ARRAYS

INTRODUCTION :

If a program to store the roll numbers of students of a class is to be written., then we will have 2 options:-

Suppose , students = 60

<u>Option 1:</u>-	declare 60 different variables and store each roll no in a variable.

<u>Option 2:</u>-	declare a single variable in such a way that, we can store all the roll no's in it.

Definitely, the programmer's choice will be option-2. In order to declare a variable, to store many values in it, we have to use the concept of <u>"ARRAY"</u> .

DEFINITION:

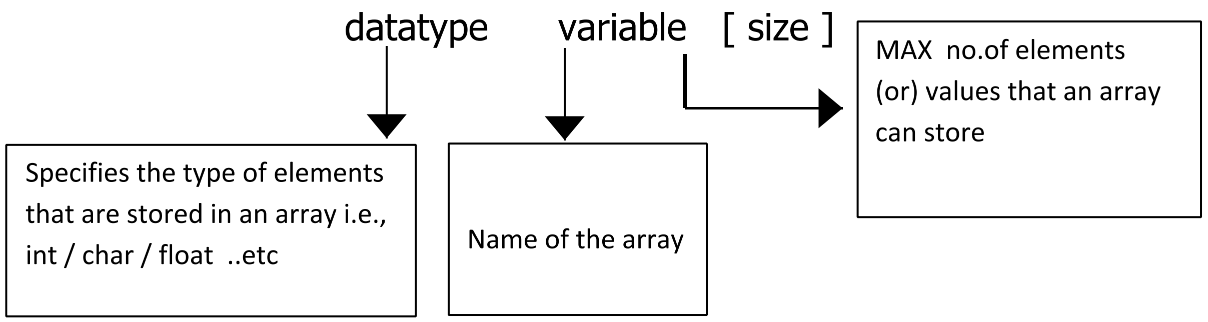
An array can be defined as, a group of related data items that share a common name. these data items may be int / char / float / double ..etc

- All these are stored in continuous memory locations (on RAM)
- Sometimes an array can be called as a SUBSRCIPTED VARIABLE

Difference between ARRAY and VARIABLE:

A variable can store only a single value at a time. Where as an array can store a multiple values at a time.

Syntax of Array:


datatype
variable
[size]
MAX no.of elements (or) values that an array can store
Specifies the type of elements that are stored in an array i.e., int / char / float ..etc
Name of the array

Note :	An array at a time can hold multiple values of similar data type only i.e., at a time array can hold group of integers, (or) group of floating point numbers etc..,

Declaration of variable:

```
int  a;
```

this syntax indicates:-

- can hold 1 integer value
- only 2 bytes of memory is registered. Therefore it is integer data type.

Declaration of array:

int a [5] ;

- can hold 5 integer values
- for each integer value 2 bytes of memory will be registerd i.e.,

5 * 2 = 10 bytes of memory is registerd

Total memory size of Array:

Total size = size * (size of data type)

Ex:-

(i) float k[10];

total size = 10 * 4 => 40 bytes

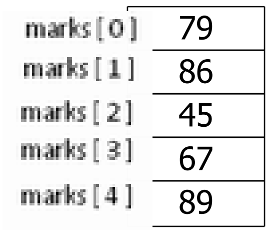
(ii) char c[5];

total size = 5 * 1 => 5 bytes

Declaring an array and Defining an array:

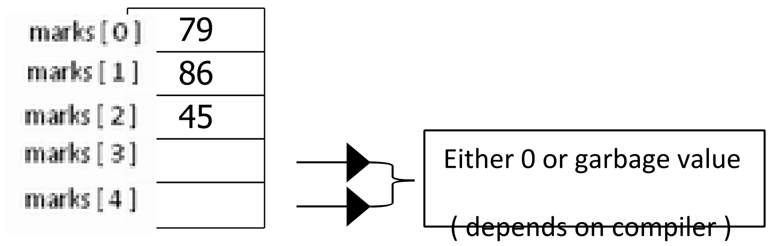
(i) int marks [5] = { 70, 86, 45, 67, 89 };

in the above declaration, the array 'marks [5]' will store the values as ahown below:



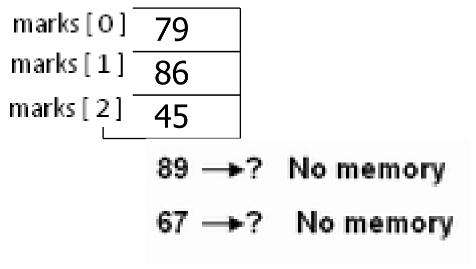
(ii) int marks [5] = { 79, 86, 45 };

in the above declaration the values are stored in memory as follows:-



(iii) int marks [3] = {79, 86, 45, 89, 67};

Memory allocation will not be done properly for the above declaration and it results in an error - (too many parameters in declaration)



This is because , size of array is given as only 3. Thefore for 4th and 5th elements memory will not allocated, which results in an error.

(iv) int marks [3] = { 79, 86, 45, 89, 67 } ;

for this declaration , memory will be allocated based on the input values of an array. i.e.,

in above declaration , 5 values are being given as inputs, the compiler automatically, fits the size of array as 5 and gives 5*2 bytes = 10 bytes of memory.

NOTE: Initialization of an array must be done using curly braces (flower brackets) only.

Datatype name[size] = {element1,…};

WITHOUT ARRAY	WITH ARRAY

void main()
{
int a-10, b=20, c=30;
printf(" a = %d ,", a);
printf("b = %d ,", b);
printf("c = %d .", c);
}
output:
a=10 , b=20 , c=30 .
	void main()
	{
	int a[5]= {10, 20, 30}, i;
	printf("array elements are:");
	for(i=0;i<5;i++)
	{
	Printf(" %d ",a[i]);
	} }
	Output: array elements are: 10 20 30

NOTE: all the array elements are numbered, starting from zero (0).

int m[5] ={75, 85, 68, 97, 54};
In above declaration,
m[2] =68 i.e. 3rd element of the list, as it starts from 0, not 1.

ENTERING DATA INTO AN ARRAY:

Generally, values at runtime can be assigned to an array using "for loop".

Ex: printf("enter marks");

```
for(i=0;i<10;i++)
{
scanf("%d",&marks[i]);
}
```

The for loop causes the process of assigning values at a runtime, till i=9 i.e. 10 values (0-9) are stored in an array. Values will be stored, in the array as:-

Marks[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

// WAP to accept values for five integers, at runtime and display them.

```
Void main()
{
Int a[20], i;
```

```
Printf("enter values into array");

For(i=0;i<5;i++)

{

Scanf("%d",&a[i]);

}
```

→ To accept values into an array.

```
Printf("values in the array are:");

For(i=0;i<5;i++)

{

printf("%d",&a[i]);

}
```

→To display values of array.

NOTE: Even to print the values of array, “for loop” should be used

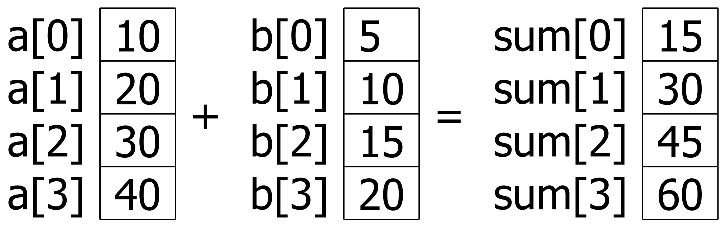
//Program to accept 2 arrays and add the corresponding elements of those arrays

```
void main()
{
        int a[10], b[10], sum[10];
        int I, j;
        printf(“enter size of Arrays”);
        scanf(“%d”, &j);
        printf(“enter values for array – 1”);
        for (i=0; i<j; i++)
        {
                Scanf(“%d”,&a[i]);
        }
        Printf(“enter values for array – 2”);
        for (i=0;i<j;i++)
        {
                Scanf(“%d”,&b[i]);
        }
        for(i=0;i<j;i++)
        {
                Sum[i]=a[i]+b[i];
        }
        printf(“sum of arrays is : “);
        for(i=0;i<j;i++)
        {
                printf(“%d”,sum[i]);
        }
}
```

OUTPUT:-

```
Enter size of arrays 4
Enter values for array -1   10 20 30 40
Enter values for array -2   5 10 15 20
Sum of arrays is : 10 30 45 60
```

Elements are stored in array as shown below:-



//Program to print the largest element of the array

```
#include<stdio.h>
Main()
{
        int x[6], large;
        int i;
        printf("enter elements in array");
        for(i=0; i<6;i++)
        {
                scanf("%d",&x[i]);
        }
large=x[0];                     /*Assuming x[0] in large */
for(i=1, i<6;i++)
{
        if(x[i]>large)
        {
                large=x[i];
        }
}
printf("largest element is %d",large);
}
```

SEARCHING

Gathering any information (or) trying to find any data is said to be a Searching process.

Searching technique can be used more efficiently if the data is present in an ordered manner.

Most widely used Searching methods are:-

i) Linear Search (Sequential Search)
ii) Binary Search

LINEAR SEARCH:-

Suppose an array is given, which contains 'n' elements. If no other information is given and we are asked to search for an element in array, than we should compare that element, with all the elements present in the array. This method which is used to Search the element in the array is known as Liner Search. Since the key element/ the element which is to be searched in array, if found out by comparing with every element of array one-by-one, this method is also known as Sequential Search.

Example:-
An array 'x' is given, which contains 5 elements in it i.e.,

int x[5]= { 75, 52, 61, 43,88};

These are stored in the memory in continuous memory location.
Fig.(a)

75	52	61	43	88
x[0]	x[1]	x]2]	x[3]	x[4]

We are asked to search for an element '43' in the array.

Then we have to compare '43' with each and every element of the array. This is represented in the below figures:-

Fig.(b)

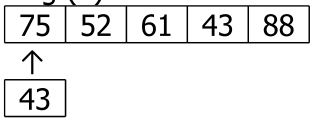
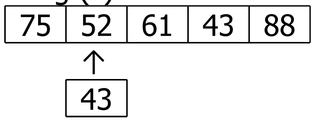


fig.(c)



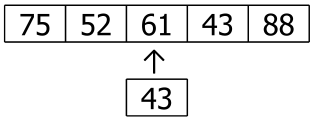


Fig.(d)

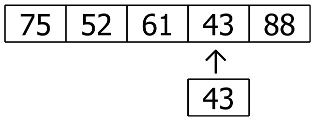


fig.(e)

// WAP to demonstrate LINEAR SEARCH

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int  linear[20],n,i,k,temp=0;
clrscr();
printf(" enter range of elements");
scanf(" %d",&n);
printf("enter elements into array");
for(i=0; i<n; i++)
{
        scanf("%d",&linear[i]);
}
printf("enter search key:");
scanf("%d",&k);

for(i=0, i<n; i++)
{
        if(k==linear[i])
        {
        temp=1;
        printf("%d is found at location %d", k, i+1);
        break;
        }
}
if(temp!=1)
{
        printf("element not found");
}
getch();
}
```

OUTPUT:-

```
Enter range ofelements
5
Enter elements into array
45  68  75  83  99
Enter search key
83
83 is found at location 4
```

Reason for using 'temp" variable:

Without "temp", if we write else statement (or) else-if, they should be written after 'if' inside for loop. If we write outside for loop and error misplaced else will occur.

If we write inside for loop, every time those statements will be executed. Therefore we use 'temp' variable and assign its value to 1, and use the 'temp' variable in 'if' condition. Whenever search key is not found in the list.

EXPLANATION:-

Number of elements for which Linear search technique is to be performed, should be taken i.e. array size. (n).

Let n=5.

5 elements are entered into an array using for loop and stored in continuous memory locations.

	1	2	3	4	5
	72	45	68	59	83
Linear	[0]	[1]	[2]	[3]	[4]

The element which is to be searched is taken i.e., Search key element (k)

k=59

According to the linear search method, now, '59' should be compared with every element in the list and when '59' matches with the element in the list, then its position should be displayed. To implement this logic, the c-Program code is:-

```
for(i=0; i<n; i++)
{
        if(k==liner[i]);
        {
        temp=1;
        printf("%d found at %d",k, i+1);
        }
}
if (temp!=1)
printf("element not found");
```

BINARY SEARCH:-

Efficient search method for large arrays. Liner search, will required to compare the key element, with every element in array. If the array size is large, liner search requires more time for execution.

In such cases, binary search technique can be used.

To perform binary search:-

i) Elements should be entered into array in Ascending Order

ii) Middle element of the array must be found. This is done as follows

Find the lowest position & highest position of the array i.e., if an array contains 'n' elements then:-

Low=0
High =n-1
Mid =(low+high)/2

Note: We are calculating mid position of the array not the middle element. The element present in the mid position is considered as middle element of array.

Now the search key element is compared with middle element of array. Three cases arises

Case 1 : If middle element is equal to key, then search is end.
Case 2 : If middle element is greater than key, then search is done, before the middle element of array.
Case 3 : If middle element is less than key, then search is done after the middle element of array.

This process is repeated till we get the key element (or) till the search comes to an end, since key element is not in the list.

//PROGRAM TO DEMONSTRATE BINARY SEARCH

```
#include<stdio.h>
#include<conio.h>
Void main()
{
int binary[20],n, i, k, low, mid, high;
clrscr();
printf("enter range of elements");
scanf("%d",&n);
printf("enter elements into array");
for(i=0, i<n; i++)
{
        scanf("%d",&binary[i]);
}
printf("enter search key");
scanf("%d",&k);
low=0;
high=n-1;
while(low<=high)
{
        mid=(low+high)/2;
        if(binary[mid]<k)
        {
                low=mid+1;
        }
        else if(binary[mid]>k)
        {
                high=mid-1;
        }
        else
        break;
}
if(binary[mid]==k)
{
        printf("element is found at location %d",mid+1);
}
else
printf("element not found");
getch();
}
```

OUTPUT:-

```
Enter the range of elements
5
Enter elements into array
22    28   34    45    58
Enter search key
34
Element found at location 3
```

SORTING

Sorting means arranging the given data in a particular order.
Some of the Sorting techniques are:

i) Bubble sort
ii) Selection sort

BUBBLE SORT

In bubble sort each element is compared with its adjacent element.

If first element is larger than second one, then both elements are swapped. Other wise, element are not swapped.

Consider the following list of numbers.
Example:

74	39	35	97	84
a[0]	a[1]	a[2]	a[3]	a[4]

74 & 39 are compared
Bcoz 74 > 39, both are swapped. Now list is

39	74	35	97	84
a[0]	a[1]	a[2]	a[3]	a[4]

74 &35 are compared
Bcoz 74 > 35, both are swapped, Now list is

39	35	74	97	84
a[0]	a[1]	a[2]	a[3]	a[4]

74 & 97 are compared
Bcoz 74 < 97 list remain unchanged

97 & 84 are compared
Bcoz 97> 84, both are swapped, the list becomes

39	35	74	84	97
a[0]	a[1]	a[2]	a[3]	a[4]

Note: After first pass, largest element in given list occupies the last position.
After second pass, second largest element is placed at second last position and so on..

//PROGRAM TO DEMONSTRATE BUBBLE SORT

```
#include<stdio.h>
#include<conio.h>
void main()
{
          int  i, j, n, bubble[20], temp;
          clrscr();
          printf("enter range of elements");
          scanf("%d",&n);
          printf("enter elements");
          for(i=0, i<n; i++)
          {
                    scanf("%d",&bubble[i]);
          }
for(i=0; i<n; i++)
{
          for(j=0;j<n-1;j++)
          {
          if(bubble[j] > bubble[j+1])
          {
          temp=bubble[j];
          bubble[j]=bubble[j+1];
          bubble[j+1]=temp;
          }
          }
}
printf("after sorting");
for(i=0; i<n; i++)
{
          printf("%d",bubble[i]);
}
}
```

OUTPUT

```
enter range of elements
5
Enter elements
3 2 5 4 6
After sorting
2 3 4 5 6
```

SELECTION SORT:

In selection sort, element at first location (0th location) is considered as least element, and it is compared with the other elements of the array. If any element is found to be minimum than the element in first location, then that location is taken as minimum and element in that location will be the minimum element.

After completing a set of comparisions, the minimum element is swapped with the element in first location (0th location).

Then again element second location is considered as minimum and it is compared with the other elements of array and the process continues till the array is sorted in ascending order.

Note: After first pass, smallest element in given list occupies the first position.
After second pass, second largest element is placed at second position and so on..

//PROGRAM TO DEMONSTRATE SELECTION SORT

```
#include<stdio.h>
#include<conio.h>
void main()
{       int i, j, n, a[100], t, min;
        clrscr();
        printf("enter range of elements");
        scanf("%d",&n);
        printf("enter elements:");
        for(i=0, i<n; i++)
        {
                scanf("%d",&a[i]);
        }
        printf("elements before sorting");
        for(i=0; i<n; i++)
        {
                printf("%d",a[i]);
        }
        for(i=0; i<n-1; i++)
        {
                min=i;
                for(j=i+1; j<n; j++)
                {
                        if(a[j]<a[min])
                        min=j;
                }
                if(min!=i)
                {
                        t=a[i];
                        a[i]=a[min];
                        a[min]=t;
                }
        }
                printf("elements after sorting are");
                for(i=0; i<n; i++)
                {
                        printf("%d", a[i]);
                }
        getch();
}
```

OUTPUT

```
enter range of elements
5
Enter elements
3 2 5 4 6
After sorting   2 3 4 5 6
```

TWO DIMENSIONAL ARRAYS

There are 3 types of arrays:-

1. 1 D Array
2. 2 D Array
3. Multi – Dimensional Array

- Maximum limit of Arrays is compiler dependent.

If we want to represent an array in a matrix form we will be using 2 D Arrays.

General form of an 2D array is

```
Datatype     array_ name[row_size][column_size];
```

Ex: int i[4][3];

An array 'i' is declared which contains 12 integer values in 4 rows and 3 columns.

Initializing a 2D array in program:

```
int   i[4][3]  =  { { 1,2,3 } , { 4,5,6 } , { 7,8,9 } , { 10,11,12 } };
```

or

```
int   i[4][3]  =  { 1,2, 3,4, 5, 6 , 7, 8, 9,10,11,12 };
```

or

```
int   i[][3]  =  { { 1,2,3 } , { 4,5,6 } , { 7,8,9 } , { 10,11,12 } };
```

NOTE: It is important to remember that while intialising an array it is necessary to mention the second(column) dimension, whereas the first dimension(row) is optional

$$\begin{array}{c|ccc|} & c1 & c2 & c3 \\ r1 & 1 & 2 & 3 \\ r2 & 4 & 5 & 6 \\ r3 & 7 & 8 & 9 \\ r4 & 10 & 11 & 12 \end{array} \; 4 \times 3$$

MEMORY OF 2 D ARRAY:

In memory it is not possible to store elements in form of rows and columns. Whether it is a 1 D (or) 2 D Array, the elements are stored in continuous memory locations. The arrangement of elements of a 2 D is shown below:

int a[4][3] = = { { 10,20,30 } , { 4,8,9 } , { 23,41,32 } , { 15,18,24 } };

a[0][0] a[0][1] a[0][2] a[1][0] a[1][1] a[1][2] ……
…..a[3][2]

10	20	30	4	8	9	23	41	32	15	18	24

1000 1002 1004 1006 1008 …………….
1022

// WAP to read a 2-D Array and print it.

```
Void main()
{
int a[10][10] , i , j , m , n ;
Printf "enter the order of matrix  \n ");
Scanf( " %d%d " , &m, &n);
Printf " \n enter the elements of array:  \n ");
for(i=0;i<m;i++)                //     for rows.
{
for (j=0;j<n;j++)               //     for columns.
{
Scanf("%d", &a[i][j]);
}
}
for(i=0;i<m;i++)                //     for rows.
{
for (j=0;j<n;j++)               //     for columns.
{   printf("%d", a[i][j]);
}
print( " \n ");
} }
```

// Write a program for ADDITION OF 2 MATRICES

```
# include <stdio.h>
# include <conio.h>
void main()
{
  int a[10][10],b[10][10],c[10][10];
  int i,j;
  int r1,r2,c1,c2;
  printf("Enter the size of the first matrix ( rows and coloums)\n");
  scanf("%d%d",&r1,&c1);
  printf("Enter the size of the seond matrix (rows and coloums)\n");
  scanf("%d%d",&r2,&c2);
  printf("Enter the elements in matrix one");
  for(i=0;i<r1;i++)
 {   for(j=0;j<c1;j++)
 {
  scanf("%d",&a[i][j]);
}
}
```

```
 printf("Enter the elements in matrix two");
 for(i=0;i<r2;i++)
 {
 for(j=0;j<c2;j++)
 {
 scanf("%d",&b[i][j]);
 }   }
 for(i=0;i<r1;i++)
 {
 for(j=0;j<c1;j++)
 {
 c[i][j]=a[i][j]+b[i][j];
 }
}
 printf("The sum of the two matrices is \n");
 for(i=0;i<r1;i++)
 {
 for(j=0;j<c2;j++)
 {
```

```
printf("   %d",c[i][j]);

}

printf("\n");

}

}
```

// Program for MULTIPLICAION OF 2 MATRICES

```
#include<stdio.h>

#include<conio.h>

void main()

{

        int a[10][10],b[10][10],c[10][10],r1,c1,r2,c2,i,j,k;

        printf("Enter the number of rows and columns in 1st matrix\n");

        scanf("%d%d",&r1,&c1);

        printf("Enter the number of rows and columns in 2nd matrix\n");

        scanf("%d%d",&r2,&c2);

      if(c1 == r2)

        {       printf("Enter the elements of the matrix of a\n");

                for(i=0;i<r1;i++)

                {

                        for(j=0;j<c1;j++)

                        {

                                scanf("%d",&a[i][j]);

                        }

                }
```

```
printf("Enter the elements of the matrix of b\n");
for(i=0;i<r2;i++)
{
        for(j=0;j<c2;j++)
        {
                scanf("%d",&b[i][j]);
        }
}
for(i=0;i<r1;i++)
{
        for(j=0;j<c2;j++)
        {
                c[i][j]=0;
                for(k=0;k<c2;k++)
                {
                        c[i][j] =c[i][j] + (a[i][k] * b[k][j]);
                }
        }
}
```

```
        printf("\nProduct of the two matrices is\n");
        for(i=0;i<r1;i++)
        {
            for(j=0;j<c2;j++)
            {
                printf("  %d",c[i][j]);
            }
            printf("\n");
        }
    }
    else
        printf("Matrix multiplication not possible");
}
```

// Program to find transpose of given matrix and print the identity matrix of order m x n

```
#include<stdio.h>

#include<conio.h>

void main()

{

        int a[10][10],b[10][10],c[10][10],r1,c1,r2,c2,i,j,k;

        clrscr();

        printf("Enter the number of rows and columns\n");

        scanf("%d%d",&r1,&c1);

        printf("Enter the %d elements\n",r1*c1);

        for(i=0;i<r1;i++)

        {

                for(j=0;j<c1;j++)

                {

                        scanf("%d",&a[i][j]);

                }

        }
```

```
for(i=0;i<r1;i++)

{

        for(j=0;j<c1;j++)

        {

                b[j][i] = a[i][j];

        }

}

printf("Transpose of given matrix is\n");

for(i=0;i<c1;i++)

{

        for(j=0;j<r1;j++)

        {

                printf("%d  ",b[i][j]);

        }

        printf("\n");

}

if(r1 == c1)

{

        for(i=0;i<r1;i++)
        {       for(j=0;j<c1;j++)
                {       if(i==j)
                                a[i][j] = 1;
                        else
                                a[i][j] = 0;
                }
        }
        printf("Identity matrix of given order is\n");
        for(i=0;i<r1;i++)
        {        for(j=0;j<c1;j++)
                {
                   printf("%d  ",a[i][j]);
                }
                printf("\n");
        }
    }
    else
        printf("\nIdentity matrix should be square matrix");
}
```

// Program to check the EQUALITY OF 2 MATRICES

```
void main()

{

int a[10][10],b[10][10];

int sum,i,j,k,r1,r2,c1,c2,temp;

printf("enter rows and columns matrix-1\n");

scanf("%d %d",&r1,&c1);

printf("enter rows and columns matrix-2\n");

scanf("%d %d",&r2,&c2);

if(r1==r2&&c1==c2)

{    printf("enter elements of the matrix-1\n");

for(i=0;i<r1;i++)

{

        for(j=0;j<c1;j++)

        {

                scanf("%d",&a[i][j]);

                printf("\n");

        }

}
```

```
printf("enter elements of the matrix-2\n");
for(i=0;i<r2;i++)
{
        for(j=0;j<c2;j++)
        {
                scanf("%d",&b[i][j]);
                printf("\n");
        }
}
/*printing matrix-1*/
printf("matrix-1\n\n");
for(i=0;i<r1;i++)
{
        for(j=0;j<c1;j++)
        {
                printf("%d\t",a[i][j]);
        }
        printf("\n\n\n");
}
```

```
/* printing matrix-2*/
printf("matrix-2\n\n");
for(i=0;i<r2;i++)
{
        for(j=0;j<c2;j++)
        {
                printf("%d\t",b[i][j]);
        }
        printf("\n\n\n");
}
   /* checking equality */
        for(i=0;i<r1;i++)
        {
                for(j=0;j<c2;j++)
                {
                        if(a[i][j]!=b[i][j])
                        {
                                temp=1;
                                break;
                        }
                }
        }
}

else
printf("cannot be compared");
if(temp==1)
printf("matrices are not equal");
else
printf("matrices are equal");
}
```

// Program to check whether the given matrix is SYMMETRIC MATRIX or not

```
void main()

{

int a[10][10],b[10][10];

int sum,i,j,k,m,n,temp;

clrscr();

printf("enter size of square matrix\n");

scanf("%d %d",&m,&n);

if(m==n)

{

printf("enter elements of the matrix\n");

for(i=0;i<m;i++)

{

        for(j=0;j<n;j++)

        {

                scanf("%d",&a[i][j]);

                printf("\n");

        }

}
```

```
printf("the input matrix is\n\n\n\n");
for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        {
                printf("%d\t",a[i][j]);


        }
        printf("\n\n\n\n\n\n");
}


for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        {
          b[i][j]=a[j][i];


        }
}
```

```
printf("transpose of a matrix is\n\n\n\n\n");
for(i=0;i<m;i++)
{       for(j=0;j<n;j++)
        {
                printf("%d\t",b[i][j]);


        }
        printf("\n\n\n\n\n\n");
}
for(i=0;i<m;i++)
{
        for(j=0;j<n;j++)
        {
                if(a[i][j]!=b[i][j])
                {
                        temp=0;
                        //printf("matrix is symmetric");
                }
                }
        }
        if(temp==0)


        printf("matrix is not symmetric");
        else
        printf("matrix is symmetric");
}
else
printf("symmetric matrix shud be a square matrix");
}
```

C - Strings

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**.

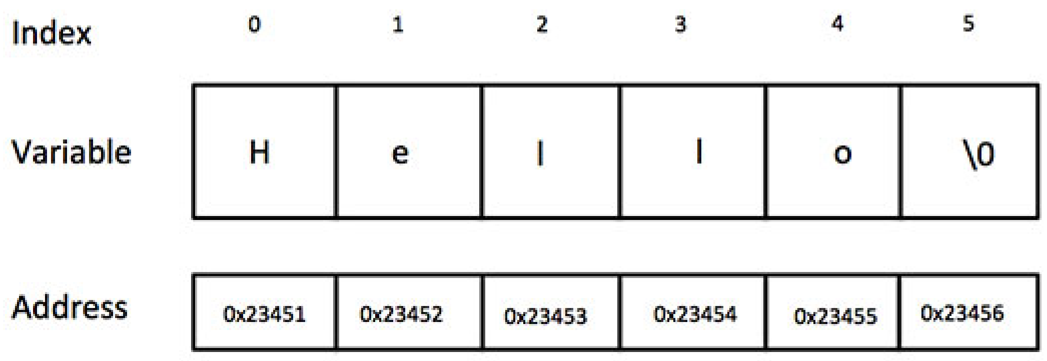
The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

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

If you follow the rule of array initialization then you can write the above statement as follows −

```
char greeting[] = "Hello";
```

Following is the memory presentation of the above defined string in C/C++ −



Actually, you do not place the *null* character at the end of a string constant. The C compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print the above mentioned string −

```
#include <stdio.h>


int main () {


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

   printf("Greeting message: %s\n", greeting );

   return 0;
}
```

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

```
Greeting message: Hello
```

C supports a wide range of functions that manipulate null-terminated strings −

Sr.No.	Function & Purpose
1	**strcpy(s1, s2);** Copies string s2 into string s1.
2	**strcat(s1, s2);** Concatenates string s2 onto the end of string s1.
3	**strlen(s1);** Returns the length of string s1.
4	**strcmp(s1, s2);** Returns 0 if s1 and s2 are the same; less than 0 if s1<s2; greater than 0 if s1>s2.
5	**strchr(s1, ch);** Returns a pointer to the first occurrence of character ch in string s1.
6	**strstr(s1, s2);** Returns a pointer to the first occurrence of string s2 in string s1.

String Handling Functions in C

C programming language provides a set of pre-defined functions called **string handling functions** to work with string values. The string handling functions are defined in a header file called **string.h**. Whenever we want to use any string handling function we must include the header file called **string.h**.

The following table provides most commonly used string handling function and their use...

Function	Syntax (or) Example	Description
strcpy()	strcpy(string1, string2)	Copies string2 value into string1
strncpy()	strncpy(string1, string2, 5)	Copies first 5 characters string2 into string1
strlen()	strlen(string1)	returns total number of characters in string1
strcat()	strcat(string1,string2)	Appends string2 to string1
strncat()	strncpy(string1, string2, 4)	Appends first 4 characters of string2 to string1
strcmp()	strcmp(string1, string2)	Returns 0 if string1 and string2 are the same; less than 0 if string1<string2; greater than 0 if string1>string2
strncmp()	strncmp(string1, string2, 4)	Compares first 4 characters of both string1 and string2
strcmpi()	strcmpi(string1,string2)	Compares two strings, string1 and string2 by ignoring case (upper or lower)

Function	Syntax (or) Example	Description
stricmp()	stricmp(string1, string2)	Compares two strings, string1 and string2 by ignoring case (similar to strcmpi())
strlwr()	strlwr(string1)	Converts all the characters of string1 to lower case.
strupr()	strupr(string1)	Converts all the characters of string1 to upper case.
strdup()	string1 = strdup(string2)	Duplicated value of string2 is assigned to string1
strchr()	strchr(string1, 'b')	Returns a pointer to the first occurrence of character 'b' in string1
strrchr()	'strrchr(string1, 'b')	Returns a pointer to the last occurrence of character 'b' in string1
strstr()	strstr(string1, string2)	Returns a pointer to the first occurrence of string2 in string1
strset()	strset(string1, 'B')	Sets all the characters of string1 to given character 'B'.
strnset()	strnset(string1, 'B', 5)	Sets first 5 characters of string1 to given character 'B'.
strrev()	strrev(string1)	It reverses the value of string1

The following example uses some of the above-mentioned functions –

The following example uses Strcpy(), strcat() and strlen()

```
#include <stdio.h>

#include <string.h>

int main () {

   char str1[12] = "Hello";
   char str2[12] = "World";
   char str3[12];
   int  len ;

   /* copy str1 into str3 */
   strcpy(str3, str1);
   printf("strcpy( str3, str1) :  %s\n", str3 );

   /* concatenates str1 and str2 */
   strcat( str1, str2);
   printf("strcat( str1, str2):   %s\n", str1 );

   /* total lenghth of str1 after concatenation */
   len = strlen(str1);
   printf("strlen(str1) :  %d\n", len );

   return 0;

}
```

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

```
strcpy( str3, str1) :  Hello
strcat( str1, str2):   HelloWorld
strlen(str1) :  10
```

****

Structures in C

In C programming language, a structure is a collection of elements of the different data type. The structure is used to create user-defined data type in the C programming language. As the structure used to create a user-defined data type, the structure is also said to be “user-defined data type in C”.
In other words, a structure is a collection of non-homogeneous elements. Using structure we can define new data types called user-defined data types that holds multiple values of the different data type. The formal definition of structure is as follows...

Structure is a collection of different type of elements under a single name that acts as user defined data type in C.

Generally, structures are used to define a record in the c programming language. Structures allow us to combine elements of a different data type into a group. The elements that are defined in a structure are called members of structure.

How to create structure?

To create structure in c, we use the keyword called "***struct***". We use the following syntax to create structures in c programming language.

```
struct <structure_name>
{
    data_type member1;
    data_type member2, member3;
    .
    .
}
```

Following is the example of creating a structure called Student which is used to hold student record.

Creating Structure in C

```
struct student
{
char stud_name[30];
int roll_number;
float percentage;
}
```

Importent Points to be Remembered

Every structure must terminated with semicolon symbol (;).
"struct" is a keyword, it must be used in lowercase letters only.

Creating and Using structure variables

In a c programming language, there are two ways to create structure variables. We can create structure variable while defining the structure and we can also create after terminating structure using struct keyword.
To access members of a structure using structure variable, we use dot (.) operator. Consider the following example code...

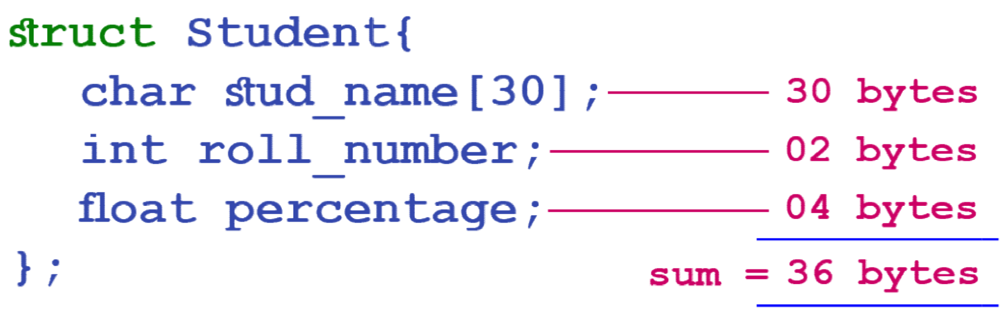
Creating and Using structure variables in C

```
struct Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} stud_1 ;                  // while defining structure

void main(){
    struct Student stud_2;  // using struct keyword

    printf("Enter details of stud_1 : \n");
    printf("Name : ");
    scanf("%s", stud_1.stud_name);
    printf("Roll Number : ");
    scanf("%d", &stud_1.roll_number);
    printf("Percentage : ");
    scanf("%f", &stud_1.percentage);

    printf("***** Student 1 Details *****\n);
    printf("Name of the Student : %s\n", stud_1.stud_name);
    printf("Roll Number of the Student : %i\n", stud_1.roll_number);
    printf("Percentage of the Student : %f\n", stud_1.percentage);
}
```

In the above example program, the stucture variable "**stud_1** is created while defining the structure and the variable "**stud_2** is careted using struct keyword. Whenever we access the members of a structure we use the dot (.) operator.

Memory allocation of Structure

When the structures are used in the c programming language, the memory does not allocate on defining a structure. The memory is allocated when we create the variable of a particular structure. As long as the variable of a structure is created no memory is allocated. The size of memory allocated is equal to the sum of memory required by individual members of that structure. In the above example program, the variables stud_1 and stud_2 are allocated with 36 bytes of memory each.



```
struct Student{
    char stud_name[30];
    int roll_number;
    float percentage;
};
```

Here the variable of **Student** structure is allocated with 36 bytes of memory.

Importent Points to be Remembered

All the members of a structure can be used simultaneously.
Until variable of a structure is created no memory is allocated. The

memory required by a structure variable is sum of the memory required by individual members of that structure.

Unions in C

In C programming language, the union is a collection of elements of the different data type. The union is used to create user-defined data type in the C programming language. As the union used to create a user-defined data type, the union is also said to be “user-defined data type in C”.
In other words, the union is a collection of non-homogeneous elements. Using union we can define new data types called user-defined data types that holds multiple values of the different data type. The formal definition of a union is as follows...

Union is a colloction of different type of elements under a single name that acts as user defined data type in C.

Generally, unions are used to define a record in the c programming language. Unions allow us to combine elements of a different data type into a group. The elements that are defined in a union are called members of union.

How to create union?

To create union in c, we use the keyword called "***union***". We use the following syntax to create unions in c programming language.

```
union <structure_name>
{
     data_type member1;
     data_type member2, member3;
     .
     .
} ;
```

ollowing is the example of creating a union called Student which is used to hold student record.

Creating union in C

```
union Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} ;
```

Importent Points to be Remembered
Every union must terminated with semicolon symbol (;).
"union" is a keyword, it must be used in lowercase letters only.

Creating and Using union variables

In a c programming language, there are two ways to create union variables. We can create union variable while the union is defined and we can also create after terminating union using union keyword.
TO access members of a union using union variable, we use dot (.) operator. Consider the following example code...

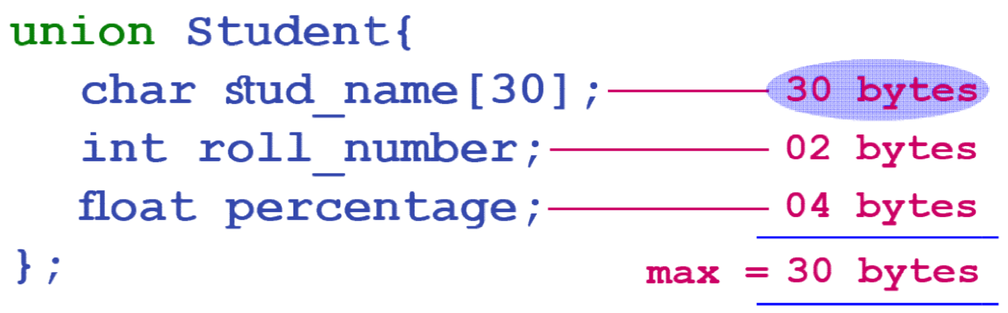
Creating and Using union variables in C

```
union Student
{
    char stud_name[30];
    int roll_number;
    float percentage;
} stud_1 ;                  // while defining union

void main(){
    union Student stud_2;  // using union keyword

    printf("Enter details of stud_1 : \n");
    printf("Name : ");
    scanf("%s", stud_1.stud_name);
    printf("Roll Number : ");
    scanf("%d", &stud_1.roll_number);
    printf("Percentage : ");
    scanf("%f", &stud_1.percentage);

    printf("***** Student 1 Details *****\n);
    printf("Name of the Student : %s\n", stud_1.stud_name);
    printf("Roll Number of the Student : %i\n", stud_1.roll_number);
    printf("Percentage of the Student : %f\n", stud_1.percentage);
}
```

In the above example program, the union variable "**stud_1** is created while defining the union and the variable "**stud_2** is careted using union keyword. Whenever we access the members of a union we use the dot (.) operator.

Memory allocation of Union

When the unions are used in the c programming language, the memory does not allocate on defining union. The memory is allocated when we create the variable of a particular union. As long as the variable of a union is created no memory is allocated. The size of memory allocated is equal to the maximum memory required by an individual member among all members of that union. In the above example program, the variables stud_1 and stud_2 are allocated with 30 bytes of memory each.



Here the variable of **Student** union is allocated with 30 bytes of memory and it is shared by all the members of that union.

POINTERS

INTRODUCTION

Pointers are one of the derived types in C. One of the powerful tool and easy to use once they are mastered.

Some of the advantages of pointers are listed below:

- ❖ A pointer enables us to access a variable that is defined outside the function. Pointers are more efficient in handling the data tables.
- ❖ Pointers reduce the length and complexity of a program.
- ❖ The use of a pointer array to character strings save data storage space in memory.

The real power of C lies in the proper use of pointers.

POINTER CONCEPTS

The basic data types in C are int, float, char double and void. Pointer is a special data type which is derived from these basic data types.

There are three concepts associated with the pointers are,

- ❖ Pointer Constants
- ❖ Pointer Values
- ❖ Pointer Variables

POINTER CONSTANT

- ❖ As we know, computers use their memory for storing the instructions of a program, as well as the values of the variables that are associated with it.
- ❖ The computer's memory is a sequential collection of 'storage cells'.
- ❖ Each cell can hold one byte of information, has a unique number associated with it called as 'address'.
- ❖ The computer addresses are numbered consecutively, starting from zero. The last address depends on the memory size.

- ❖ Let us assume the size of the memory is 64K then,

The total memory locations = 64K
= 64 * 1K
= 64 * 1024 bytes
= 65536 bytes (locations)

- ❖ So, here the last address is 65535(started with 0).
- ❖ Physically they are divided into even bank and odd bank.
- ❖ Even bank is set of memory locations with even addresses. Like 0, 2, 4, 6……65534.
- ❖ Odd bank is set of memory locations with odd addresses. Like 1, 3, 5 ….65535.

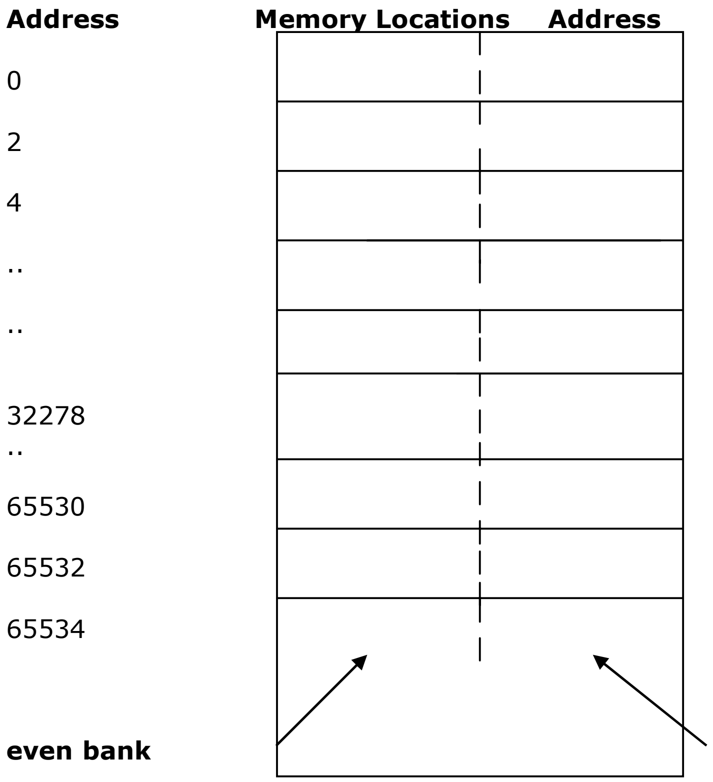


Figure: 3.1 Memory Organization.

- These memory addresses are called pointer constants.
- We cannot change them, but we can only use them to store data values.
- For example, in the above memory organization, the addresses ranging from 0 to 65535 are known as pointer constants.
- Remember one thing, the address of a memory location is a pointer constant and cannot be changed .

POINTER VALUE

Whenever we declare a variable, the system allocates, an appropriate location to hold the value of the variable somewhere in the memory.

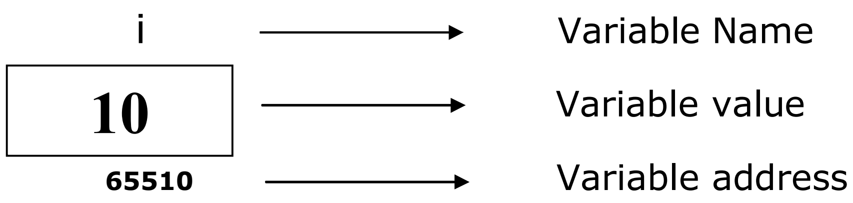
Consider the following declaration,

int i=10;

This declaration tells the C compiler to perform the following activities:

- Reserve **space** in the memory to hold the integer value.
- Associate the name i with this memory location.
- Store the value **10** at this location.

We can represent i's location in the memory by the following memory map:



Pointer Values

- Memory is divided into number of storage cells called locations.
- Out of these the addresses, the system assigns some addresses of the memory locations to the variables.
- These memory locations assigned to the variables by the system are called pointer values.
- For example, the address 65510 which is assigned to the variable i is a pointer value.

The & Operator

- The address of the variable cannot be accessed directly. The address can be obtained by using address operator**(&)** in C language.
- The address operator can be used with any variable that can be placed on the left side of an assignment operator.
- The format specifier of address is %u(unsigned integer),the reason is addresses are always positive values. We can also use %x to know the address of a variable.
- Example, to know the address of variable n, just use &n.

Note: Constants, expressions, and array name cannot be placed on the left side of the assignment and hence accessing address is invalid for constants, array names and expressions.

The following are illegal use of address Operator.

```
&125            (Pointing at constant)

int a[10];
&a              (pointing to array name)


&(x+y)          (pointing at expressions)
```

POINTER VARIABLE

- A variable which holds the address of some other variable is called pointer variable.
- A pointer variable should contain always the address only.

The * Operator

- It is called as **'Value at address'** operator. It returns the value stored at a particular address.
- It is also Known as **Indirection** or **Dereferencing Operator**

ACCESSING A VARIABLE THROUGH POINTER

For accessing the variables through pointers, the following sequence of operations have to be performed, to use pointers.

1. Declare an ordinary variable.
2. Declare a pointer variable.
3. Initialize a pointer variable (Provide link between pointer variable and ordinary variable).
4. Access the value of a variable using pointer variable.

We already familiar with the declaration and initialization of variable. Now we will discuss the remaining here.

Declaring a pointer variable

In C , every variable must be declared before they are used. Since the pointer variables contain address that belongs to a separate data type, they must be declared as pointers before we use them.
The syntax for declaring a pointer variable is as follows,

```
data type    *ptr_name;
```

This tells the compiler three things about the variable ptr_name.

1. The asterisk(*) tells that the variable ptr_name is a pointer variable.
2. ptr_name needs a memory location.
3. ptr_name points to a variable of type data type.

For example,

int *pi;

declares the variable p as a pointer variable that points to an integer data type.
Remember that the type **int** refers to the data type of the variable being pointed by **pi**.

Initializing Pointers

- ❖ Once a pointer variable has been declared, it can be made to point to a variable using statement such as

```
ptr_name=&var;
```

- ❖ Which cause **ptr_name** to point to **var**.Now **ptr_name** contains the address of **var**. This is known as pointer initialization.
- ❖ Before a pointer is initialized it should not be used.

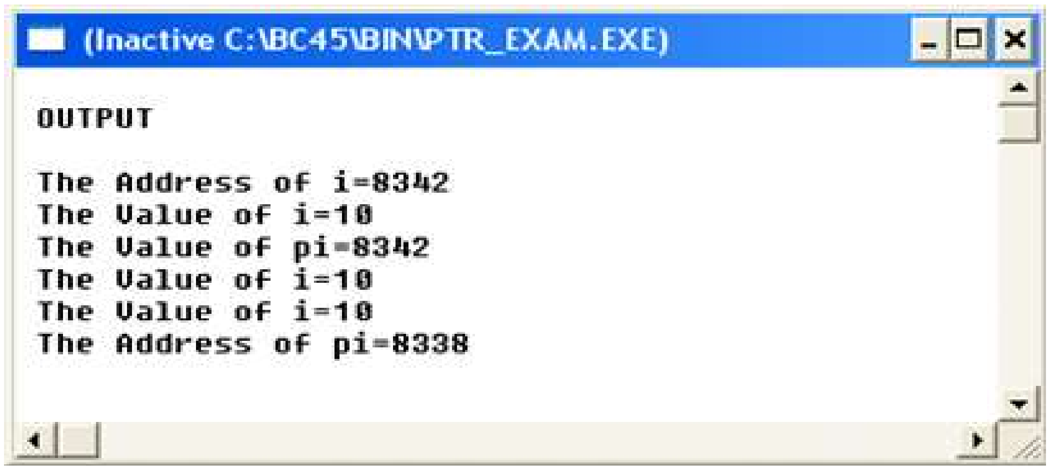
Access the value of a variable using pointer variable

Once a pointer variable has been assigned the address of a variable, we can access the value of a variable using the pointer. This is done by using the indirection operator**(*)**.

```
*ptr_name
```

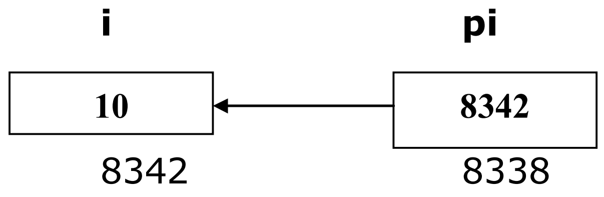
Example1

```
// C Program illustrating accessing of variables using pointers
#include<stdio.h>
int main()
{
  int i=10;
  int *pi;
  pi=&i;
  printf("\n OUTPUT\n");
  printf("\n The Address of i=%u",&i);
  printf("\n The Value of i=%d",i);
  printf("\n The Value of pi=%u",pi);
  printf("\n The Value of i=%d",*(&i));
  printf("\n The Value of i=%d",*pi);
  printf("\n The Address of pi=%u",&pi);
  return 0;
}
```



```
OUTPUT

The Address of i=8342
The Value of i=10
The Value of pi=8342
The Value of i=10
The Value of i=10
The Address of pi=8338
```

The above program illustrates how to access the variable using pointers. After finding the first statement i=10, the compiler creates a variable i with a value of 10 at a memory location. Then coming to line 2 and 3 a pointer variable pi is create and initialized with the address of the i variable. Then the compiler automatically provides a link between these two variables as follows.

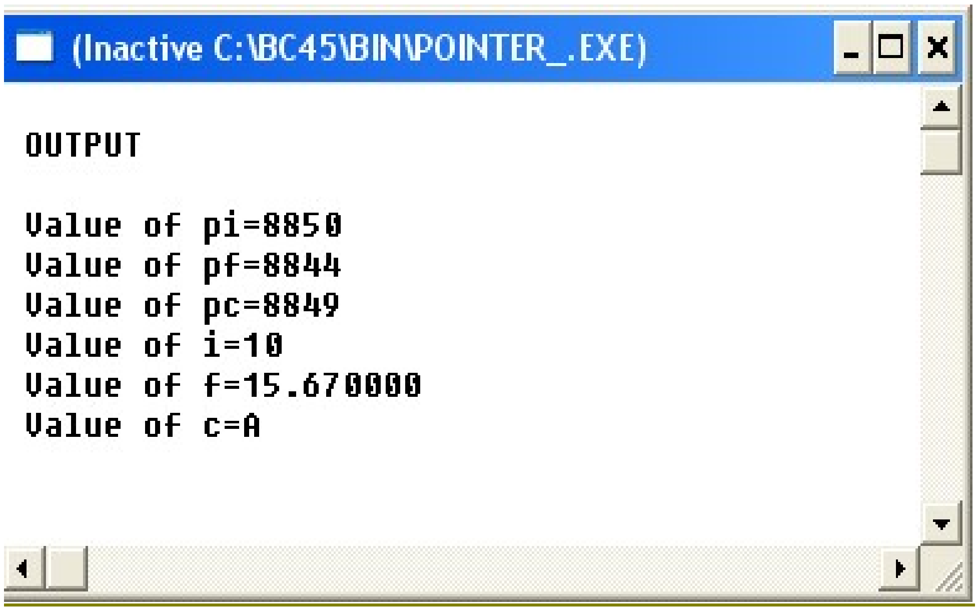


Note: Pointer variable always points to a address of the another variable .Following statements are not valid with respect to pointers.
int i=10, k, *pi=&i;
k=pi; // pointer value cannot be accessed by integer
pi=65506(constant); // we cannot directly assign a value to a pointer variable

Example2

The following code illustrates how to declare int, char and float pointers. Here we have declared three variables of type int, float and char ,also three pointer variables points to int, float and char. Remember here pf points to the value of type float but its type is unsigned integer only.

```
#include<stdio.h>
int main()
{
   int i=10;
   char c='A';
   float f=15.67;
   int *pi; /*Pointer to an integer */
   float *pf; /*pointer to a float number */
   char *pc; /*pointer to a character; */
   pi=&i;
   pf=&f; /*initialization of pointer variables */
   pc=&c;
   printf("\n OUTPUT \n");
   printf("\n Value of pi=%u",pi);
   printf("\n Value of pf=%u",pf);
   printf("\n Value of pc=%u",pc);
   printf("\n Value of i=%d",*pi);
   printf("\n Value of f=%f",*pf);
   printf("\n Value of c=%c",*pc);
   return 0;
}
```



```
OUTPUT

Value of pi=8850
Value of pf=8844
Value of pc=8849
Value of i=10
Value of f=15.670000
Value of c=A
```

Declaration versus Redirection:

❖ When an asterisk is used for declaration, it is associated with a type.

❖ Example: int* pa;
int* pb;

❖ On the other hand, we also use the asterisk for redirection.
When used for redirection, the asterisk is an operator that redirects the operation from the pointer variable to a data variable.

❖ Example: Sum = *pa + *pb;

Dangling Pointers

A pointer variable should contain a valid address. A pointer variable which does not contain a valid address is called dangling pointer.

For example, consider the following declaration,

```
int *pi;
```

This declaration indicates that pi is a pointer variable and the corresponding memory location should contain address of an integer variable. But, the declaration will not initialize the memory location and memory contains garbage value as shown in below.

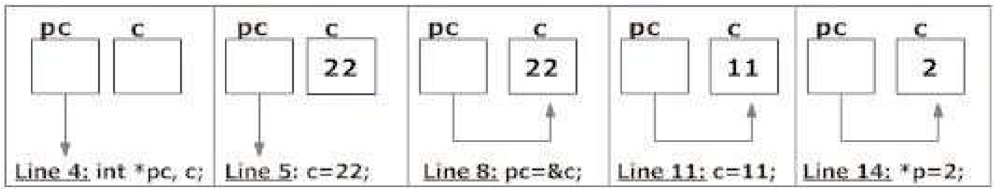
Note: We cannot use a pointer variable to the register variable. The reason is that, user does not know the address of the register variable. So we are not able to use pointer variable on register variables.

Example To Demonstrate Working of Pointers

```
/* Source code to demonstrate, handling of pointers in C program */
#include <stdio.h>
int main()
{
   int *pc,c;
   c=22;
   printf("Address of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
   pc=&c;
   printf("Address of pointer pc:%d\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   c=11;
   printf("Address of pointer pc:%d\n",pc);
   printf("Content of pointer pc:%d\n\n",*pc);
   *pc=2;
   printf("Address of c:%d\n",&c);
   printf("Value of c:%d\n\n",c);
   return 0;
}
```

Output

```
Address of c: 2686784
Value of c: 22

Address of pointer pc: 2686784
Content of pointer pc: 22

Address of pointer pc: 2686784
Content of pointer pc: 11

Address of c: 2686784
Value of c: 2
```



Explanation of program and figure

1. Code int *pc, p; creates a pointer *pc* and a variable *c*. Pointer *pc* points to some address and that address has garbage value. Similarly, variable *c* also has garbage value at this point.
2. Code c=22; makes the value of c equal to 22, i.e.,22 is stored in the memory location of variable *c*.
3. Code pc=&c; makes pointer, point to address of c. Note that, &c is the address of variable *c* (because *c* is normal variable) and *pc* is the address of *pc* (because pc is the pointer variable). Since the address of pc and address of c is same, *pc (value of pointer pc) will be equal to the value of *c*.
4. Code c=11; makes the value of *c*, 11. Since, pointer *pc* is pointing to address of *c*. Value of **pc* will also be 11.
5. Code *pc=2; change the address pointed by pointer *pc* to change to 2. Since, address of pointer *pc* is same as address of *c*, value of *c* also changes to 2.

Review Questions:

1. Write a C program to read in an array of integers. Instead of using subscripting, however, employ an integer pointer that points to the elements currently being read in and which is incremented each time.

2. Write a C program using pointers to read in an array of integers and print its elements in reverse order.

3. Write a C program to arrange the given numbers in ascending order using pointers.

4. Write a 'C' program to find factorial of a given number using pointers.

5. a) What is a pointer variable? How is a pointer variable different from an ordinary variable.
   b) Distinguish between address operator and dereferencing operator.
   c) Write a C Program to illustrate the use of indirection operator "*" to access the value pointed by a pointer.
6. Explain the uses of Pointers with an example.

Snippet Programs:

1.
```
#include<stdio.h>

int main()
{
   int i=3, *j, k;
   j = &i;
   printf("%d\n", i**j*i+*j);
   return 0;
}
```

2.
```
 #include<stdio.h>
int main()
{
   int x=30, *y, *z;
   y=&x; /* Assume address of x is 500 and integer is 4 byte */
   z=y;
   *y++=*z++;
   x++;
   printf("x=%d, y=%d, z=%d\n", x, y, z);
   return 0;
}
```

3.
```
 #include<stdio.h>
int main()
{
  char *str;
  str = "%s";
  printf(str, "K\n");
  return 0;
}
```

POINTERS AND FUNCTIONS

- Pointers can be used to pass addresses of variables to called functions, thus allowing the called function to alter the values stored there.
- We looked earlier at a swap function that did not change the values stored in the main program because only the values were passed to the function swap.
- This is known as "call by value".
- Here we are going to discuss how to pass the address.

Call by Reference

Instead of passing the values of the variables to the called function, we pass their addresses, so that the called function can change the values stored in the calling routine. This is known as "call by reference", since we are referencing the variables.

Here the addresses of actual arguments in the calling function are copied into formal arguments of the called function. Here The formal parameters should be declared as pointer variables to store the address.

The following shows the swap function modified from a "call by value" to a "call by reference". Note that the values are now swapped when the control is returned to main function.

```
#include <stdio.h>
void swap ( int *pa, int *pb ) ;
int main ( )
{
  int a = 5, b = 6;
  printf("a=%d b=%d\n",a,b) ;
  swap (&a, &b) ;
  printf("a=%d b=%d\n",a,b) ;
  return 0 ;
}
```

```
void swap( int *pa, int *pb )
{
  int temp;
  temp= *pa; *pa= *pb;  *pb = temp ;
  printf ("a=%d  b=%d\n", *pa, *pb);
}
Results:
   a=5  b=6
   a=6  b=5
   a=6  b=5
```

Observe the following points when the program is executed,

- The address of actual parameters a and b are copied into formal parameters pa and pb.
- In the function header of swap (), the variables a and b are declared as pointer variables.
- The values of a and b accessed and changed using pointer variables pa and pb.

Call by Value	***Call by Reference***
When Function is called the values of variables are passed.	When a function is called address of variables is passed.
Formal parameters contain the value of actual parameters.	Formal parameters contain the address of actual parameters.
Change of formal parameters in the function will not affect the actual parameters in the calling function	The actual parameters are changed since the formal parameters indirectly manipulate the actual parameters
Execution is slower since all the values have to be copied into formal parameters.	Execution is faster since only addresses are copied.

Table: 3.1 Difference between Call by Value and Call by Reference

FUNCTION RETURNING POINTERS

- ❖ The way function return an int, float and char, it can return a pointer.
- ❖ To make a function return a pointer it has to be explicitly mentioned in the calling function as well as in the function declaration.
- ❖ Three things should be done to avail the feature of functions return pointer.

1. Declaration of function returning pointer
2. Declaring pointer and assigning function call
3. Defining function returning pointer

- ❖ Syntax for declaration of function returning pointer

```
return_type *function_name (arguments);
```

This declaration helps the compiler to recognize that this function returns address.

- ❖ Now declare pointer variable and place the function call

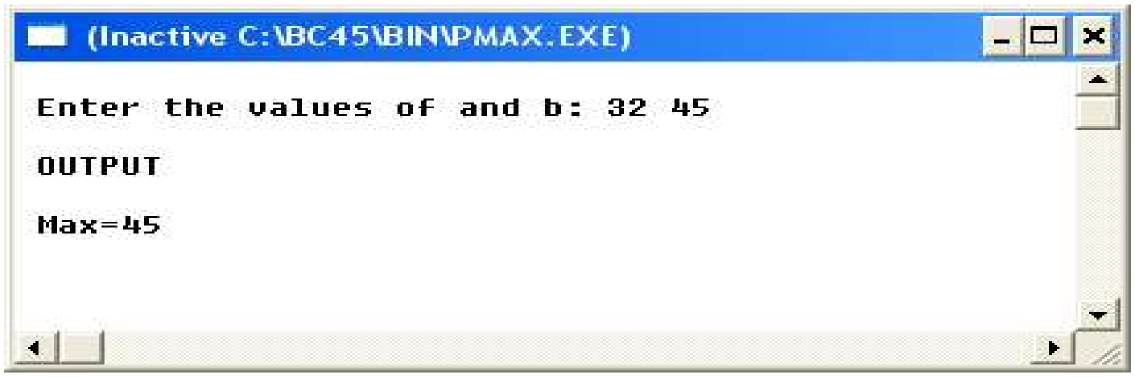
```
ptr = function_name (arguments);
```

- ❖ After executing above statement ptr consisting of the address that is returned by the function. Remember the return type of the function and pointer type should match here.
- ❖ The function Definition returning pointer takes of the form,

```
return_type *function_name (arguments)
{
  // local declarations
  // executable statements

  return (&variable); Here don't forget to send address
                with return statement.
}
```

Example:

```
#include<stdio.h>
int *max(int* ,int* );
int main()
{
   int a,b;
   int *ptr;
   printf("\n Enter the values of and b: ");
   scanf("%d%d",&a,&b);
   ptr=max(&a,&b);
   printf("\n OUTPUT \n");
   printf("\n Max=%d",*ptr);
   getch();
}
int *max(int*pa,int*pb)
{
   if(*pa>*pb)
    return pa;
   else
    return pb;
}
```



```
Enter the values of and b: 32 45

OUTPUT

Max=45
```

The execution of the program as follows,

- ❖ Execution of the program starts at main.
- ❖ Two variables and b are created and initialized at run-time.
- ❖ A pointer variable is created and initialized with the return value of the function max ().
- ❖ Once the control is transferred from function main () to max (), it got executed and returns the pointer value to main().
- ❖ Here we are having the address of the maximum variable address to display it just use indirection operator (*).

Note: function return pointer does not have any advantage except in the handling of strings.

POINTERS TO FUNCTIONS

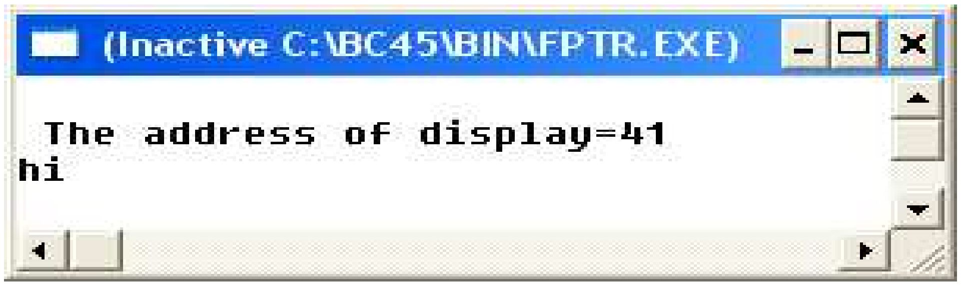
Pointer to a function (also known as function pointer) is a very powerful feature of C. Function pointer provides efficient and elegant programming technique. Function pointers are less error prone than normal pointers since we will never allocate or de-allocate memory for the functions.

Every variable with the exception of register has an address. We have seen how we can refer variables of type char, int and float. Through their addresses, by using pointers.

Functions exist in memory just like variables. C will allow you to define pointers to functions. Just like variables, a function name gives the starting address of function stored in memory.

The below code illustrate how to get the address of a function.

```
#include<stdio.h>

int main()
{
 void display();
 printf("\n The address of display=%u",display);
 getch();
}
void display()
{
  printf("hi");
}
```



```
 The address of display=41
hi
```

DEFINING POINTERS TO FUNCTIONS

Like declaring pointer variables, we can define pointers to function variables and store the address. The below figure illustrate how function pointer can be represented.

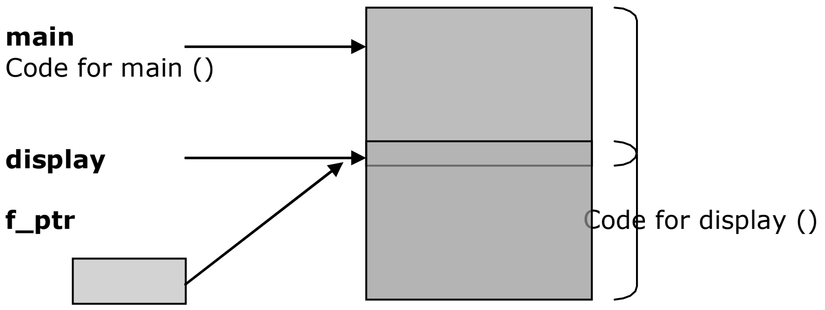


Figure: 3.3. Functions in Memory.

The syntax for declaring pointer to function as follows,

```
return_type (*f_ptr) (arguments);
```

Everything is same as function declaration except the braces for the name, to tell the compiler that this is a fuction pointer braces are required here and as usual for pointer declarations * is used.

Note that the return type of function pointer, number of arguments and type of arguments must match with the normal function.

The next after the declaration is calling the function using function pointer. before calling takes place we must initialize the function pointer with the address of the function.

The syntax for this assignment,

```
f_ptr=function_name;
```

After this assignment we need to call the function, the syntax associated with the function call is as follows,

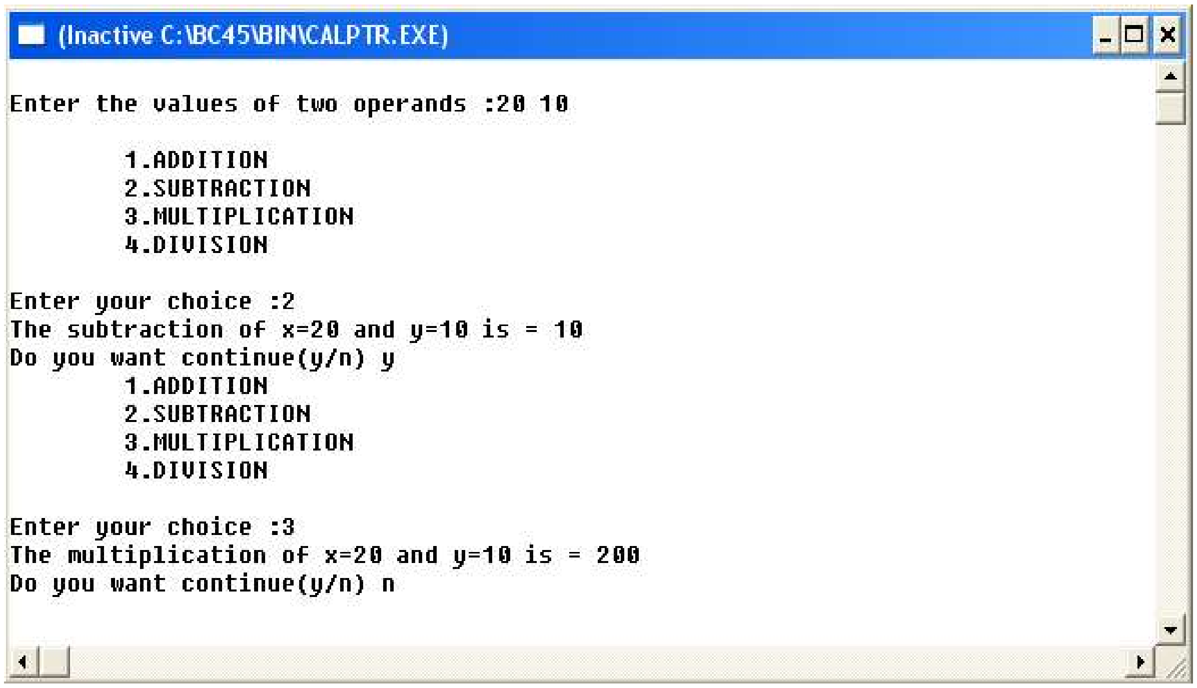
```
(*f_ptr)(argument's);
```

This is another way of calling the function. There are no changes in the declaration of the function body.

The below program simulates a simple calculator using function pointers.

```
#include<stdio.h>

int add(int* ,int* );
int sub(int* ,int* );
int mul(int* ,int* );
float div(int* ,int* );
int (*fadd)(int* ,int* );
int (*fsub)(int* ,int* );
int (*fmul)(int* ,int* );
float (*fdiv)(int* ,int* );

int main()
{
  int x,y,ch;
  char c;
  fadd=add;
  fsub=sub;
  fmul=mul;
  fdiv=div;
  printf("\nEnter the values of two operands :");
  scanf("%d%d",&x,&y);
  printf("\n");
  do
  {
    printf("        1.ADDITION        \n");
    printf("        2.SUBTRACTION     \n");
    printf("        3.MULTIPLICATION  \n");
    printf("        4.DIVISION        \n");
    printf("\nEnter your choice :");
    scanf("%d",&ch);
    switch(ch)
    {
      case 1:
```

```
    printf("The addition of x=%d and y=%d is = %d",x,y,(*fadd)(&x,&y));
    break;
  case 2:
   printf("The subtraction of x=%d and y=%d is = %d",x,y,(*fsub)(&x,&y));
   break;
  case 3:
   printf("The multiplication of x=%d and y=%d is = %d",x,y,(*fmul)(&x,&y));
   break;
  case 4:
   printf("The division of x=%d and y=%d is = %f",x,y,(*fdiv)(&x,&y));
   break;
}
printf("\nDo you want continue(y/n) ");
fflush(stdin);
scanf("%c",&c);
   }while(c=='y'||c=='Y');
   getch();
 }

int add(int *a,int *b)
 {
  return(*a+*b);
 }

int sub(int *a,int *b)
 {
  return(*a-*b);
 }
int mul(int *a,int *b)
 {
  return(*a * (*b));
 }

float div(int *a,int *b)
 {
  return(*a/(*b));
 }
```



```
Enter the values of two operands :20 10

        1.ADDITION
        2.SUBTRACTION
        3.MULTIPLICATION
        4.DIVISION

Enter your choice :2
The subtraction of x=20 and y=10 is = 10
Do you want continue(y/n) y
        1.ADDITION
        2.SUBTRACTION
        3.MULTIPLICATION
        4.DIVISION

Enter your choice :3
The multiplication of x=20 and y=10 is = 200
Do you want continue(y/n) n
```

Review Questions:

1. Write a ‘C’ program to find factorial of a given number using pointers.

2. a) How to use pointers as arguments in a function? Explain through an example.
   b) Write a ‘C’ function using pointers to exchange the values stored in two locations in the memory.

3. Explain pointer to function and function returning pointer with example.

Snippet Programs:

```
1.     #include<stdio.h>
       void fun(void *p);
       int i;

       int main()
       {
           void *vptr;
           vptr = &i;
           fun(vptr);
           return 0;
       }
       void fun(void *p)
       {
           int **q;
           q = (int**)&p;
           printf("%d\n", **q);
       }
```

2. #include<stdio.h>

```
int *check(static int, static int);

int main()
{
    int *c;
    c = check(10, 20);
    printf("%d\n", c);
    return 0;
}
int *check(static int i, static int j)
{
    int *p, *q;
    p = &i;
    q = &j;
    if(i >= 45)
        return (p);
    else
        return (q);
}
```

POINTER ARITHMETIC

The following operations can be performed on a pointer:

- ❖ Addition of a number to a pointer. Pointer can be incremented to point to the next locations.

Example:
int i=4 ,pi=&i; //(assume address of i=1000)
float j,*pj=&j;// (assume address of j=2000)
pi = pi + 1; // here pi incremented by (1*data type times)
pi = pi + 9; // pi = 1000 + (9*2) → 1018 address
pj = pj + 3; // pj=1018+(3*4)→1030 address

- ❖ Subtraction of a number from a pointer. Pointer can be decremented to point to the earlier locations.

Example:
int i=4,*pi=&i; //assume address of i =1000)
char c, *pc=&c; // assume address of c = 2000
double d, *pd=&d; // assume address of d=3000
pi = pi-2; /* pi=1000-(2*2)=996 address */
pc = pc-5; /* pc=2000-(5*1)=1985 address
pd = pd-6; /* pd=3000-(6*8)=2952 address */

- ❖ Pointer variables may be subtracted from one another. This is helpful while finding array boundaries. Be careful while performing subtraction of two pointers.
- ❖ Pointer variables can be used in comparisons, but usually only in a comparison to **NULL**.
- ❖ We can also use increment/decrement operators with pointers this is performed same as adding/subtraction of integer to/from pointer.

The following operations cannot be performed on pointers.

- ❖ Addition of two pointers.
- ❖ Multiplication of a pointer with a constant, two pointers.
- ❖ Division of a pointer with a constant, two pointers.

POINTER EXPRESSIONS

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

```
a= *p1 + *p2;
sum = sum + *p1;
z = 10 / *p2;
f = *p1 * i;
```

Note: be careful while writing pointer expressions .The expression ***p++** will result in the increment of the address of p by data type times and points to the new value. Whereas the expression **(*p) ++** will increments the vale at the address. If you are not properly coded you will get some unwanted result.

NULL Pointer

- If wish to have a pointer that points to "nowhere", should make this explicit by assigning it to NULL.
- If it is too early in the code to assign a value to a pointer, then it is better to assign NULL (i.e., \0 or 0).

```
double *pval1 = NULL;
double *pval2 = 0;
```

- The integer constants 0 and 0L are valid alternatives to NULL, but the symbolic constant is (arguably) more readable.
- A NULL pointer is defined as a special pointer.
- It can be used along with memory management functions.

POINTERS TO POINTERS

- It is possible to make a pointer to point to another pointer variable. But the pointer must be of a type that allows it to point to a pointer.
- A variable which contains the address of a pointer variable is known as **pointer to** pointer.
- Its major application is in referring the elements of the two dimensional array.
- Syntax for declaring pointer to pointer,

```
data type **ptr_ptr;
```

- This declaration tells compiler to allocate a memory for the variable **ptr_ptr** in which address of a **pointer variable** which points to value of type data type can be stored.
- Syntax for initialization

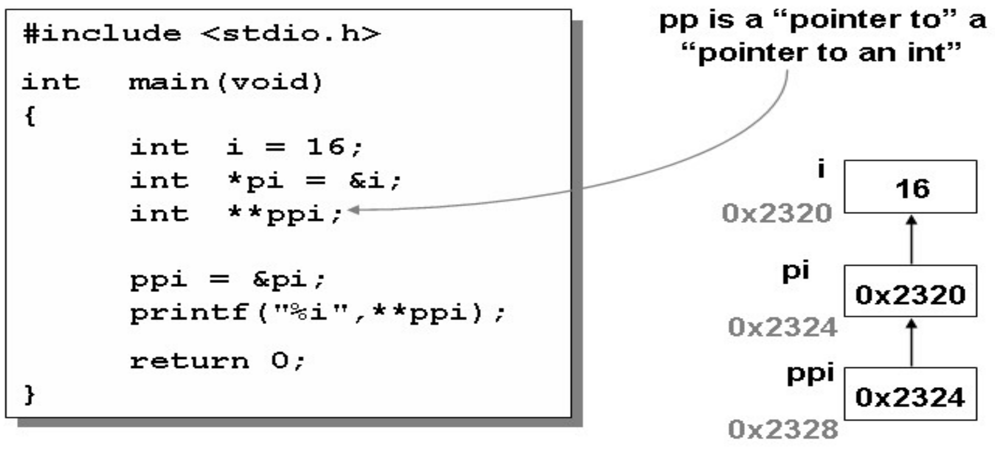
```
ptr_ptr=&ptr_name;
```

- This initialization tells the compiler that now ptr_ptr points to the address of a pointer variable.
- Accessing the element value,

```
**ptr_ptr;
```

- It is equalent to *(*(&ptr_name));

Example



```
#include <stdio.h>

int   main(void)
{
        int   i = 16;
        int   *pi = &i;
        int   **ppi;

        ppi = &pi;
        printf("%i",**ppi);

        return 0;
}
```

The above program illustrates the use of pointers to pointers. Here, using two indirection operators the data item 16 can be accessed (i.e., *ppi refers to pi and **ppi refers to i).

POINTER COMPATIBILITY

- ❖ We should not store the address of a data variable of one type into a pointer variable of another type.
- ❖ During assigning we should see that the type of data variable and type of the pointer variable should be same or compatible.
- ❖ Other wise it will result in unwanted output.
- ❖ The following program segment is wrong,

```
int i=10;

float *pf;

pf=&i; // data variable is integer and pointer
variable is float
```

It is possible to use incompatible pointer types while assigning with type casting pointer.

Casting pointers

When assigning a memory address of a variable of one type to a pointer that points to another type it is best to use the cast operator to indicate the cast is intentional (this will remove the warning).

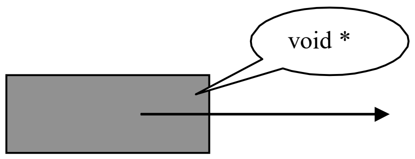
Example:

```
int V = 101;
float *P = (float *) &V; /* Casts int address to float * */
```

Removes warning, but is still a somewhat unsafe thing to do.

Void Pointer

- ❖ A pointer to void is a generic type that is not associated with a reference type.
- ❖ It is neither the address of a character nor an integer, nor a float nor any other type.
- ❖ It is compatible for assignment purposes only with all other pointer types.
- ❖ A pointer of any reference type can be assigned to a pointer to void type.
- ❖ A pointer to void type can be assigned to a pointer of any reference type.
- ❖ Certain library functions return void * results.
- ❖ No cast is needed to assign an address to a void * or from a void * to another pointer type.
- ❖ Where as a pointer to void can not be deferenced unless it is cast.



void
Figure 3.2 pointer to void

- ❖ Example:

```
int V = 101;
float f=98.45;
void *G = &V;      /* No warning */
printf ("%d",*((int*)G)); /* Now it will display 101
float *P = G;       /* No warning, still not safe */
printf ("%f",*((float*)G)); /* Now it will display 98.45
```

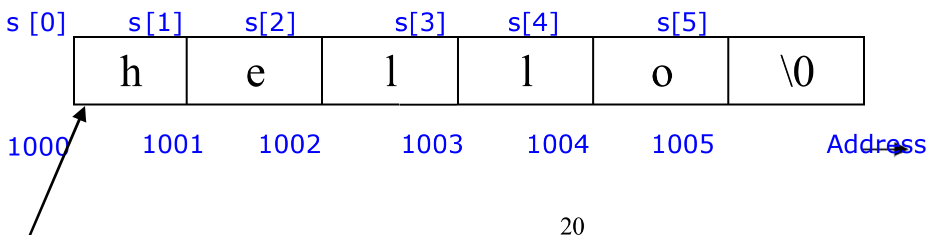
CHARACTER POINTER

Suppose we wish to store "Hello". We may either store it in a string or we may ask the C compiler to store it at some location in memory and assign the address of the string in a char pointer.

Consider the following declaration with string initialization,

char *p="hello";

Here the string length is 5 bytes. So the compiler allocates 6 bytes memory locations. Probably the characters are stored in the constant memory area of the computer, and the pointer p points to the base address as shown in the below figure 3.13.



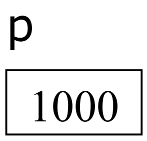


Figure 3.13: Initializing using Character Pointer

- We cannot assign a string to another. But, we can assign a char pointer to another char pointer.

Example: char *p1=”hello”;
char *p2;
p1=p2; //valid
printf (“%s”, p1); //will print hello

- *p will refer to a particular character only, p will refer whole string.

POINTERS AND STRINGS

A string is a character array. so the name of the string it self is a pointer. Like referring array elements also string characters also refereed with its name.

Example:
char s [] =”hello”;

The following notations are used to refer individual characters
s[i] --> *(s+i) --> *(i+ s) all will point to the ith character in the given string.
We can also use pointer variable to refer the string elements.

char s [ ]=”hello”;
char *ptr;
ptr=s; // now ptr points to the base address of the string.

then the following notations are same,

***ptr --> *(ptr+i) --> *(i+ptr)** will point value at the ith character.

Example: // illustrates displaying characters using pointer

```
#include<stdio.h>
void main ()
{
char s [] =”hello”;
char *ptr;
ptr=s;
while (*ptr! =’\0’)
{
printf (“ %c”,*ptr);
ptr++;
}
```

OUTPUT

```
h e l l o
```

Review Questions:

1. Write short notes on pointer to void.
2. Write short notes on Address Arithmetic.
3. Explain pointer to pointer with an example.

Snippet Programs:

1.
```
#include<stdio.h>
int main()
{
    static char *s[] = {"black", "white", "pink", "violet"};
    char **ptr[] = {s+3, s+2, s+1, s}, ***p;
    p = ptr;
    ++p;
    printf("%s", **p+1);
    return 0;
}
```

2.
```
#include<stdio.h>
int main()
{
    char str[20] = "Hello";
    char *const p=str;
    *p='M';
    printf("%s\n", str);
    return 0;
}
```

3.
```
#include<stdio.h>
int main()
{
    int ***r, **q, *p, i=8;
    p = &i;
    q = &p;
    r = &q;
    printf("%d, %d, %d\n", *p, **q, ***r);
    return 0;
}
```

POINTERS AND ARRAYS

- An array is a collection of similar elements stored in contiguous memory locations.
- When an array is declared, the compiler allocates a base address and sufficient amount of memory depending on the size and data type of the array.
- The base address is the location of the first element of the array.
- The compiler defines the array name as a constant pointer to the first element.

POINTERS AND ONE DIMENSIONAL ARRAY

Let us take the following declaration,

int num [5] = {1, 2, 3, 4, 5};

- ❖ After having this declaration, the compiler creates an array with name **num**, the elements are stored in contiguous memory locations, and each element occupies two bytes, since it is an integer array.

- ❖ The name of the array **num** gets the base address. Thus by writing *num we would be able to refer to the zeroth element of the array, that is 1.
- ❖ Then *num and *(num+0) both refer to the element 1.and *(num+2) will refer 3.
- ❖ When we have num[i] , the compiler internally converts it to *(num+i).
- ❖ In this light the following notations are same.

$$\boxed{num[i] \rightarrow *(num+i) \rightarrow *(i+num) \rightarrow i\ [num]}$$

- ❖ Then we can also define a pointer and initialize it to the address of the first element of the array (base address).
- ❖ Example, for the above array we can have the following statement,

int *ptr=a; (or) int *ptr=&a[0];

- ❖ To refer the array elements by using pointer the following notations are used.

$$\boxed{*ptr \rightarrow *(ptr+i) \rightarrow *(i+ptr) \rightarrow i\ \ [ptr]}$$

- ❖ p++ will point to the next location in the array.
- ❖ Accessing array elements by using pointers is always faster than accessing them by subscripts.
- ❖ The below figure shows the array element storage in the memory.

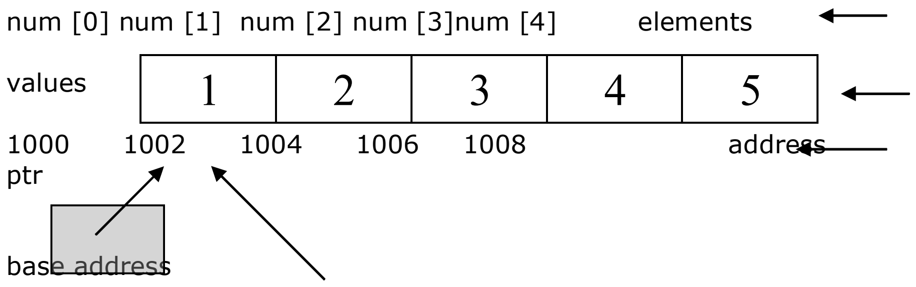
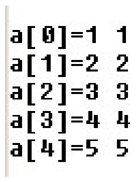


Figure 3.4 Storage representation of array

Example

```
#include<stdio.h>
int main()
{
        int num[5]={1,2,3,4,5};
        int i,*ptr;
        ptr=num;
        for(i=0;i<5;i++)
        {
          printf("\na[%d]=%d %d",i,*ptr,*(num+i));
          ptr++;
        }
        return 0;
}
```



```
a[0]=1 1
a[1]=2 2
a[2]=3 3
a[3]=4 4
a[4]=5 5
```

The above program illustrates displaying the array elements using pointers.

Note: Note that the array name num is a constant pointer points to the base address, then the increment of its value is illegal, num++ is invalid.

POINTERS AND TWO DIMENSIONAL ARRAYS

A two dimensional array is an array of one dimensional arrays. The important thing to notice about two-dimensional array is that, just as in a one-dimensional array, the name of the array is a pointer constant the first element of the array, however in 2-D array, the first element is another array.

Let us consider we have a two-dimensional array of integers. When we dereference the array name, we don't get one integer, we get an array on integers. In other words the dereference of the array name of a two-dimensional array is a pointer to a one-dimensional array. Here we require two indirections to refer the elements

Let us take the declaration

int a [3][4];

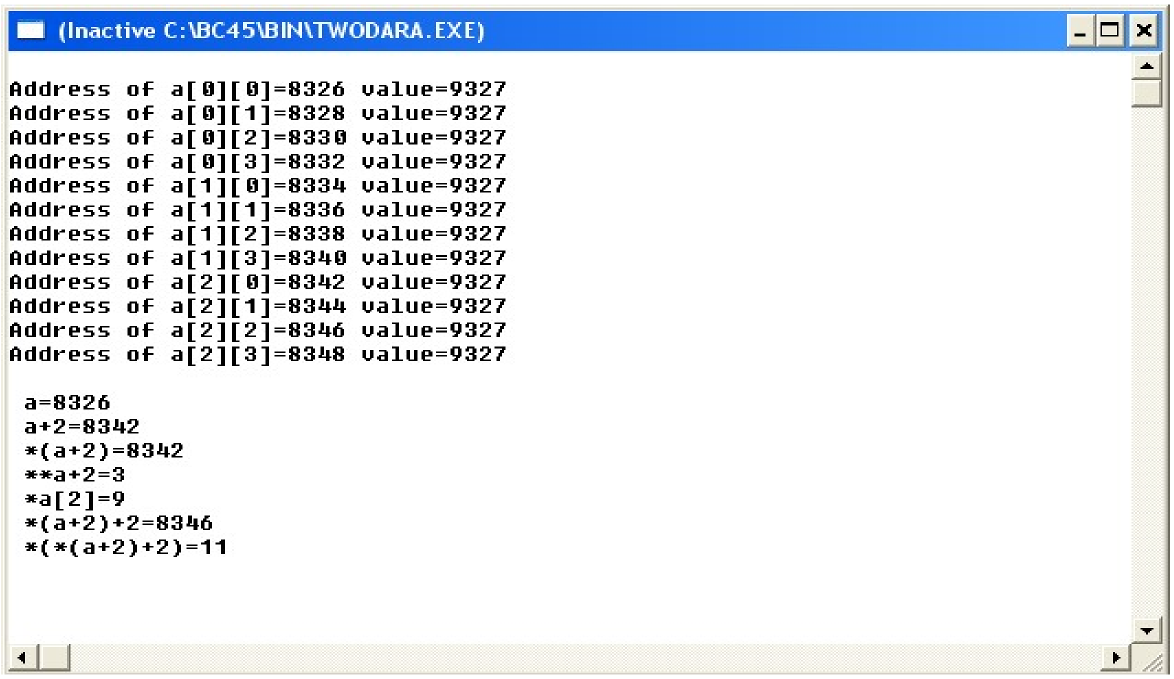
Then following notations are used to refer the two-dimensional array elements,

a------------------- > points to the first row
a+i ---------------- > points to i^{th} row
***(a+i) ------------ > points to first element in the i^{th} row**
***(a+i) +j--------- > points to j^{th} element in the i^{th} row**
***(*(a+i)+j)---- >value stored in the i^{th} row and j^{th} column**

	0	1	2	3
0	1	2	3	4
1	5	6	7	8
2	9	10	11	12

Example

```
#include<stdio.h>
int main()
{
     int a[3][4]={
                       {1,2,3,4},
                       {5,6,7,8},
                       {9,10,11,12}
                    };
     int i,j;
     for(i=0;i<3;i++)
     {
      for(j=0;j<4;j++)
      {
     printf("\nAddress of a[%d][%d]=%u value=%d",i,j,*(a+i)+j,*(*(a+i)+j));
      }
     }
     printf("\n");
     printf("\n a=%u",a);
     printf("\n a+2=%u",a+2);
     printf("\n *(a+2)=%u",*(a+2));
     printf("\n *a[2]=%d",*a[2]);
     printf("\n *(a+2)+2=%u",*(a+2)+2);
     printf("\n *(*(a+2)+2)=%d",*(*(a+2)+2));
     getch();
}
```



(Inactive C:\BC45\BIN\TWODARA.EXE)

```
Address of a[0][0]=8326 value=9327
Address of a[0][1]=8328 value=9327
Address of a[0][2]=8330 value=9327
Address of a[0][3]=8332 value=9327
Address of a[1][0]=8334 value=9327
Address of a[1][1]=8336 value=9327
Address of a[1][2]=8338 value=9327
Address of a[1][3]=8340 value=9327
Address of a[2][0]=8342 value=9327
Address of a[2][1]=8344 value=9327
Address of a[2][2]=8346 value=9327
Address of a[2][3]=8348 value=9327

 a=8326
 a+2=8342
 *(a+2)=8342
 **a+2=3
 *a[2]=9
 *(a+2)+2=8346
 *(*(a+2)+2)=11
```

POINTERS AND THREE DIMENSIONAL ARRAYS

Three-dimensional array can be thought of array of two-dimensional array. To refer the elements here we require tree indirections.

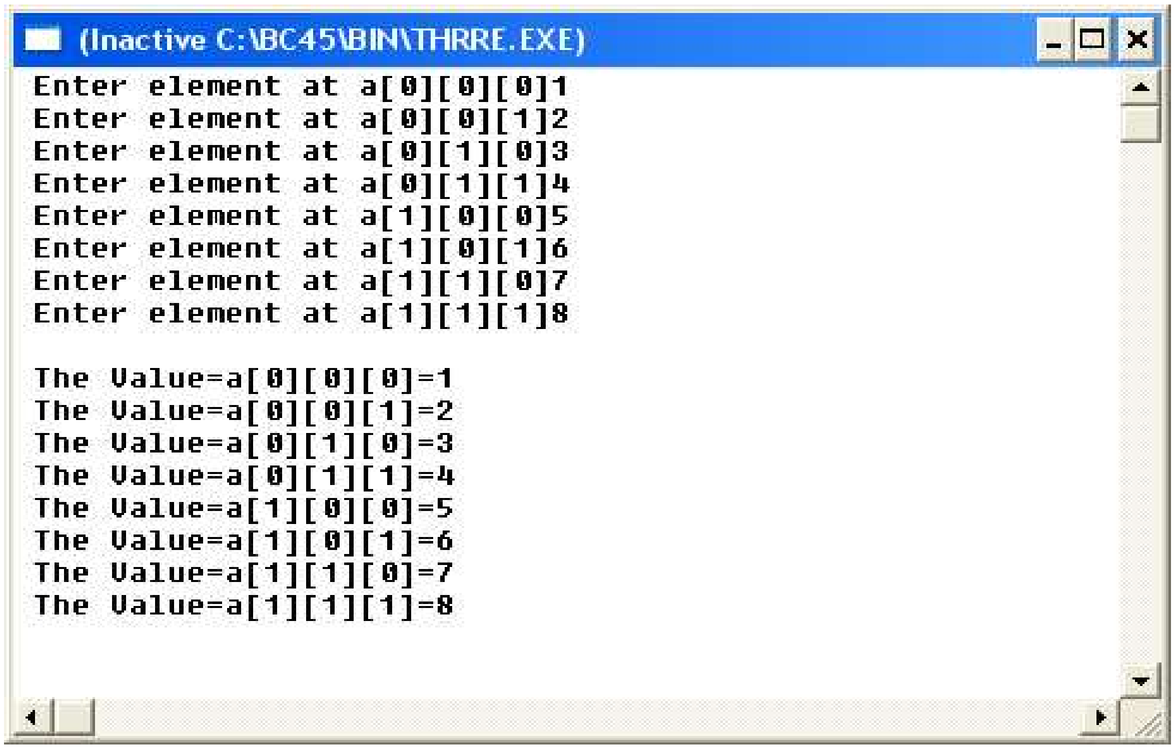
The notations are,

***(*(a+i) +j) +k** --> points to the address of kth dimension in ith row jth column

***(*(*(a+i) +j) +k)** --> value stored at kth dimension ith row jth column

Example

```
#include<stdio.h>
int main()
{
 int a[2][2][2],i,j,k;
 for(i=0;i<2;i++)
 {
  for(j=0;j<2;j++)
  {
   for(k=0;k<2;k++)
   {
    printf(" Enter element at a[%d][%d][%d]",i,j,k);
    scanf("%d",*(*(a+i)+j)+k);
   }
  }
 }
 for(i=0;i<2;i++)
 {
  for(j=0;j<2;j++)
  {
   for(k=0;k<2;k++)
   {
    printf("\n The Value=a[%d][%d][%d]=%d",i,j,k,*(*(*(a+i)+j)+k));
   }
  }
 }
 getch();
}
```



```
Enter element at a[0][0][0]1
Enter element at a[0][0][1]2
Enter element at a[0][1][0]3
Enter element at a[0][1][1]4
Enter element at a[1][0][0]5
Enter element at a[1][0][1]6
Enter element at a[1][1][0]7
Enter element at a[1][1][1]8

The Value=a[0][0][0]=1
The Value=a[0][0][1]=2
The Value=a[0][1][0]=3
The Value=a[0][1][1]=4
The Value=a[1][0][0]=5
The Value=a[1][0][1]=6
The Value=a[1][1][0]=7
The Value=a[1][1][1]=8
```

FUNCTIONS AND ARRAYS

To process arrays in large program, we have to be able to pass them to function. We can pass arrays in two ways,

- Passing individual elements
- Passing entire array elements

Passing individual elements

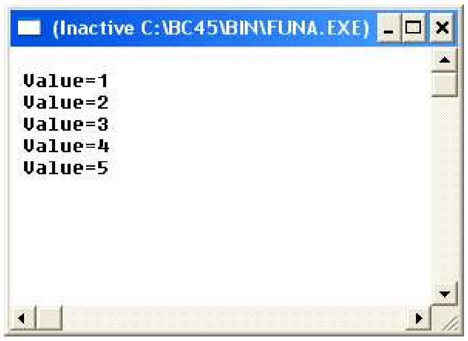
We can pass individual elements by either their data values or by passing their addresses.

1. Passing Data Values

- This is same as passing data values. Here we are passing an individual array element at a time tp the function.
- The called function cannot tell whether the value it receives comes from an array, or a variable.

Example

```
#include<stdio.h>
void fun(int );
int main()
{
 int a[5]={1,2,3,4,5};
 int i;
 for(i=0;i<5;i++)
 {
  fun(a[i]);
 }
 return 0;
}
void fun(int x)
{
 printf("\n Value=%d",x);
}
```



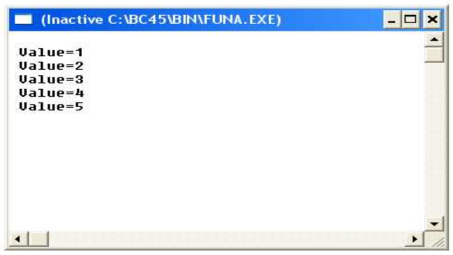
```
Value=1
Value=2
Value=3
Value=4
Value=5
```

2. Passing Addresses of array elements

Here we are passing the individual elements address. the called function requires declaration of arguments as pointer variables.

Example

```
#include<stdio.h>
void fun(int* );
int main()
{
 int a[5]={1,2,3,4,5};
 int i;
 for(i=0;i<5;i++)
 {
  fun(&a[i]);
 }
 return 0;
}
void fun(int *px)
{
 printf("\n Value=%d",*px);
}
```



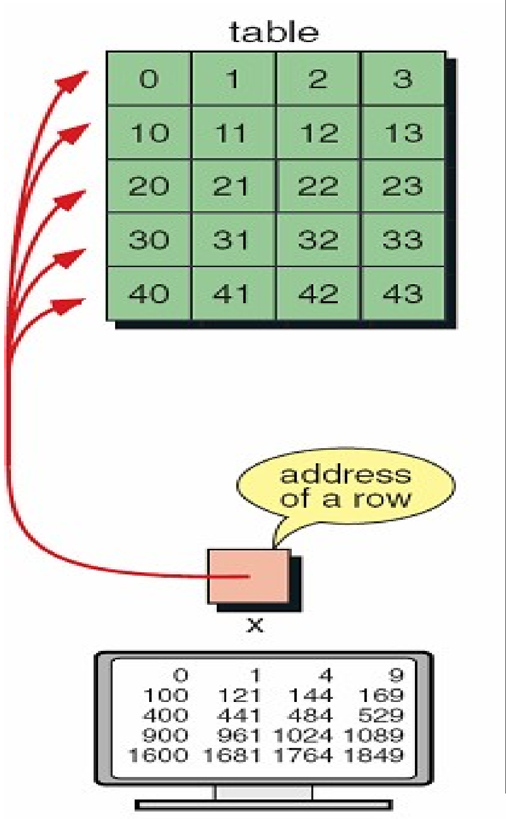
```
Value=1
Value=2
Value=3
Value=4
Value=5
```

Passing the Entire Array

Here we see the first situation in which C does not pass values to a function. The reason for this change is that a lot of memory and time would be used in passing large arrays every time we wanted to use one in function.

For example, if an array containing 1200 elements were passed by value to a function, another 1200 elements would have to be allocated in the called function and each element would be copied from one array to the other. So, instead of passing the whole array, C passes the address of the array.

In C, the name of the array value is address of the first element in the array. When we pass the name, it allows the called function to refer to the array back in the calling function. Passing both two dimensional array and one dimensional are same but subscripts are changed.



0	1	2	3
10	11	12	13
20	21	22	23
30	31	32	33
40	41	42	43

```
   0     1     4    9
 100   121   144  169
 400   441   484  529
 900   961  1024 1089
1600  1681  1764 1849
```

```
#include<stdio.h>
void square (int [] [4]);
int main ()
{

  int a[5][4]={
                    {0, 1, 2, 3},
                    {10, 11, 12, 13},
                    {20, 21, 22, 23},
                    {30, 31, 32, 33},
                    {40, 41, 42, 43}
                    };
  square(a);
  return 0;
}
void square(int x[5][4])
{
  int i,j;
for(i=0;i<5;i++)
{
 for(j=0;j<4;j++)
{
  printf ("\t%d",x[i][j]*x[i][j]);
}
printf("\n");
}
}
```

The above program find the squares of the elements of the array .Here we passed the name of the array as an argument to the function an in the called function the formal argument is created and points to the elements of the calling function argument array.

POINTERS AND STRINGS

A string is a character array. so the name of the string it self is a pointer. Like referring array elements also string characters also refereed with its name.

Example:

char s [] =”hello”;

The following notations are used to refer individual characters
s[i] --> *(s+i) --> *(i+ s) all will point to the ith character in the given string.
We can also use pointer variable to refer the string elements.

char s [ ]=”hello”;
char *ptr;
ptr=s; // now ptr points to the base address of the string.

then the following notations are same,

***ptr --> *(ptr+i) --> *(i+ptr)** will point value at the ith character.

ARRAY OF POINTERS

- A pointer variable always contains an address; an array of pointers would be nothing but a collection of addresses.
- The addresses present in the array of pointers can be addresses of variables or addresses of array elements or any other addresses.
- The major application of this is in handling data tables efficiently and table of strings.
- All rules that apply to an ordinary array apply to the array of pointes as well.
- The Syntax for declaration of array of pointers as follows,

data type *arr_ptr [size];

- This declaration tells the compiler arr_ptr is an array of addresses, pointing to the values of data type.
- Then initialization can be done same as array element initialization. Example **arr_ptr [3] =&var**, will initialize 3rd element of the array with the address of var.
- The dereferencing operator is used as follows

***(arr_ptr [index])** --> will give the value at particular address.

- Look at the following code **array of pointers to ordinary Variables**

Example

```
#include<stdio.h>
void main ()
{
   int i=1, j=2, k=3, l=4, x;
   int *arr [4];        //declaration of array of pointers
   arr [0] =&i;         //initializing 0th element with address of i
   arr [1] =&j;         //initializing 1st  element with address of j
   arr [2] =&k;         //initializing 2nd element with address of k
   arr [3] =&l;         //initializing 3rd element with address of l
   for (x=0; x<4; x++)
        printf ("\n %d",*(arr[x]));
}
```

OUTPUT

```
1
2
3
4
```

Here, arr contains the addresses of int variables i, j, k and l. The for loop is used to print the values present at these addresses.

A two-dimensional array can be represented using pointer to an array. But, a two-dimensional array can be expressed in terms of array of pointers also.

Using array of pointers a two dimensional array can be defined as,

data type *arr_ptr [size];

where data type refers to the data type of the array. arr_ptr refers to the name of the array and size is the maximum number of elements in the row.

Example int *arr [3];

Here, p is an array of pointers, and then following notations are used to refer elements.

p [i] --> points the address of the element ith row,
p[i] +j --> points the address of the element ith row and jth column
***(p[i] +j) --> value at ith row and jth column.**

Array of pointers and Strings

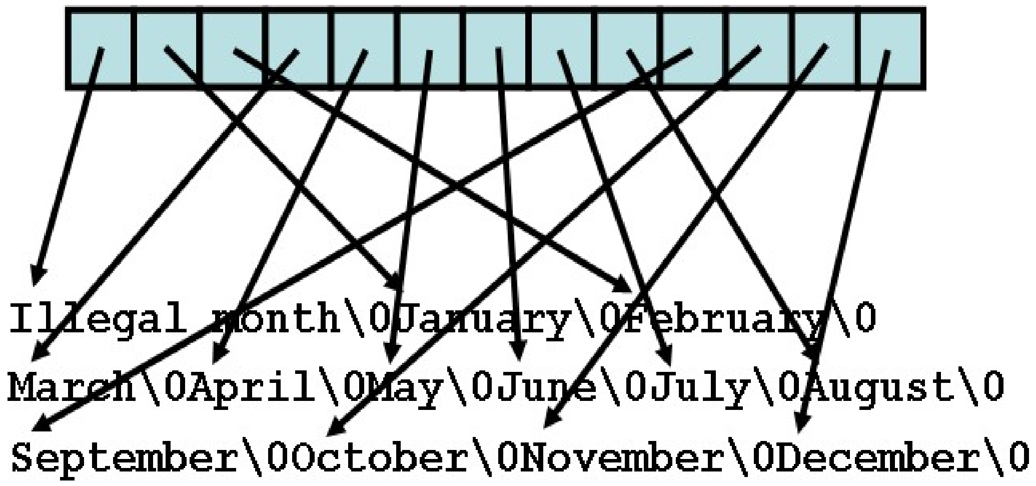
- ❖ Each element of the array is a pointer to a data type (in this case character).
- ❖ A block of memory (probably in the constant area) is initialized for the array elements.
- ❖ Declaration:

char *names [10]; // array of 10 character pointers.

- ❖ In this declaration names [] is an array of pointers. It contains base address of respective names. That is, base address of first name is stored in name [0] etc., a character pointer etc.
- ❖ The main reason to store array of pointers is to make more efficient use of available memory.

```
# include <stdio.h>
int main ()
{
char *name [] = {
"Illegal month",
"January", "February", "March",
"April", "May", "June",
"July", "August", "September",
"October", "November", "December"
};
}
```

The pointers are initialized like so


Illegal month\0January\0February\0
March\0April\0May\0June\0July\0August\0
September\0October\0November\0December\0

Note: When we are using an array of pointers to strings we can initialize the string at the place where we are declaring the array, but we cannot receive the string from keyword using scanf ().

Review Questions:

4. Write a C program to read in an array of integers. Instead of using subscripting, however, employ an integer pointer that points to the elements currently being read in and which is incremented each time.

5. Write a program to reverse the strings stored in array of pointers.

6. Write a program to find rank of a matrix using pointers.

7. Write a C program using pointers to read in an array of integers and print its elements in reverse order.

8. Write a 'C' Program to compute the sum of all elements stored in an array using pointers.

9. Write a C program to find the desired element in an array of N elements. Use pointers to search an element.

10. a) Explain how strings can be stored using a multidimensional arrays?
    b) What are the string in-built functions available? Write in detail about each one of them with an example.

11. What is the purpose of array of pointers? illustrate with an example.

12. Distinguish between array of pointers and pointer to array with examples.

Snippet Programs:

1.
```
#include<stdio.h>
int main()
{
    int arr[2][2][2] = {10, 2, 3, 4, 5, 6, 7, 8};
    int *p, *q;
    p = &arr[1][1][1];
    q = (int*) arr;
    printf("%d, %d\n", *p, *q);
    return 0;
}
```

2.
```
#include<stdio.h>
int main()
{
    int a[3][4] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 };
    printf("%u, %u, %u\n", a[0]+1, *(a[0]+1), *(*(a+0)+1));
    return 0;
}
```

3.
```
#include<stdio.h>
int main()
{
    int arr[3] = {2, 3, 4};
    char *p;
    p = arr;
    p = (char*)((int*)(p));
    printf("%d, ", *p);
    p = (int*)(p+1);
    printf("%d", *p);
    return 0;
}
```

Dynamic Memory Allocation

- ✓ The exact size of array is unknown until the compile time, i.e., time when a compiler compiles code written in a programming language into a executable form. The size of array you have declared initially can be sometimes insufficient and sometimes more than required.
- ✓ Dynamic memory allocation allows a program to obtain more memory space, while running or to release space when no space is required.
- ✓ Although, C language inherently does not have any technique to allocated memory dynamically, there are 4 library functions under "stdlib.h" for dynamic memory allocation.

Function	Use of Function
malloc()	Allocates requested size of bytes and returns a pointer first byte of allocated space
calloc()	Allocates space for an array elements, initializes to zero and then returns a pointer to memory
free()	deallocate the previously allocated space
realloc()	Change the size of previously allocated space

malloc()

The name malloc stands for "memory allocation". The function malloc() reserves a block of memory of specified size and return a pointer of type void which can be casted into pointer of any form.

Syntax of malloc()

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

Here, *ptr* is pointer of cast-type. The malloc() function returns a pointer to an area of memory with size of byte size. If the space is insufficient, allocation fails and returns NULL pointer.

```
ptr=(int*)malloc(100*sizeof(int));
```

This statement will allocate either 200 or 400 according to size of int 2 or 4 bytes respectively and the pointer points to the address of first byte of memory.

calloc()

The name calloc stands for "contiguous allocation". The only difference between malloc() and calloc() is that, malloc() allocates single block of memory whereas calloc() allocates multiple blocks of memory each of same size and sets all bytes to zero.

Syntax of calloc()

ptr=(cast-type*)calloc(n,element_size);

This statement will allocate contiguous space in memory for an array of *n* elements. For example:

```
ptr=(float*)calloc(25,sizeof(float));
```

This statement allocates contiguous space in memory for an array of 25 elements each of size of float, i.e, 4 bytes.

Differences between malloc() and Calloc():

S.no	malloc	calloc
1	It allocates only single block of requested memory	It allocates multiple blocks of requested memory
2	int *ptr;ptr = malloc(20 * sizeof(int));For the above, 20*4 bytes of memory only allocated in one block. Total = 80 bytes	int *ptr;Ptr = calloc(20, 20 * sizeof(int));For the above, 20 blocks of memory will be created and each contains 20*4 bytes of memory. Total = 1600 bytes
3	malloc () doesn’t initializes the allocated memory. It contains garbage values	calloc () initializes the allocated memory to zero
4	type cast must be done since this function returns void pointerint *ptr;ptr = (int*)malloc(sizeof(int)*20);	Same as malloc () functionint *ptr;ptr = (int*)calloc(20, 20 * sizeof(int));

free()

Dynamically allocated memory with either calloc() or malloc() does not get return on its own. The programmer must use free() explicitly to release space.

syntax of free()

free(ptr);

This statement causes the space in memory pointer by ptr to be deallocated.

Example 1:

```
1.      #include <stdio.h>
2.      #include <stdlib.h>
3.      int main()
4.      {
5.      int n,i,*ptr,sum=0;
6.      printf("Enter number of elements: ");
7.      scanf("%d",&n);
8.      ptr=(int*)malloc(n*sizeof(int));
9.      if(ptr==NULL)
10.     {
11.     printf("Error! memory not allocated.");
12.     exit(0);
13.     }
14.     printf("Enter elements of array: ");
15.     for(i=0;i<n;++i)
16.     {
17.     scanf("%d",ptr+i);
18.     sum+=*(ptr+i);
19.     }
20.     printf("Sum=%d",sum);
21.     free(ptr);
22.     return 0;
23.     }
```

- The above Program finds sum of n elements entered by user. To perform this program, allocate memory dynamically using malloc() function.
- In line 8, a size of (n*4) bytes will be allocated using malloc() and the address of first byte is stored in ptr. If the allocation is not successful, malloc returns NULL.
- Line numbers 15-20 finds and displays the sum of entered values.
- In Line 21, free() deallocates the memory allocated by malloc().

Example 2:

In the above program if line 8 is replaced by the following statement

ptr=(int*)calloc(n,sizeof(int));

then, calloc() allocates contiguous space in memory for an array of n elements each of size of int, i.e, 4 bytes.

realloc()

If the previously allocated memory is insufficient or more than sufficient. Then, you can change memory size previously allocated using realloc().

Syntax of realloc()

```
ptr=realloc(ptr,newsize);
```

Here, *ptr* is reallocated with size of newsize.

```
1.	#include <stdio.h>
2.	#include <stdlib.h>
3.	int main()
4.	{
5.	int *ptr,i,n1,n2;
6.	printf("Enter size of array: ");
7.	scanf("%d",&n1);
8.	ptr=(int*)malloc(n1*sizeof(int));
9.	printf("Address of previously allocated memory: ");
10.	for(i=0;i<n1;++i)
11.	printf("%u\t",ptr+i);
12.	printf("\nEnter new size of array: ");
13.	scanf("%d",&n2);
14.	ptr=realloc(ptr,n2);
15.	for(i=0;i<n2;++i)
16.	printf("%u\t",ptr+i);
17.	return 0;
18.	}
```

In the above Program,

- In line 8, a size of (n1*4) bytes will be allocated using calloc() and the address of first byte is stored in ptr. Line numbers 15-20 finds and displays the sum of entered values.
- In Line 14, The allocated memory is modified into(n2*4) bytes using realloc() function.

Review Questions:

1. Explain how memory management can be done dynamically in C language?
2. Explain the differences between memory allocations malloc() and calloc()

Snippet Programs:

1.
```
#include<stdio.h>
#include<stdlib.h>

int main()
{
    int *p;
    p = (int *)malloc(20);
    printf("%d\n", sizeof(p));
    free(p);
    return 0;
}
```

2.
```
#include<stdio.h>
#include<stdlib.h>
#define MAXROW 3
#define MAXCOL 4

int main()
{
    int (*p)[MAXCOL];
    p = (int (*) [MAXCOL])malloc(MAXROW *sizeof(*p));
    return 0;
 }
```

3.
```
#include<stdio.h>
#include<stdlib.h>
int main()
{
 int *p;
 p = (int *)malloc(256 * 256);
 if(p == NULL)
     printf("Allocation failed");
 return 0;
 }
```

3.14 Command Line Arguments

- ✓ It is possible to pass some values from the command line to your C programs when they are executed. These values are called **command line arguments** and many times they are important for your program specially when you want to control your program from outside instead of hard coding those values inside the code.
- ✓ The command line arguments are handled using main() function arguments where **argc**(argument count) refers to the number of arguments passed, and **argv[**argument vector**]** is a pointer array which points to each argument passed to the program. Following is a simple example which checks if there is any argument supplied from the command line and take action accordingly:

Enter the following code and compile it:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
   int x;

   printf("%d\n",argc);
   for (x=0; x<argc; x++)
      printf("%s\n",argv[x]);
   return 0;
}
```

In this code, the main program accepts two parameters, argv and argc. The argv parameter is an array of pointers to string that contains the parameters entered when the program was invoked at the UNIX command line. The argc integer contains a count of the number of parameters. This particular piece of code types out the command line parameters. To try this, compile the code to an executable file named **aaa** and type **aaa xxx yyy zzz**. The code will print the command line parameters xxx, yyy and zzz, one per line.

The **char *argv[]** line is an array of pointers to string. In other words, each element of the array is a pointer, and each pointer points to a string (technically, to the first character of the string). Thus, **argv[0]** points to a string that contains the first parameter on the command line (the program's name), **argv[1]** points to the next parameter, and so on. The argc variable tells you how many of the pointers in the array are valid. You will find that the preceding code does nothing more than print each of the valid strings pointed to by argv.

Because argv exists, you can let your program react to command line parameters entered by the user fairly easily. For example, you might have your program detect the word **help** as the first parameter following the program name, and dump a help file to stdout. File names can also be passed in and used in your fopen statements.

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {
      printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with a single argument, it produces the following result.

```
$./a.out testing
The argument supplied is testing
```

When the above code is compiled and executed with a two arguments, it produces the following result.

```
$./a.out testing1 testing2
Too many arguments supplied.
```

When the above code is compiled and executed without passing any argument, it produces the following result.

```
$./a.out
One argument expected
```

It should be noted that **argv[0]** holds the name of the program itself and **argv[1]** is a pointer to the first command line argument supplied, and *argv[n] is the last argument. If no arguments are supplied, argc will be one, otherwise and if you pass one argument then **argc** is set at 2.

You pass all the command line arguments separated by a space, but if argument itself has a space then you can pass such arguments by putting them inside double quotes "" or single quotes ''. Let us re-write above example once again where we will print program name and we also pass a command line argument by putting inside double quotes:

```
#include <stdio.h>

int main( int argc, char *argv[] )
{
   printf("Program name %s\n", argv[0]);

   if( argc == 2 )
   {
      printf("The argument supplied is %s\n", argv[1]);
   }
   else if( argc > 2 )
   {
      printf("Too many arguments supplied.\n");
   }
   else
   {

     printf("One argument expected.\n");
   }
}
```

When the above code is compiled and executed with a single argument separated by space but inside double quotes, it produces the following result.

```
$./a.out "testing1 testing2"

Progranm name ./a.out
The argument supplied is testing1 testing2
```

The following example program demonstrates:

```
    #include <stdio.h>

  void main( int argc, char *argv[] )
  {
    int ctr;
    for( ctr=0; ctr < argc; ctr++ )
    {
      puts( argv[ctr] );
    }
```

```
   }
```

If this program is run from the command line as follows:

```
$./a.out stooges moe larry curley
```

-- the output is:

```
   stooges
   moe
   larry
   curley
```

Parameter Parsing Example

Command-line parameters can be used for many things. Let’s take a look at another example that will print different things depending on the flags given. If the –f flag is used then the given argument is printed once.If the –d flag is used the argument is printed twice.

Let’s look at the example:

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

void usage(void)
{
      printf("Usage:\n");
      printf(" -f<name>\n");
      printf(" -d<name>\n");
      exit (8);
}

int main(int argc, char *argv[])
{
      printf("Program name: %s\n", argv[0]);

      while ((argc > 1) && (argv[1][0] == '-'))
      {
                switch (argv[1][1])
                {
                          case 'f':
                                    printf("%s\n",&argv[1][2]);
                                    break;
```

```
                    case 'd':
                            printf("%s\n",&argv[1][2]);
                            printf("%s\n",&argv[1][2]);
                            break;

                    default:
                            printf("Wrong Argument: %s\n", argv[1]);
                            usage();
            }

            ++argv;
            --argc;
    }
    return (0);
}
```

After compiling you can run the program with the following parameters:

- # program
- # program –fhello
- # program –dbye
- # program –fhello –dbye
- # program –w

Note: that there is no space between the flag and the flag argument.

First the program name is printed (it will always be printed.)

The "while loop" looks for the dash sign. In case of the –f the flag argument is printed once. If the flag is –d then the flag argument is printed twice.

The argv[][] array will be used as follows:

For example: –f<name>

- argv[1][0] contains the dash sign
- argv[1][1] contains the "f"
- argv[1][2] contains the flag argument name

If a wrong flag sign is given then the usage is printed onto the screen.

<u>Review Questions</u>

1. Define Command line arguments? Does C support Command line arguments? Justify your answer.
2. Explain Commandline arguments in C language through an example.

Snippet Programs:

1.
```
#include<stdio.h>
int main(int argc, char **argv)
{
    printf("%c\n", **++argv);
    return 0;
}

#include<stdio.h>
#include<stdlib.h>
int main(int argc, char **argv)
{
   printf("%s\n", *++argv);
    return 0;
}
```

2.
```
#include<stdio.h>
void fun(int);
int main(int argc)
{
    printf("%d ", argc);
    fun(argc);
    return 0;
}
void fun(int i)
{
    if(i!=4)
        main(++i);
}
```

3.
```
#include<stdio.h>
int main(int argc, char *argv[])
{
    while(--argc>0)
        printf("%s", *++argv);
    return 0;
}
```

By :
Mr.Shaik Masthan Valli
Asst.,Prof..
Dept of CSE
GITAM University
HTP Campus.

Enumerated Types (enum) in C

In C programming language, an enumeration is used to create user-defined datatypes. Using enumeration, integral constants are assigned with names and we use these names in the program. Using names in programming makes it more readable and easy to maintain.

Enumeration is the process of creating user defined datatype by assigning names to integral constants

We use the keyword **enum** to create enumerated datatype. The general syntax of enum is as follows...

enum {name1, name2, name3, ... }

In the above syntax, integral constant '0' is assigned to name1, integral constant '1' is assigned to name2 and so on. We can also assign our own integral constants as follows...

enum {name1 = 10, name2 = 30, name3 = 15, ... }

In the above syntax, integral constant '10' is assigned to name1, integral constant '30' is assigned to name2 and so on.

Example Program for enum with default values

```
#include<stdio.h>

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

enum day { Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} ;

void main(){

   enum day today;

   today = tuesday ;

   printf("\ntoday = %d ", today) ;

}
```
n the above example program a user defined datatype "**day** is created seven values, monday as integral constant '0', tuesday as integral constant '1', wednesday as integral constant '2', thursday as integral constant '3', friday as integral constant '4', saturday as integral constant '5'

and sunday as integral constant '6'. Here, when we display tuesday it displays the respective integral constant '1'.

We can also change the order of integral constants, consider the following example program.

Example Program for enum with changed integral constant values

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

enum day { Monday = 1, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday} ;

void main(){

   enum day today;

   today = tuesday ;

   printf("\ntoday = %d ", today) ;

}
```

In the above example program, the integral constant value starts with '1' instead of '0'. Here, tuesday value is displayed as '2'.

We can also create enum with our own integral constants, consider the following example program.

Example Program for enum with defferent integral constant values

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

enum threshold {low = 40, normal = 60, high = 100} ;

void main(){

   enum threshold status;

   status = low ;

   printf("\nYou are at %d state. Work hard to improve!!", status) ;

}
```

Some times we may assign our own integral constant from other than the first name. In this case, compiler follows default integral constants for the name that is before the user-defined integral constant and from user-defined constant onwards it resumes. Consider the following example program.