Chapter-9

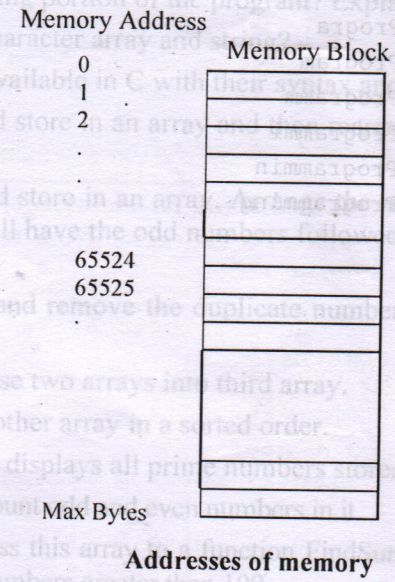
Pointer

The knowledge of memory address is necessary before understanding pointer. A computer has primary memory, also known as RAM (Random Access Memory). RAM holds programs that the computer is currently running along with the data (i.e. variables) which are currently manipulated. All the variables used in a program reside in the memory when the program is being executed. The computer memory (i.e. RAM) is divided into a number of small units or locations. Each location is represented by some unique number known as memory address. Each memory location is capable of storing a small number (0 to 255), which is known as a byte. For example: A char data is one byte in size and hence needs one memory location of the memory. Similarly, an integer data is two bytes in size and hence needs two memory locations of the memory. The smallest unit for memory address is bit (binary digit) i.e. 0 or 1.

Memory Unit Conversion is as follows:

```
8 bits=1 bytes. For example: 1001 1111 is one byte
1024 bytes= 1 kilo Bytes (KB)
1024 KB=1 Mega Bytes (MB)
1024 MB=1 Giga Bytes (GB)
1024 GB=1 Tera Bytes(TB)
```

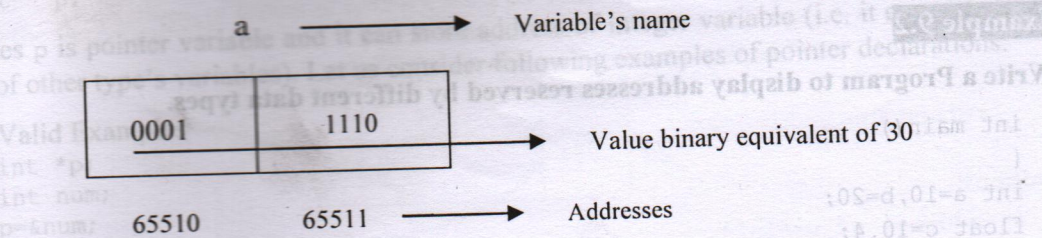
Each byte in memory is associated with a unique address. An address is an integer that represents a particular memory location. The integer is always positive value that ranges from zero to some positive integer constant corresponding to the last location in the memory. Thus, a computer having 1 GB RAM has 1024*1024*1024 i.e. 1073741824 Bytes and these 1073741824 Bytes are represented by 1073741824 different address locations. For example, the memory address 65524 represents a Memory Bytes in memory and it can store data of one Bytes.



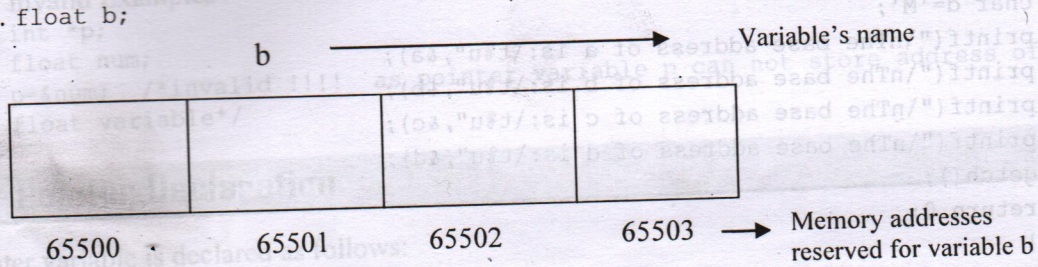
Every variable in a C program is assigned a space in memory. When a variable is declared, it tells computer the type of variable and name of the variable. According to the type of variable declared in a program, the necessary memory locations are reserved. For example, `int` requires two bytes, `float` requires four bytes and `char` requires one bytes.

For examples:

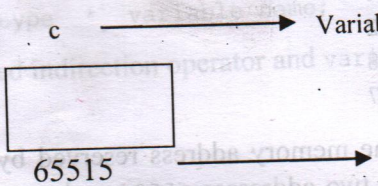
i. `int a=30;`



ii. `float b;`



iii. `char c;`



Example 9.1

Write a program to display memory location reserved by a variable.

```
int main()
  {
  int a;
  a=20;
  printf("\nThe address of a is:\t%u",&a);
  printf("\nThe value of a is:\t%d",a);
  getch();
  return 0;
  }
```

Output:

```
The address of a is:    65524
The value of a is:      20
```

Here, &a denotes address of variable a. The variable a takes two bytes in memory as it is of type int. It takes two memory locations 65524 and 65525. Although it takes a block two bytes memory consisting two different addresses, this memory is accessed by the first address. Thus, 65524 is the first byte reserved by the variable and it is also known as base address. Again, the control string u (unsigned) is used to display address of the variables, as the value of address is positive integer.

Example 9.2

Write a Program to display addresses reserved by different data types.

```
int main()
{
int a=10,b=20;
float c=10.4;
char d='M';
printf("\nThe base address of a is:\t%u",&a);
printf("\nThe base address of b is:\t%u",&b);
printf("\nThe base address of c is:\t%u",&c);
printf("\nThe base address of d is:\t%u",&d);
getch();
return 0;
}
```

Output

```
The base address of a is:    65524
The base address of b is:    65522
The base address of c is:    65518
The base address of d is:    65517
```

Here, memory address is in decreasing order. The memory address reserved by variable a is 65524. The variable a is type int. Thus, it takes two addresses 65524 and 65525. Thus, base address for b is 65522. Similarly, the variable b takes two memory locations 65522 and 65523. The base address of float variable c is 65518 (i.e. it reserves four memory locations 65518, 65519, 65520 and 65521). The type of variable d is char. As char variable takes only one byte in memory, the variable c takes only one memory address 65517.

9.1 Introduction to Pointer

A pointer is special type of a variable. It is special because it contains a memory address instead of values (i.e. data). A pointer variable is declared with specific data type, like any other normal variable, so that it will work only with memory address of a given type. Just as integer variable can hold only integers, a character variable can hold the only characters and so forth, each pointer variable can point only to one specific type (int, char, float, double or any user defined data type). A pointer can have any name that is legal for other variable and it is declared in the same fashion like other variables but is always preceded by '*' (asterisk) operator.

et us consider an integer variable declared as

```
int num ;
```

ere, num is considered as integer variable and it can store only integer data.

imilarly, if a pointer variable p is declared as

```
int * p;
```

signifies p is pointer variable and it can store address of integer variable (i.e. it can not store ddress of other type's variables). Let us consider following examples of pointer declarations:

i. Valid Examples

```
int *p;
int num;
p=&num;
```

ii. Invalid Examples

```
int *p;
float num;
p=&num;  /*invalid !!!!  as pointer variable p can not store address of
float variable*/
```

9.2 Pointer Declaration

A pointer variable is declared as follows:

Syntax:

```
data_type  *  variable_name;
```

Here, * is called indirection operator and variable_name is now pointer..

Examples:

```
int *x;
float *f;
char *y;
```

In the first statement, 'x' is an integer pointer and it tells to the compiler that it holds the address of any int variable. In the same way 'f' is a float pointer which stores the address of any float variable and 'y' is a character pointer that stores the address of any char variable.

Example 9.3

Write a program to define a pointer variable of type int to store memory address of another integer variable. Display content of the integer variable using pointer variable.

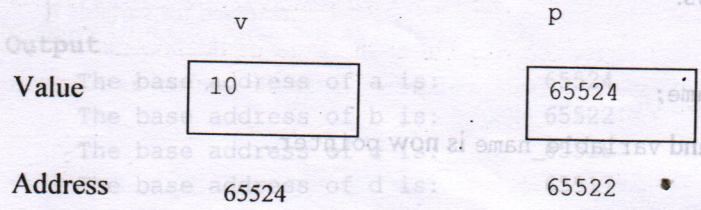
```
int main( )
 {
  int v=10, *p;        /* Here, * indicates p as pointer variable */
      p=&v;   /* address of integer variable v is assigned to integer pointer p */
  printf("\n address of v=%u",&v);
```

```
        /* display address of variable v directly*/
    printf("\n address of v=%u",p);
        /*display address of variable indirectly */
    printf("\n value of v=%d",v);    /* display value of v directly */
    printf("\n value of v=%d",*p);   /* display value of v indirectly */
        /* Here, *p indicates value at address pointed by p
           i.e. value at address of variable v which is stored in pointer p */
    printf("\n address of p=%u",&p);
        /*display address of pointer variable p*/
    getch();
    return 0;
}
```

Output

```
address of v=65524
address of v=65524
value of v=10
value of v=10
address of p=65522
```

It can be illustrated as



Here, it is important to know that

- v and *p is same
- &v and p is same

Here `v` is an integer variable and its value is `10`. The variable `p` is declared as pointer variable. The statement `p=&v` assigns address of 'v' to 'p' (i.e. 'p' is the pointer to variable 'v'). To access the address and value of 'v', the pointer 'p' can be used. The value of 'p' is nothing but address of the variable 'v' (i.e. `&v` is equivalent to `p` due to statement `p=&v`). Similarly, `*p` is used to represent the value stored at a location pointed by pointer variable `p` (i.e. it is equivalent to `v`). A pointer variable can also be initialized at the time of declaration as follows.

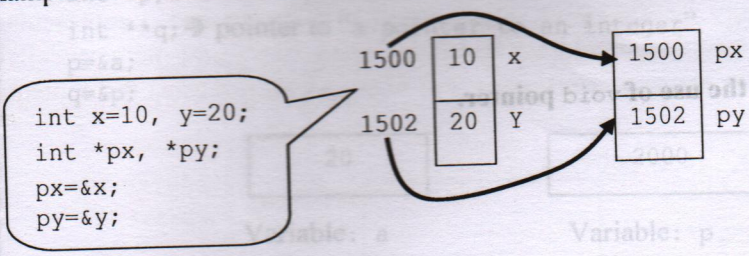
```
int a;
int *p=&a;
```

9.2.1 Indirection or Dereference Operator

The operator *, used in front of a variable, is called `pointer` or `indirection or dereference operator`. A normal variable provides direct access to its value whereas a pointer provides indirect access to the value of the variable whose address it stores/points. The indirection operator `(*)` is used in two distinct ways with pointers, `declaration` and `dereference`. When the pointer is declared, the star indicates that it is a pointer, not a normal variable. When the

pointer is dereferenced (i.e. use of * in front of variable except in variable declaration), the indirection operator indicates "value at the memory location stored in the pointer" or "the content of the location pointed by pointer variable" instead of address itself. Also note that * is the same operator that can be used as the multiplication operator. The compiler knows which operator to call, based on the context.

A pointer constant is an address, while a pointer variable is a place to store addresses. For example the declaration is:



9.2.2 Address Operator

The operator & is also known as address operator as we can represent address of a variable using this operator. Thus, &a denotes the address of variable a.

9.3 Initializing Pointers

Address of a variable can be assigned to a pointer variable at the time of declaration of the pointer variable. The initializing a pointer with address of another variable is known as pointer initialization. For examples:

```
int num;
int *ptr=&num;
```

These two statements can be replaced with following statements:

```
int num;
int *p;
p=&num;
```

9.4 Bad Pointer

When a pointer variable is first declared, it does not have a valid address. The pointer is uninitialized or bad and it is called bad pointer. The dereference operation on a bad pointer is a serious run time error. Thus, each pointer must be assigned a valid address before dereferenceing it.

In fact, every pointer contains garbage value before assignment of valid address. The pointer assignment statement overwrites the garbage value with a correct reference to an address, and thereafter the pointer works fine. There is nothing automatic that gives a pointer a valid address.

9.5 void Pointer

A `void` pointer is a special type of pointer. It can point to variables of any data type (i.e single pointer can point to `int`, `float`, `char` or other type variables). Using `void` pointer, the pointed data can not be referenced directly (i.e * operator can not be used on them). The type casting or assignment must be used to change the `void` pointer to a concrete data type to which we can refer.

Example 9.4

Write a program to illustrate the use of `void` pointer.

```
int main()
{
int a=10;
double b=4.5;
void *vptr;
vptr=&a;
printf("\na=%d",*((int *)vptr));/* not simply *vptr */
vptr=&b;
printf("\nb=%lf",*((double *)vptr));
getch();
return 0;
}
```

Output

```
a=10
b=4.500000
```

Explanation: Here, `vptr` is `void` pointer. In statement `vptr=&a;`, the memory address of integer variable `a` has been assigned to the `void` pointer `vptr` while the statement `vptr=&b;` assigns the memory address of `float` variable `b` to the `void` pointer `vptr`. Thus, the same pointer can point to different data types' variables using concept of `void` pointer.

9.6 NULL Pointer

A `NULL` pointer is a special pointer value that points nowhere or nothing in memory address. That is, no other valid pointer to any other variable or array cell or anything else will ever be equal to a `NULL` pointer. We can define a `NULL` pointer using predefined constant `NULL`, which is defined in header files: `stdio.h`, `stdlib.h`, `string.h`. For example: the following statement defines a pointer `ptr` which points nothing in memory address.

```
int *ptr=NULL;
```

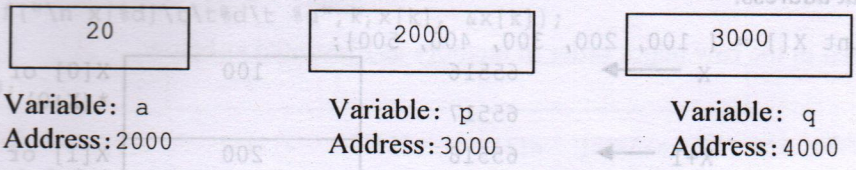
To test a pointer for `NULL` before inspecting the value pointed to it, the program statements such as the following can be used.

```
if(ptr!=NULL)
    printf("value=%d",*ptr);
```

9.7 Pointer to Pointer (Double Pointer)

C allows the use of a pointer that points to other pointers, and these, in turn, point to data. In case of pointer to pointer declaration, we need to add asterisk (*) for each level of reference. For examples:

```
int a=20;
int *p;
int **q;➔ pointer to "a pointer to an integer"
p=&a;
q=&p;
```



Here, a is normal variable, p is pointer variable and q is double pointer variable. The address of pointer p has been assigned to double pointer q (i.e. q points another pointer p) and address of variable a has been assigned to pointer p. Thus, *p represents the value at memory address pointed by p (i.e. value of a). Similarly, *q represents value of memory address pointed by q (i.e. memory address of a) and **q represent value at memory address pointed by *q (i.e. p) that signifies value of a (i.e. 20).

Thus, to refer to variable a using pointer q, dereference it twice because there are two levels of indirection involved. Here, both *p and **q displays 20 if they are printed with a printf statement;

9.8 Array of Pointers

The multiple pointers of same type can be represented by an array. Each element of such array represents pointer and they can point to different locations. An array of pointers can be declared as

```
data_type *pointer_name[size];
```

For example:

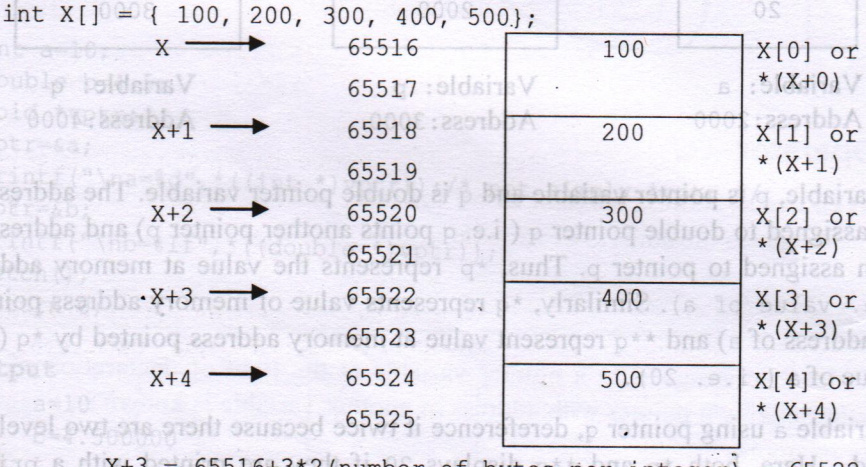
```
int *p[10];
```

Explanation: Here, p is an array of 10 pointers, each of which points to an integer. The first pointer is called p[0], the second is p[1] and so on up to p[9]. Initially, these pointers are uninitialized. Examples:

```
int a=10, b=100, c=1000;
p[0]=&a;
p[1]=&b;
p[2]=&c; and so on.
```

9.9 Relationship between 1-D Array and Pointer

An array name by itself is a pointer which represents base address of the array. Thus, if x is a one dimensional array, then address of the first array element can be expressed as either &x[0] or simple as x (i.e. x is same as &x[0]). The address of the second array element can be written as either &x[1] or as x+1 and so on. In general, the address of the array element i+1 can be expressed as either &x[i] or as x+i. In the expression x+i, x represents array name (address of first element) whose elements may be integers, characters, float etc and i represents integer quantity. Thus, here x+i specifies an address that is a certain number of memory cells beyond the address of the first array element. Thus, x[i] and *(x+i) both represent the content of that address.



X+3 = 65516+3*2(number of bytes per integer) = 65522

Example 9.5

Write a program to initialize an one-dimensional array and display its elements with their memory addresses using array name as a pointer.

```
int main()
  {
  int x[5]={20,40,6,8,100}, k;
  printf("array element \t\t element's value \t\t address");
  for(k=0;k<5;k++)
     printf("\n x[%d]\t\t%d\t %u",k,*(x+k), x+k);
  getch();
  return 0;
  }
```

output

```
array element    element's value        address
 x[0]                  20                65516
 x[1]                  40                65518
 x[2]                  6                 65520
 x[3]                  8                 65522
 x[4]                  100               65524
```

Explanation: Here, the integer variable k acts as the element number and its value varies from 0 to 4. When it is added with an array name 'x' (i.e. Address of the first element), it points to the the next consecutive memory location. Thus, the array element number, element's value and their corresponding addresses are displayed using pointer notation. The above program can be solved without using pointer (i.e. using array notation) as follows:

```
int main()
    {
     int x[5]={20,40,6,8,100}, k;
     printf("\narray element\telement's value\taddress");
     for(k=0;k<5;k++)
        printf("\n x[%d]\t\t%d\t %u",k,x[k], &x[k]);
     getch();
     return 0;
    }
```

Thus, it can be concluded that in the case of array, `&x[k]` is same as `x+k` and `x[k]` is same as `*(x+k)`.

Example 9.6

Write a program to read marks of 10 students secured in a subject and store them in an array using pointer notation. Calculate average marks of them and display on screen.

```
int main()
    {
     int marks[10],i,sum=0;
     float avg;
     printf("Enter marks of each student:\n");
     for(i=0;i<10;i++)
       {
       scanf("%d",marks+i);
       sum+=*(marks+i);
       }
     avg=sum/10;
     printf("\nThe average is=%f",avg);
     getch();
     return 0;
    }
```

Output

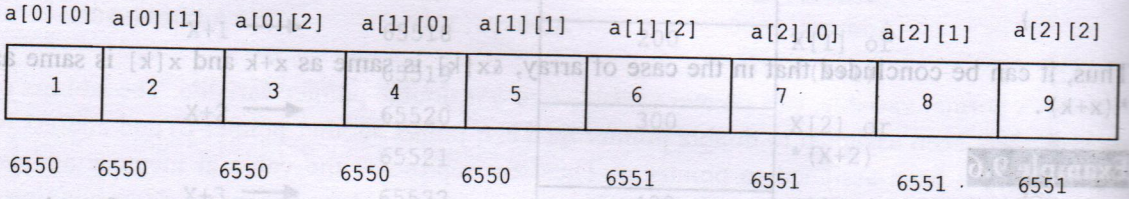
```
Enter marks of each student:
   56    90    65    45    34    23    89    67    56    78
The average is=60.000000
```

9.10 Pointers and 2-D Arrays

The multi-dimensional array can also be represented with an equivalent pointer notation like in single dimensional array. A 2-D array is actually a collection of 1-D arrays, each indicating a row (i.e. 2-D array can be thought as 1-D array of rows). It is stored in memory in the row form. For example, the matrix,

$$a=\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix}$$

is stored in the row major order in memory as illustrated as below.



It is possible to access 2-D array elements using pointer notation in the same way as in 1-D array. Each row of the 2-D array is treated as a 1-D array. We can define a 2-D array in the form of a pointer to a group of contiguous 1-D arrays.

Syntax for declaration of 2-D array using pointer notation:

```
data_type (*ptr_variable)[size2];
```

(Instead of `data_type array[size1][size2];` in normal array declaration)

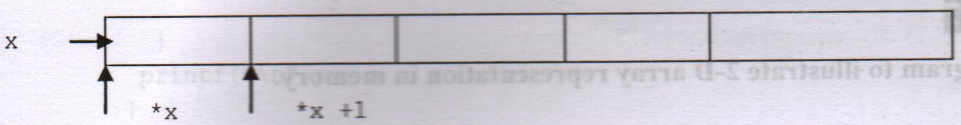
Example:

Let us suppose `x` is a 2-D integer array having 4 rows and 5 columns. We can declare `x` as

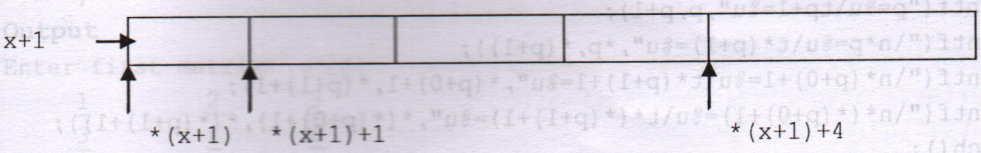
```
int (*x)[5];
```

rather than `int x[4][5];`

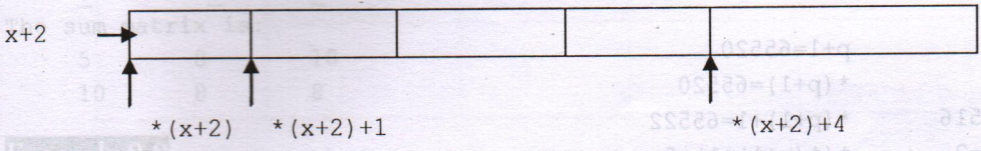
Here, `x` is defined to be a pointer to a group of contiguous one dimensional `5`-element integer arrays. The `x` points to the first `5`-element array, which is actually first row of the 2-D array. Similarly, `x+1` points to the second `5`-element array, which is the second row of the 2-D array. It is illustrated as



1st 1-D array or first row



2nd 1-D array or second row



3rd 1-D array or third row

It can be summarized as:

`x` ➔ pointer to 1st row

`x+i` ➔ pointer to i^{th} row

`*(x+i)` ➔ pointer to first element in the i^{th} row

`*(x+i)+j` ➔ pointer to j^{th} element in the i^{th} row

`*(*(x+i)+j)` ➔ value stored in the cell `i,j`

Thus, in 2-D array,

`&x[0][0]` is same as `*x` or `*(x+0)+0`

`&x[0][1]` is same as `*x+1` or `*(x+0)+1`

`&x[2][0]` is same as `*(x+2)` or `*(x+2)+0`

`&x[2][4]` is same as `*(x+2)+4`

and

`x[0][0]` is same as `**x` or `*(*(x+0)+0)`

`x[0][1]` is same as `*(*x+1)` or `*(*(x+0)+1)`

`x[2][0]` is same as `*(*(x+2))` or `*(*(x+2)+0)`

`&x[2][4]` is same as `*(*(x+2)+4)`

In general, `&x[i][j]` is same as `*(x+i)+j` and `x[i][j]` is same as `*(*(x+i)+j)`.[see section 9.12 for further understanding]

Example 9. 7

Write a program to illustrate 2-D array representation in memory.

```
int main()
  {
    int p[2][3]={   {1,2,3},   {4,5,6}  };
    printf("p=%u\tp+1=%u",p,p+1);
    printf("\n*p=%u\t*(p+1)=%u",*p,*(p+1));
    printf("\n*(p+0)+1=%u\t*(p+1)+1=%u",*(p+0)+1,*(p+1)+1);
    printf("\n*(*(p+0)+1)=%u\t*(*(p+1)+1)=%u",*(*(p+0)+1),*(*(p+1)+1));
    getch();
    return 0;
  }
```

Output

```
p=65514          p+1=65520
*p=65514         *(p+1)=65520
*(p+0)+1=65516   *(p+1)+1=65522
*(*(p+0)+1)=2    *(*(p+1)+1)=5
```

Example 9.8

Write a program to define two matrices of order mxn using pointer notation. Read elements of the matrices from user and add them using pointer.

```
#define m 2
#define n 3
 int main()
     {
      int (*a)[n],(*b)[n],*(sum)[n],i,j;
      printf("Enter first matrix:\n");
      for(i=0;i<m;i++)
        for(j=0;j<n;j++)
          scanf("%d",*(a+i)+j);

      printf("Enter second Matrix:\n");
        for(i=0;i<m;i++)
          for(j=0;j<n;j++)
            scanf("%d",*(b+i)+j);

       printf("\nThe sum matrix is:\n");
       for(i=0;i<m;i++)
        {
          for(j=0;j<n;j++)
           {
            *(*(sum+i)+j)= *(*(a+i)+j)+ *(*(b+i)+j);
            printf("\t%d",*(*(sum+i)+j));
```

```
        }
        printf("\n");
      }
    getch();
    return 0;
}
```

Output

```
Enter first matrix:
1       2       3
3       4       6
Enter second Matrix:
4       6       7
7       4       2
The sum matrix is:
5       8       10
10      8       8
```

Example 9.9

Write a program to read two matrices of order mxn and pxq, multiply them and display the product matrix using pointer.

```
#define m 2
#define n 3
#define p 3
#define q 2
int main()
  {
  int (*matrix1)[n],(*matrix2)[q],product[m][q],i,j,k;
  int partialSum=0;
  printf("\nEnter elements of first matrix\n");
  for(i=0;i<m;i++)
    for(j=0;j<n;j++)
      scanf("%d",*(matrix1+i)+j);

  printf("\nEnter elements of second matrix\n");
    for(i=0;i<p;i++)
      for(j=0;j<q;j++)
        scanf("%d",*(matrix2+i)+j);

  for(i=0;i<m;i++)
    for(j=0;j<p;j++)
      {
        for(k=0;k<n;k++)
          {
            partialSum+=*(*(matrix1+i)+k)**(*(matrix2+k)+j);
          }
```

```
            *(*(product+i)+j)=partialSum;
            partialSum=0;
            }

    printf("\nThe product matrix is\n");
    for(i=0;i<m;i++)
      {
       for(j=0;j<q;j++)
          printf("\t%d",*(*(product+i)+j));
       printf("\n");
      }
    getch();
    return 0;
}
```

Output

```
Enter elements of first matrix
 2   3   5
 1   3   7
Enter elements of second matrix
 4   2
 1   6
 7   1
The product matrix is
        46      27
        56      27
```

9.11 Pointer Operations

We can perform various operations on pointers like operations on ordinary variables. To illustrate the pointer operations, let us consider following declaration of ordinary variables and pointer variables.

```
int a,b;
float c;
int *p1, *p2;
float *f;
```

1) Address of an ordinary variable of a particular type can be assigned to a pointer variable the same type.

   For examples:

   ```
   p1=&a;   p2=&b ; f=&c;
   ```

2) The content of one pointer can be assigned to another pointer variable provided they both are of the same data type. For example:

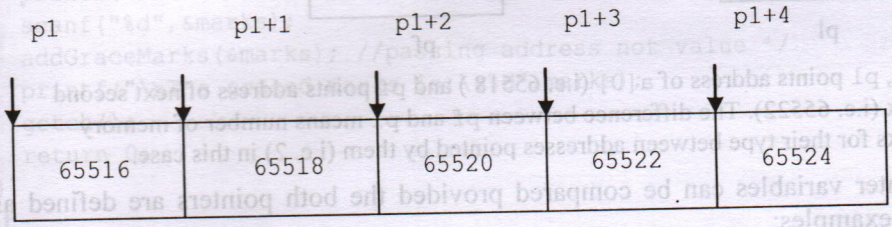
   ```
   p1=p2;   /* valid */
   f=p1;    /* invalid  as two pointers are not of same type*/
   ```

3) An integer can be added to or subtracted from a pointer. For examples:

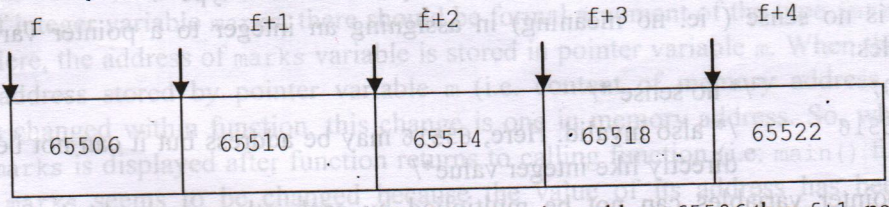
i. `p1+2` : It specifies an address which is two memory blocks for integer (i.e. 4 memory btyes) beyond the address pointed by `p1`.

ii. `f+1` : It specifies an address which is one memory block for float data (i.e. 4 memory bytes) beyond the address pointed by `f`.

Here, memory block for integer means 2 bytes and memory block for float means 4 bytes in memory.



Here, `p1` is integer type pointer. If `p1` points or stores address `65516`, then `p1+1` means address pointed by `p1+ size in bytes of data type of pointer` i.e. `65516+2=65518`. Thus, `p1+1` represent address `65518` in stead of `65517`.



Here, `f` is pointer of `float` type. If pointer `f` points address `65506` then `f+1` points next block's address. As `float` variable takes four bytes in size, the block address is `65510`. Thus, `f+1=65506+`one block of memory for float`=65506+4=65510`.

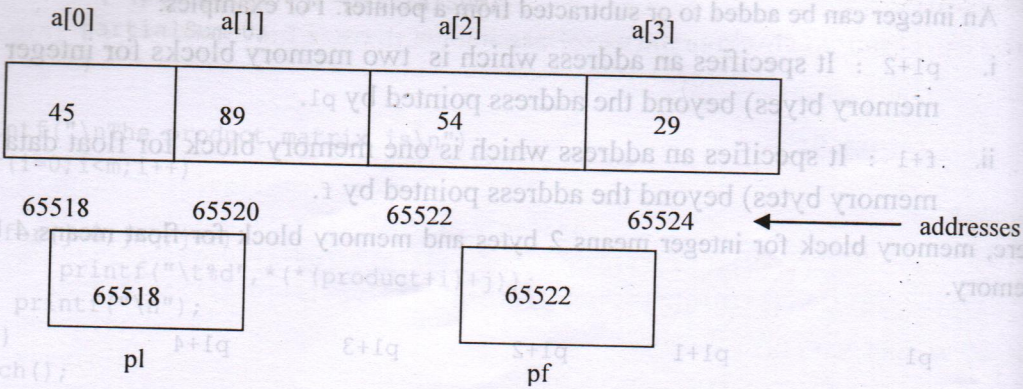
4) One pointer can be subtracted from other pointer if both pointers point to elements of same array. For example:

```
void main()
 {
  int a[]={45,89,54,29},*pf,*pl;
  pl=a;
  pf=a+2;
  printf("%d",pf-pl);
 }
```

Output

```
2
```

Here, the difference of two pointers `pl` and `pf` means memory blocks for their type between addresses pointed by them.



Here, `pl` points address of `a[0]` (i.e. 65518) and `pf` points address of next second block (i.e. 65522). The difference between `pf` and `pl` means number of memory blocks for their type between addresses pointed by them (i.e. 2) in this case.

5) Two pointer variables can be compared provided the both pointers are defined as same type. For examples:

```
if( p1<p2)
{
}          /* is valid comparison when p1 and p2 are of same type */
```

6) There is no sense (ie. no meaning) in assigning an integer to a pointer variable. For examples:

```
p1=100;          /* no sense */
p2 =65516        /* also invalid. Here, 65516 may be address but it can not be assigned
                    directly like integer value*/
```

7) Two pointer variables can not be multiplied by each others and added together. For examples:

```
p1+p2 ;          /*invalid */
p1*p2;           /*invalid */
```

8) A pointer variable can not be multiplied by a constant. For example:

```
p1*2;            /* invalid */
```

9) `NULL` value can be assigned to a pointer variable.

```
p1=NULL;     /*valid. Here NULL represents value 0 */
```

9.12 Passing Pointer to Function

A pointer can be passed to a function as an argument. Passing a pointer means passing address of a variable instead of value of the variable. As address is passed in this case, this mechanism is also known as `call by address` or `call by reference`. When a pointer is passed to a function while function calling, the formal argument of the function must be compatible with the supplied pointer in the actual argument (For example: if an integer pointer is being passed, the formal argument in function must be pointer of the type integer and so forth). As address of a variable is passed in this mechanism, if value stored in the passed address is changed within function definition, the value of actual variable is also changed.

Example 9.10

Write a program to illustrate the use of passing pointer to a function.

```
void addGraceMarks(int *m)
    {
    *m=*m+10;
    }
int main()
    {
     int marks;
     printf("Enter actual marks\t");
     scanf("%d",&marks);
     addGraceMarks(&marks); //passing address not value */
     printf("\nThe graced marks is:\t%d",marks);
     getch();
     return 0;
    }
```

Output

```
Enter actual marks    22
The graced marks is:    32
```

Explanation: Here, the address of variable `marks` has been passed to a function. To receive the address of integer variable `marks`, there should be formal argument of the type `integer` pointer (i.e. `m`). Here, the address of `marks` variable is stored in pointer variable `m`. When the content of memory address stored by pointer variable `m` (i.e. content of memory address of variable `marks`) is changed within function, this change is one in memory address. So, when value of variable `marks` is displayed after function returns to calling function (i.e. `main()` function), the value of `marks` seems to be changed because the value of its address has been changed. Although there is no return statement within function definition of this example, we are changing the value of variable whose addess is being passed as argument within main function.

Example 9.11

Write a program to convert upper case letter into lower and vice versa using call by reference or passing pointer to a function.

```
void conversion(char *); //function prototypes
int main()
    {
     char input;
     printf("Enter Character of Your Choice:\n");
     scanf("%c",&input);
     conversion(&input);
     printf("\nThe corresponding character is:\t%c",input);
     getch();
     return 0;
    }
void conversion(char *c)
```

```
{
 if(*c>=97 && *c<=122)
    *c=*c-32;
 else if(*c>=65 && *c<=90)
    *c=*c+32;
}
```

Output

```
a) Enter Character of Your Choice:  a
   The corresponding character is:  A
b) Enter Character of Your Choice: M
   The corresponding character is:  m
```

9.13 String and Pointer

It is already known that a string is an array of characters.An array is closely related to pointer (i.e. array name is itself pointer which points to first element of the array). Thus, we can say that string and pointer are closely related. For examples:

```
char name[5]="Shyam";
```

Here, the string variable `name` is an array of characters, it is a pointer also which points to the first character of the string. Thus, it can be used to access and manipulate the characters of the string. When a pointer to `char` is printed in the format of a string, it will start to print the pointer character and then successive characters until the end of string is reached. Thus, `name` prints "Shyam", `name+1` prints "hyam" and `name+2` prints "yam" and so on (i.e. `name+i` represents characters in string `name` starting from `ith` position).

Example 9.12

Write a program to illustrate the use of string pointer to display characters in the string.

```
#include<stdio.h>
#include<string.h>
int main (void)
 {
  char *namaste = "NAMASKAR SIR";
  //char namaste[20]="NAMASKAR SIR";
  char name[40];
  printf("Enter your name:");
  gets(name);
  puts(namaste);
  printf("\nNamaskar %s Sir", name);
  getch();
  return 0;
 }
```

Output

```
Enter your name:ram
NAMASKAR SIR
NAMASKAR ram SIR
```

In the above program, we initialized a string with pointer. Alternatively the commented line (`char Namaste[] = "Namaskar Sir"`) is also the same in which the string is initialized using array. Is there a difference? Yes. The array version of this statement reserve a memory of 20 bytes where as pointer only reserve a memory of 13 bytes.

'N'	'A'	'M'	'A'	'S'	'K'	'A'	'R'	' '	'S'	'I'	'R'	'\0'
13 bytes only												

Similarly, we can explain the same scenario for initialization of two dimensional array in convention method and pointer method (i.e. pointers to strings).

i. Initialization of array of strings using two dimensional conventional array

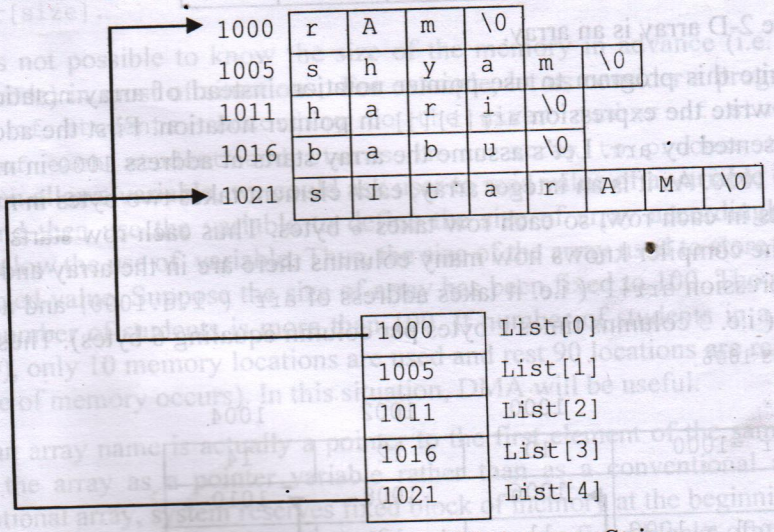
```
char list[5][10] ={"ram", "shyam", "hari","babu", "sitaram"};
```

	0	1	2	3	4	5	6	7	8	9
1000	r	a	M	\0						
1010	s	h	Y	a	m	\0				
1020	h	a	R	i	\0					
1030	b	a	B	u	\0					
1040	s	i	T	a	r	a	m	\0		

Array of Strings (Array Version)

ii. Initialization of array of strings using array of pointers

```
char *list[5] = {"ram", "shyam", "hari","babu", "sitaram"};
```



Array of Pointers to String

It shows that the pointer version takes up less memory; it ends at 2028, while the array version ends at 1049. Hence, to initialize strings as pointers is to use memory more efficiently and achieve greater flexibility in the manipulation of the strings in the array.

> Initializing a group of strings using pointer notation uses less memory than initializing them as a two dimensional array.

2-D array and double indirection pointer to pointer

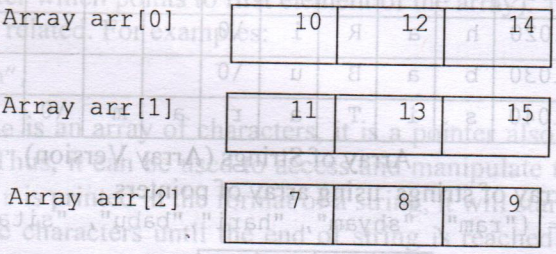
The above explanation of string manipulation using pointers in 2-D array is simply a double indirection pointer to pointer. Let us take an example of assigning values to the elements of 2-D array and perform equivalent pointer operation on them.

```
int arr[3][3]= { {10, 12, 14},
                 {11, 13, 15}
                 {7, 8, 9}     };
```

10	12	14
11	13	15
7	8	9

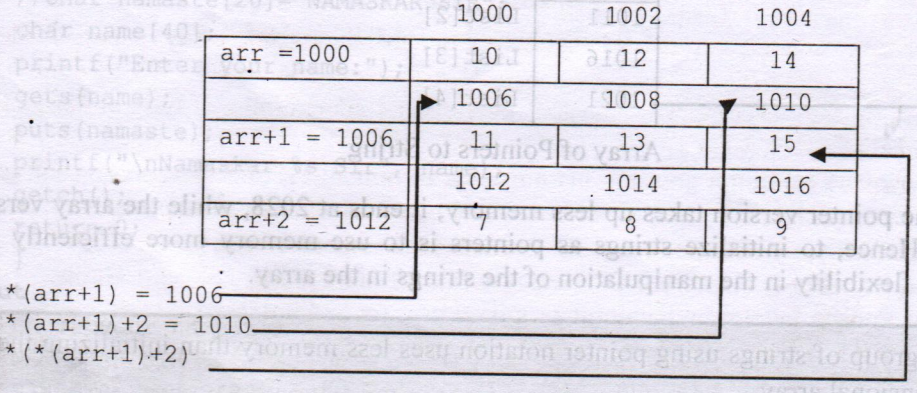
Table: Elements in an array `arr[3][3]`

The two dimensional array can be understood using array of 1-D arrays as follow:



Here, each row of above 2-D array is an array.

Now, suppose we rewrite this program to use pointer notation instead of array notation. The question is how do we write the expression `arr[i][j]` in pointer notation. First the address of the entire array is represented by `arr`. Let's assume the array starts at address `1000` in memory, so `arr` denotes address `1000`. As it is an integer array, each element takes two bytes in memory. There are three elements in each row, so each row takes `6` bytes. Thus each row starts `6` bytes far from the last one. The compiler knows how many columns there are in the array and knows how to interpret the expression `arr+1` (i.e. it takes address of `arr ( i.e.1000)` and adds the number of bytes in row (i.e. `3` columns times `2` bytes per column equaling 6 bytes). Thus `arr+1` is interpret as the address `1006`



Now, how do we refer to the individual elements of a row? We have already mentioned that the address of an array is the same as the address of the first element of the array. The address of the array formed by 2nd row of the `arr[][]` is `arr[1]` or `arr+1` in pointer notation. The address of first element of this array is `&arr[1][0]` or `*(arr+1)` in pointer notation. For the first element of this 1st row, the address is `(arr+1)` or `*(arr+1)`, however for the 3rd element of the 1st row the address is `*(arr+1)+2` and hence the content or value of that address is `*(*(arr+1)+2)` which is 15. In other words: `arr[i][j]` is equivalent to `*(*(arr+i)+j)`.

9.14 Dynamic Memory Allocation (DMA)

The process of allocating and freeing memory in a program at run time is known as Dynamic Memory Allocation (DMA). The DMA process reserves the memory required by the program and allows the program to utilize the memory. When the allocated memory is no more in use, it returns the allocated memory (i.e. the valuable resource) to the system. The deallocated memory now is free and any other program can use it.

Though arrays can be used for data storage, they are of fixed size. The programmer must know the size of the array or data in advance while writing the programs. A variable can not be used to define size of array while declaring an array. The following code is not valid in C programming.

```
int size=10; // or read value for variable size from user.
int arr[size].
```

But it is not possible to know the size of the memory in advance (i.e. at the time of writing source code) in most of situations,. For example, let us consider a program to ask user for number of students. According to the given number of students, it will ask marks of each student and stores in an array to process the marks. If an array definition allows variable, we could ask user to read value of a variable (i.e. number of student here) and then use the variable to define the size of array accordingly. But array definition does't allow the use of variable. Thus, the size of the array used to store the marks of students is to be fixed value. Suppose the size of array has been fixed to 100. The program wouldn't work if the number of students is more than 100. If number of students in a school is less than 100 (say 10), only 10 memory locations are used and rest 90 locations are reserved but not used (i.e. wastage of memory occurs). In this situation, DMA will be useful.

Since an array name is actually a pointer to the first element of the same array, it is possible to define the array as a pointer variable rather than as a conventional array. While defining a conventional array, system reserves fixed block of memory at the beginning of program execution which is inefficient. The system doesn't need to reserve fixed blocks of memory when the array is represented in terms of a pointer variable. The use of a pointer variable to represent an array requires some type of initial memory assignment before the array elements are processed. This is known as Dynamic Memory Allocation. Using DMA, a program can request more memory from a free memory pool and frees memory if the memory is required no more at execution time. Thus, DMA refers allocating and freeing memory at execution time (or run time).

There are four frequently used library functions `malloc()`, `calloc()`, `free()` and `realloc()` for memory management. These functions are defined within header file `stdlib.h` and `alloc.h`.

i. malloc() function

It allocates requested size of bytes and returns a pointer to the first byte of the allocated space to the program. Its syntax is as

```
ptr=(data_type*) malloc(size_of_block);
```

Here, `ptr` is a pointer of type `data_type`. The function `malloc()` returns a pointer to an area of memory with size `size_of_block`.

For example:

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

Here, the `sizeof(int)`)gives 2. Thus, memory space equivalent to `100*2` bytes (i.e. `200 bytes`) is reserved and the address of the first byte of the allocated memory is assigned to the pointer `x` of type `int` (i.e. `x` refers the first address of allocated memory).

ii. calloc() function

The function `calloc()` provides access to the C memory heap, which is available for dynamic allocation of variable-sized blocks of memory. Unlike `malloc()`, the function `calloc()` accepts two arguments: `no_of_blocks` and `size_of_each_block`. The parameter `no_of_blocks` specifies the number of items for which the memory is to be allocated and `size_of_each_block` specifies the memory size of each item. Thus, the function `calloc()` allocates multiple blocks of storage, each of the same size and then sets all allocated bytes to zero. One important difference between `malloc()` and `calloc()` function is that `calloc()` initializes all the bytes in allocated memory block to zero value.Thus, it is normally used for requesting memory space at run time for storing derived data types such as arrays and structure. Its syntax is:

```
ptr=(data_type*) calloc(no_of_blocks, size_of_each_block);
```

For example:

```
x=(int*) calloc(5, 10*sizeof(int));
```

or

```
x=(int*) calloc(5, 20);
```

The above statement allocates contiguous memory space for 5 blocks, each of size 20 bytes (i.e. we can store 5 arrays, each of 10 elements of integer types).

iii. free() function

The built-in function frees previously allocated memory space allocated by `calloc()`, `malloc()` or `realloc()` function. The memory, dynamically allocated, is not returned to the system until the programmer returns the memory explicitly. The deallocation of memory can be performed using `free()` function. Thus, the function `free()` is used to release the memory space when it is not required. Its syntax is:

```
free(ptr);
```

Here, `ptr` is a pointer to a memory block which has already been allocated by `malloc()`, `calloc()` or `realloc()` function.

iv. realloc();

The function `realloc()` is used to modify the size of previously allocated memory space. Sometimes, the previously allocated memory is not sufficient; we need additional space and sometime the allocated memory is much larger than necessary. In both situations, we can change the memory size already allocated with the help of function `realloc()`. If the original allocation is done by the statement: `ptr=malloc(size);`, then the reallocation of space may be done by the statement: `ptr=realloc(ptr,newsize);`

The function `realloc()` allocates a new memory space of size `newsize` to the pointer variable `ptr` and returns a pointer to the first byte of the new memory block and on failure the function returns `NULL`.

Example 9.13

Write a program to illustrate the use of `realloc()` function

```
#include<stdlib.h>
int main()
 {
 char *name;
 name=(char*)malloc(11);
 strcpy(name,"S.P. Sharma");
 printf("\n Name=%s",name);
 name=(char*)realloc(name,23);
 strcpy(name,"Mr. Shyam Prasad Sharma");
 printf("\n Name=%s",name);
 getch();
 return 0;
 }
```

Output

```
Name=S.P. Sharma
Name=Mr. Shyam Prasad Sharma
```

Example 9.14

Write a program to read number of students from user and then read marks of each student. Display entered marks and their average value. Use pointer instead of conventional array to represent marks of different students.

```
int main()
    {
     int n,i;
     float *p,sum=0,avg;
     printf("How many students are there?\t");
     scanf("%d",&n);
     printf("Enter marks of each student\n");
     p=(float*)malloc(n*sizeof(float));
```

```
    for(i=0;i<n;i++)
        {
          scanf("%f",p+i);
          sum+=*(p+i);
        }
     avg=sum/n;
     printf("\n The average marks of");
     for(i=0;i<n;i++)
       printf("\t%.2f",*(p+i));
    printf(" is:%.2f",avg);
    free(p);
    getch();
    return 0;
    }
```

Output

```
How many students are there? 5
Enter marks of each student
  45.5  56  89  90.5  47
The average marks of 45.50   56.00   89.00   90.50   47.00 is:65.60
```

This program asks number of students from users. According to number of students entered by the user, it allocates memory exactly required to store the marks of the students. This program works for any number of students efficiently.

Example 9.15

Write a function that takes an 1-D array of n numbers and sort the elements in ascending order. Use dynamic memory allocation.

```
void get(float *, int);
void display(float *, int);
void sort(float *, int);
int main(void)
  {
  int n;
  float *num;
  printf("\nEnter number of elements in your array\t");
  scanf("%d",&n);
  num=(float *)malloc(n*sizeof(float));
  get(num,n);
  sort(num,n);
  display(num,n);
  getch();
  return 0;
  }
void get(float *nums, int n)
  {
```

```
    int i;
    printf("\nEnter %d numbers:",n);
    for(i=0;i<n;i++)
       scanf("%f",nums+i);
    }
void sort(float *nums, int n)
   {
    float temp;
    int i,j;
    for(i=0;i<n-1;i++)
       {
        for(j=i+1;j<n;j++)
     {
     if(*(nums+i)>*(nums+j))
       {
       temp=*(nums+i);
       *(nums+i)=*(nums+j);
       *(nums+j)=temp;
       }
      }
     }
   }
void display(float *nums, int n)
   {
   int i;
   printf("\nThe numbers in ascending order:\n");
   for(i=0;i<n;i++)
    printf("\t%.2f",*(nums+i));
   }
```

Output

```
Enter number of elements in your array  8
Enter 8 numbers:12 56 7 90 32 6 54 87
The numbers in ascending order:
 6.00    7.00    12.00    32.00    54.00    56.00    87.00    90.00
```

Example 9.16

Write a program to read an array of n integers using dynamic memory allocation, and display the largest and smallest element among them.

```
int main(void)
   {
   int n,i;
   int *num,max,min;
   printf("Enter number of elements in your array:");
   scanf("%d",&n);
```

```
num=(int *)calloc(n,sizeof(int));
printf("Enter %d integers:",n);
for(i=0;i<n;i++)
  scanf("%d",num+i);
max=*num;
min=*num;
for(i=0;i<n;i++)
   {
   if(max<*(num+i))
     max=*(num+i);
   if(min>*(num+i))
     min=*(num+i);
   }
printf("\nThe maximum number is:%d",max);
printf("\nThe minimum number is:%d",min);
getch();
return 0;
}
```

Output

```
Enter number of elements in your array:  10
Enter 10 integers:12  67  89  900  54  32  123  77  43  22
The maximum number is:900
The minimum number is:12
```

9.15 Applications of Pointer

There are various application areas of pointers. Some of them are:

1) Pointer is widely used in Dynamic Memory Allocation. The functions used for allocation of memory at run time return the address of allocated memory using pointer.

2) Pointers can be used to pass information back and forth between a function and its reference point (i.e. calling program). Pointers provide a way to return multiple values from a called function to calling function using call by reference (i.e. passing arguments to function by reference or address).

3) Pointers provide an alternative way to access individual array elements. They used to manipulate arrays more easily by using pointers instead of using the arrays themselves.

4) They increase the execution speed as they refer address.

5) They are used to create complex data structure such as linked list, trees, graphs, and so on.