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i=3,*x;
    float j=1.5,*y;
    char k='c',*z;
    printf("\nValue of i = %d",i);
    printf("\nValue of j = %f",j);
    printf("\nValue of k = %c",k);
    x=&i;
    y=&j;
    z=&k;
    printf("\nOriginal address in x = %u",x);
    printf("\nOriginal address in y = %u",y);
    printf("\nOriginal address in z = %u",z);
    x++;
    y++;
    z++;
    printf("\nNew address in x = %u",x);
    printf("\nNew address in y = %u",y);
    printf("\nNew address in z = %u",z);
}
```

**Output**

```
Value of i = 3
Value of j = 1.500000
Value of k = c
Original address in x = 65524
Original address in y = 65520
Original address in z = 65519
New address in x = 65526
New address in y = 65524
New address in z = 65520
```

From the last three lines of the output, 65526 is the original value in x plus 2, 65524 is original value in y plus 4, and 65520 is original value in z plus 1. It happens because every time a pointer is incremented, it points to the immediate next location of its type. That is why, when the integer pointer x is incremented, it points to an address two locations after the current location, since an int is always 2 bytes long. Similarly, y points to an address 4 locations after the current location and z points 1 location after the current location. This is a very important result and can be effectively used while passing the entire array to a function. The way a pointer can be incremented, it can be decremented as well, to point to earlier locations. Thus, the following operations can be performed on a pointer:

1. **Adding Numbers to Pointers** Adding a number to a pointer leads the pointer to a new location which is given by- original location + (number added * bytes taken by the variable). For example, if an integer pointer “ptr” points to 65524, the statement “ptr=ptr+5” would make to pointer now point to 65524 + (2 * 5) i.e., 65534.
2. **Subtracting Numbers from Pointers** The subtraction pointers are similar to that of addition. The only difference is that in subtraction, the new location will be- original location - (number * bytes taken by variable).
3. **Subtracting Pointers** If you subtract a pointer from other pointer, the result will be the number of memory locations existing between the address pointed by both the pointers. For example, if a pointer “ptr1” pointing to location 65524, is subtracted from a pointer “ptr2” pointing to location 65532, the result will be 8.

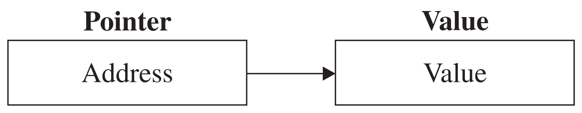
### Unacceptable Pointer Arithmetic Operations

The following arithmetic operations, involving the use of pointers, are not allowed:

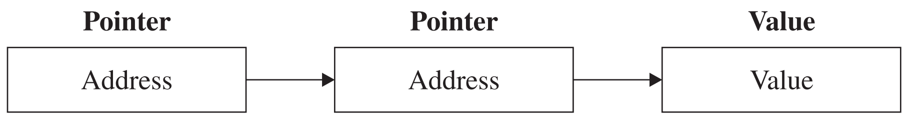
❖ Addition of two pointers.
❖ Multiplication of two pointer variables.
❖ Division of two pointer variables.

## 15.9 MULTIPLE INDIRECTION

C compiler allows creating a pointer to a pointer. We can have a pointer which points to another pointer pointing to the target value. This situation is called multiple indirection, or pointer to pointer.



(a) Single Indirection



(b) Multiple Indirection

From the above figure, the value of a normal pointer is the address of the object that contains the desired value. In case of pointer to pointer, the first pointer contains the address of the second (pointer), which points to the object that contains the desired value. Pointer to a pointer is declared by placing an additional asterisk in front of the pointer variable name.

**Example**
```
float **amt;
```

Here amt is a pointer to a pointer of type **float.** It is important to note that amt is not a pointer to a floating point number but rather a pointer to a **float pointer**.

### Example Program

```
void main()
{
int N,*P, **Q; // P is a pointer to integer, Q is pointer to a pointer to an integer.
N = 50;
P = &N;
Q = *P;
printf("\n %d", **Q); // print the value of N
}
```

**Output**

```
50
```

### Example Program

```
#include<stdio.h>
void main()
{
int *p, num ;
p = &num;
*p = 100;
printf("\n %d",num);
(*p)++;
printf("\n %d",num);
(*p)--;
printf("\n %d",num);
}
```

**Output**

```
100
101
100
```

## 15.10 POINTERS AND ARRAYS

Pointers and Arrays are interlinked interestingly. We know that array elements are located continuously in memory and array name is really a pointer to the first element in the array. This brings up an interesting relationship between array and pointers. When an array is declared, the compiler allocates a base address and enough memory to store all the elements of the array in contiguous memory locations. When array is simply referred by its name without any subscript, it specifies a pointer to its base address which is nothing but the location of the first element of the array.

**Example**

```
int A[5] = { 2,4,6,8,10};
```

Consider the base address of A is 2000. Here the array is declared as int array, so it needs 2 bytes memory for storage. In total 10 bytes are required for the entire array.

| A[0] | A[1] | A[2] | A[3] | A[4] | element |
|---|---|---|---|---|---|
| 2 | 4 | 6 | 8 | 10 | |
| 2000 | 2002 | 2004 | 2006 | 2008 | address |

Simply the array name A means, it is constant pointer pointing to the first element A[0] and therefore the value of A is address of A[0], i.e., 2000.

A = &A[0] = 2000

So array names are easily assigned to a pointer of the same type. If we declare Ptr as an integer pointer, then we can do pointer assignment as follows:

int *Ptr, A[5];

Ptr = A;

This is equivalent to Ptr = &A[0];

Now every value of the array A can be accessed through pointer Ptr as follows:

Ptr = &A[0]; (address = 2000)

Ptr+1 = &A[1]; (address = 2002)

Ptr+2 = &A[2]; (address = 2004)

Ptr+3 = &A[3]; (address = 2006)

Ptr+4 = &A[4]; (address = 2008)

Address of A[3] = base address + (3 * scale factor of int)

= 2000 + (3 * 2) = 2006

## Example Program

```
#include<stdio.h>
int Array[]={1,3,5,7,9,11,13};
int *ptr;
void main()
{
int i;
ptr = &Array[0];
printf("\n");
for ( i =0 ; i < 7 ; i++)
{
printf( "\n Array[%d] = %d", i , Array[i]);
printf("\t ptr + %d = %d",i,*(ptr+i));
}
}
```

**Output**

```
Array[0]= 1   ptr+0=1
Array[1]= 3   ptr+1=3
Array[1]= 5   ptr+1=5
Array[1]= 7   ptr+1=7
Array[1]= 9   ptr+1=9
Array[1]= 11  ptr+1=11
Array[1]= 13  ptr+1=13
```

## Example program to add all the elements of an array using pointers

```
void main()
{
int *ptr, tot, i;
int A[5] = {2,4,6,8,10};
i = 0;
ptr = A;
printf("\n Element Value Address ");
while(i<5)
{
printf(" A[%d] %d %u\n",i,*ptr);
tot = tot + *ptr;
i++; p++;
}
printf("\n Total = %d", tot);
printf("\n &A[0] = %u",&A[0]);
printf("\n ptr = %u",ptr);
}
```

**Output**

```
Element   Value    Address
A[0]        2        200
A[1]        4        202
A[2]        6        204
A[3]        8        206
A[4]       10        208
Tot = 30
&A[0] = 200
ptr = 210
```

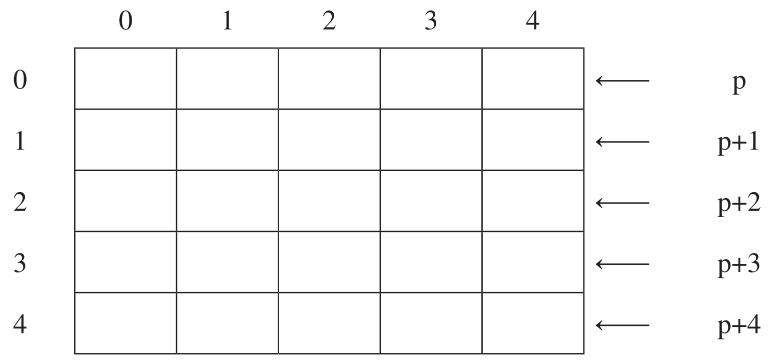
## 15.11 POINTER AND TWO DIMENSIONAL ARRAYS

Pointers can also be used to manipulate two dimensional arrays. In case of one dimensional array A, the pointer p holds the base address of an array, then the expression

*(A+i) or *(p+i) is valid, which represents the element A[i].

The same concept is used in two dimensional arrays also.

*(*(A+i) + j) or *(*(p+i) + j )



| | | |
|---|---|---|
| p | $\longrightarrow$ | pointer to first row |
| p+i | $\longrightarrow$ | pointer to $i^{th}$ row |
| *(p+i) | $\longrightarrow$ | pointer to first element in the $i^{th}$ row |
| *(p+i) +j ) | $\longrightarrow$ | pointer to the $j^{th}$ element in the $i^{th}$ row |
| *(* (p+i) + j) | $\longrightarrow$ | value stored in the cell (i,j) |

The base address of an array A is &A[0][0]. Taking this address as the starting address, the compiler allocates continuous memory for all the elements row wise. That is, the first element of second row is stored immediately after the last element of the first row, and so on.

**Example**

```
int a[3][4] = { { 1,1,1},
              {2,2,2},
              {3,3,3} ,
              {4,4,4}
            };
```

The elements are stored in computer memory as follows:

| Row 0 | | | Row 1 | | | Row 2 | | | Row 3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 4 | 4 | 4 |

$\longrightarrow$ Address = &a[0][0]

If we declare p as an int pointer with initial address of &a[0][0], then

A[i][j]= *(p+4 * i+j)

It can be noticed that, if we increment i by 1, the pointer p is incremented by 6 bytes, i.e., the size of each row. This is the reason when two dimensional arrays are declared, they need to specify the size of row, so that the compiler can determine the storage mapping.

```
#include<stdio.h>
#include<stdlib.h>
    int main(void)
{
int a[10][10],b[10][10],c[10][10],n=0,m=0,i=0,j=0,p=0,q=0,k=0;
    int *pt1,*pt2,*pt3;
printf("Enter size of 1st Matrix : ");
scanf("%d %d",&n,&m);
    for(i=0;i<n;i++)
{
       for(j=0;j<m;j++)
          {
          printf("Enter element no. %d %d :",i,j);
scanf("%d",&a[i][j]);
}
}
printf("Enter size of 2nd Matrix : ");
    scanf("%d %d",&p,&q);
for(i=0;i<p;i++)
{
for(j=0;j<q;j++)
       {
printf("Enter element no. %d %d :",i,j);
scanf("%d",&b[i][j]);
    }
}
if(m!=p)
    {
printf("Multiplication cannot be done\n");
exit (0);
}
pt1=&a[0][0];
pt2=&b[0][0];
pt3=&c[0][0];
for(i=0;i<n;i++)
    {
    for(k=0;k<q;k++)
       {
       *(pt3+(i*10+k))=0;
       for(j=0;j<m;j++)
{
*(pt3+(i*10+k))+=*(pt+(i*10+j))**(pt1+(j*10+k));
```

```
      }
}
}
Printf("\n Resultant Matrix\n");
for(i=0;i<n;i++)
      {
printf("\n "};
for(j=0;j<q;j++)
      {
      printf(" %d ",c[i][j]);
   }
printf("\n");
}
return 0;
}
```

**Output**

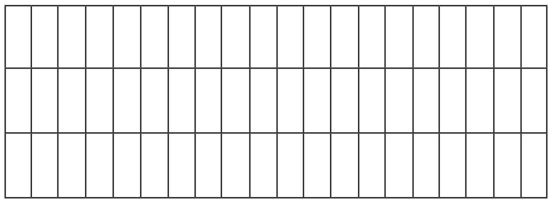
```
Enter size of 1st Matrix: 2 2
Enter element no. 0 0: 2
Enter element no. 0 1: 2
Enter element no. 1 0: 2
Enter element no. 1 1: 2
Enter size of 2nd Matrix: 2 2
Enter element no. 0 0: 2
Enter element no. 0 1: 2
Enter element no. 1 0: 2
Enter element no. 1 1: 2
Resultant Matrix
 8 8
 8 8
```

## 15.12 ARRAY OF POINTERS

Pointers can be effectively used to handle table of strings. When arrays are used to represent list of strings, they have a drawback.

char name[3][20];

The above declaration statement allocates (3*20) 60 bytes of memory to store the list of strings represented by name variable. The memory allocation is as follows:



The name table contains 3 names, each with a maximum length of 20 characters (including NULL). The total storage allocated for name variable is 60 bytes. During execution, all the three strings are not always equal in size, so some part of the memory remains unused. In order to reduce the memory wastage, pointers are utilized. Here, array of pointers are declared to allocate proper amount of memory as per the original requirement during execution.

```
char *name[3] = { "New Delhi",
                  "America",
                  "India"
                };
```

The above declares **name** to be an **array of three pointers** to characters, each pointer pointing to a particular name.

name[0] $\longrightarrow$ "New Delhi"
name[1] $\longrightarrow$ "America"
name[2] $\longrightarrow$ "India"

The system allocates only 24 bytes.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| N | e | w | | D | e | l | h | i | \0 |
| A | m | e | r | i | c | a | \0 | | |
| I | n | d | i | a | \0 | | | | |

The following program segment will read and display all three names in the allotted memory locations.

```
for (i=0; i<3; i++)
      scanf("%s", name[i]);
for (i=0; i<3; i++)
      printf("\n %s",name[i]);
```

to access the j^th^ character in the i^th^ name ,

```
      *(name[i]+j);
```

Note: *p[3] and (*)p[3] , these two are two different notation.
*p[3] declares p as an **array of three pointers**.
(*)p[3] declares p as a **pointer to an array of three elements**.
Since * has lower precedence than [].

## Program for sorting of strings in C language using pointers.

```
#include<stdio.h>
#include<string.h>
void main()
{
   int i,j,n;
   char str[20][20],temp[20];
   printf("Enter the no. of string to be sorted");
   scanf("%d",&n);
   printf("\n Enter the string one by one");
   for(i=0;i<=n;i++)
     gets(str[i]);
   for(i=0;i<=n;i++)
     for(j=i+1;j<=n;j++)
{
        if(strcmp(str[i],str[j])>0)
        {
          strcpy(temp,str[i]);
          strcpy(str[i],str[j]);
          strcpy(str[j],temp);
        }
     }
   printf("The sorted string\n");
   for(i=0;i<=n;i++)
     puts(str[i]);
   return 0;
}
```

**Output**

```
Enter the number of string to be sorted 5
Enter the string one by one
   Ram
   Aakash
   Mani
   Siva
   Divya
The sorted string
   Aakash
   Divya
   Mani
   Ram
   Siva
```

## 15.13 POINTERS AND FUNCTIONS

While passing array elements into a function, pointers play a vital role. When an array is passed to a function as an argument, only the address of the first element of the array and the number of elements in the array is enough. The starting address, the nature of the array, and number of elements in the array, are useful to find the block of memory allocated to an array based on the scale factor. Any modification done by the called function will affect the original values stored in the locations.

### Example program to sort N elements using pointer to an array.

In this example, the function parameters are declared as pointers. The dereferenced pointers are used in the function body. When the function is called, the addresses are passed as actual arguments.

```
#include<stdio.h>
#include<conio.h>
void sort(int int *);
void main()
{
int i, n,*arr;
printf("\n Enter number of elements in an array ");
sacnf("%d",&n);
printf("\n Enter the elements one by one ");
for( i =0 ; i<n; i++)
scanf(" %d',arr+i);
printf("\n Unsorted Array elements ….");
for( i =0 ; i<10; i++)
printf("\n %d',*(arr+i));
sort( n , *arr); // sort function invocation with the signature of one int and a pointer
to an int array
printf("\n Sorted Array elements ….");
for( i =0 ; i<10; i++)
printf(" %d',*(arr+i));
}
void sort(int m, int *x)
{
  int i,j,temp;
  for ( i=0;i<m; i++)
  {    for ( j= i+1; j<m; j++)
    { if( *(x+i) >= *(x+j))
      { temp = *(x+i);
        *(x+i) = *(x+j);
        *(x+j) = temp;
      }
  }
}
```

**Output**

```
Enter number of elements in an array
5
Enter the elements one by one
44
22
11
55
33
Unsorted Array elements ….
44
22
```

```
11
55
33
Sorted Array elements ….
11
22
33
44
55
```

In the above example, void sort(int, int *), tells the compiler that the formal argument that receives the array is a pointer and not array variable.

Pointer variables are effectively utilized while handling strings. Consider the following program segment, which copies the content of one string into another.

```
copy(chat *s1, char *s2)
{
   while(( *s1++ = *s2++) != ‘ \0’);
}   }
```

This copies the contents of s2 into s1.

## 15.14 FUNCTIONS RETURNING POINTERS

The main characteristic of a function is that it can return a single value or return multiple values through pointers. Since pointers are powerful data structure in C, so it is possible to return a pointer to the calling function.

**Example**

```
int *largest(int *, int *) ; // function prototype
void main()
{
   int a = 10;
   int b = 20;
   int *p;
   p = largest(&a, &b); // function call
   printf(“\n largest value = %d”,*p);
}
int *largest(int *x, int *y)
{
   if(*x > *y)
      return(x); // address of a
   else
      return(y); // address of b
}
```

## 15.15 POINTERS TO FUNCTION

According to any C programming language, when we type the program in C editor, it occupies a particular part of memory. Each function including void main() occupies its own memory in physical storage. A function, like a variable has a type and an address location in the memory. Therefore it is possible to declare pointer to a function which can be passed as argument to other functions. All the instructions are converted into machine readable form by the compiler and it gets stored in available memory space. During compilation, the system allocates necessary memory spaces for all the variables declared in the declaration statements. These memories are carefully handled by pointers and their initializations.

A pointer to a function is declared as follows:

type ( *fptr) ();

The above instruction provides the following information to the compiler:

fptr is a pointer to a function which returns type value. The parentheses around *fptr are compulsory. We can make a function pointer point to a specific function by simply assigning the name of the function to the pointer.

**Example**

```
double multiply(int,int);
double (*mptr)();
mptr = multiply;
```

The above statement declares mptr as a pointer to a function of type which returns double as the data value. Multiply as a function whose address is assigned to mptr. To call the function multiply, we can use pointer mptr with the list parameters.

(*mptr) (x,y) // function call

**Example** `int (*fptr)(int, int)`

The above line declares a function pointer ‘fptr’ that can point to a function whose return type is ‘int’ and takes two integers as arguments.

```
#include<stdio.h>
int func (int a, int b)
{
printf("\n a= %d",a);
printf("\n b= %d",b);
return 0;
}
void main()
{
int (*fptr)(int,int); // function poiner
fptr = func; // assign address to function pointer
funct(5,8);
fptr( 7,14)
}
```

In the above example, a function ‘func’ takes two integers as inputs and returns one integer as output. In the main() function, we declare a function pointer ‘fptr’ and then assign value to it. Note that, name of the function can be treated as starting address of the function so we can assign the address of function to function pointer using function’s name.

**Output**

```
      a = 5
      b = 8
      a = 7
      b = 14
```

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

int sum(int x, int y)
{
return x+y;
}

int main( )
{
  int (*fp)(int, int);
  fp = sum;
  int s = fp(10, 15);
  printf("Sum is %d",s);
  getch();
  return 0;
}
```

**Output**

```
25
```