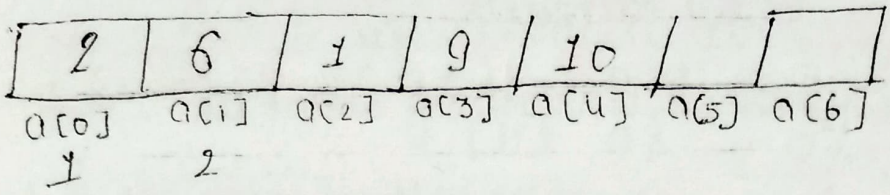


15

1. To find i = 2 and odd in an array.
2. To find odd or even in an array.

* Searching in the process of finding the location of the specified element in a list. The specified element is often called the search key. If the process of searching finds a match of the search key with a list element value, the search said to be successful, otherwise it is unsuccessful. The two most commonly used search techniques are -

- Sequential search or linear search (as applicable in both type of array's sorted and unsorted)
- Binary search:- (as applicable sorted arrays)

) linear search :-

3. WAP to search a number using linear search.

```
void main ( )
{
int a [20], i, loc, n, item, c = 0;
Print f ("Specify the number of elements");
Scan f ("%d", &n);
```

```
printf (" Enter elements ");
for (i=0; i<n; i++)
scanf ("%d", &a[i])
printf (" enter elements which you want to
          search ");
scanf ("%d", &item);
for (i=0; i<n; i++)
{
if (a[i] = item)
{
loc = i;
printf ("\n %d is present at location %d",
          item, loc+1);
    c++;
  }
}
  if (c!=0)
printf ("\n the item is present %d times",
          c);
else
printf ("\n the item is not present ");
getch ( );
}
```

② WAP to reverse the array.

```
void main ( )
{
int a[10], n, i, temp, j;
printf ("enter the size of array");
scanf ("%d", &n);
printf ("\n enter the elements");
for (i=0; i<n; i++)
{
 scanf ("%d", &n[i]);
}
for (i=0; j=n-1; i<n/2; i++, j--)
{
 temp = a[i];
  a[i] = a[j];
  a[j] = temp;
}
printf ("\n elements after reverse");
  for (i=0; i<n; i++)
{
  printf ("%d \n", a[i]);
}
getch ( );
}
```

procedural

Binary Search :- The prerequisites for binary search is that the array should be sorted. Initially we compare the item to be searched with the middle element of the array. If the item is same as the middle element, then the search is successful. Otherwise the array is divided into two portion (contains all elements

to the left of the middle element and the other one consists of all the elements on the right side of the element.

Since the array is sorted, all the elements in the left portion will be smaller than the middle element & the elements in the right portion will be greater than the middle element.

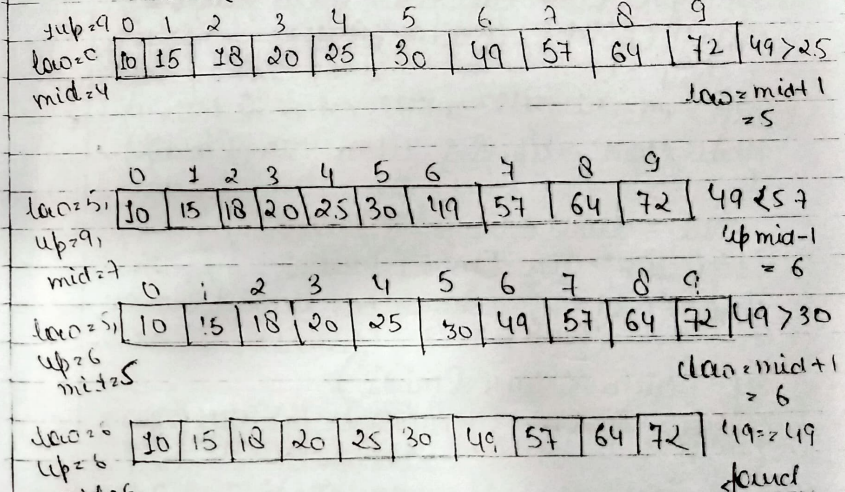
Now, if the item to be searched is less than the middle element then we search it in the left portion of the array and if it is greater than the middle element then search will be in the right portion of the array.

Procedure:- Take 3 variables i.e. low, up and mid. The value of the mid will be evaluated as -

$$mid = [low + up] / 2$$

if item > arr[mid], search will resume in right portion.
low = mid + 1, up will remain same.
if item < arr[mid], search will resume in left portion.
up = mid - 1; low will remain same.
if item = arr[mid], search is successful item found at mid position.
if low > up, search is unsuccessful item not found at mid position. i.e. array.
if low up

Ex:-
Let us take a sorted array of 10 elements. Suppose the element that we are searching is 49.



// WAP to search an element through binary search.

```
# define size 10
# include <stdio.h>
# include <conio.h>
void main()
```

```
{
int arr [SIZE];
int low, up, mid, i, item;
printf ("Enter elements of the array (in
    -Sorted order) \n");
for (i=0; i<size; i++)
scanf ("%d", &arr[i]);
printf (" Enter the item to be searched");
scanf ("%d", &item);
low = 0;
up = size - 1;
while low <= up && item != arr[mid]
{
 mid = (low + up)/2;
 if (item > arr [mid])
low = mid + 1; // Search in right portion

if (item < arr [mid])
  up = mid - 1; // Search in left portion
}
if (item == arr [mid])
printf ("%d found at position %d \n", item,
                                    mid+1);
    if (low > up)
 printf ("%d is not found in array", item);
}
```

Ans WAP to copy the Calculate of one array into another in the reverse order.

```
void main ( )
{
int arr1 [5], arr2 [5], i, j;
printf (" Enter 5 elements of array");
for (i=0; i<=4; i++)
scanf ("%d", &arr [i]);
for (i=0, j=4; i<=4; i++, j--)
 arr2 [j] = arr [i]
printf (" In elements in reverse order");
   for (i=0; i<=4; i++)
printf (" In %d ", arr [i]);
getch ( );
}
```

Concatenation of 2 linear Array.

Contenation means joining the elements of 2 arrays to form a new array.

```
void main ( )
{
int a[20], b[20], c[40], m, n, i, j;
printf (" Enter size of first array & second
                                    array");
scanf ("%d %d", &m, &n);
printf (" Enter elements of first array");
  for (i=0; i<10; i++)
scanf ("%d", &a [i]);
printf (" Enter elements of two array");
  for (i=0 ; i<n)
scanf ("%d", &b [i]);
```

```
for (i=0; i<m; i++) // Copy array elements
    c[i] = a[i];
for (i=0;
```

```
for (i=0; i<n; i++) // Copy array 2 elements
    c[m+i] = b[i];
printf ("\n\n Concetenated array is \n\n");
for (i=0; i<(m+n); i++)
printf ("%d", c[i]);
getch ();
}
```

2) Merging of two sorted Arrays :-

Merging is the process of combining two or more sorted Arrays into another array which is also called sorted.

array A

10	20	30	40	50
0	1	2	3	4

B

5	33	52	80
0	1	2	3

C

5	10	20	30	33	40	50	52	80
0	1	2	3	4	5	6	7	8

WAP Merge 2 sorted arrays into 3rd sorted array

```
void main ()
{
int a[10], b[10], c[20], m, n, i=0, j=0, k=0,
                    u, p;
printf ("Size of first array");
scanf ("%d", &m);
printf ("size of second array");
scanf ("%d", &b);
printf ("enter sorted elements in 1st array");
for (p=0 to m)
scanf ("%d", &a[i]);
printf ("Second array");
for (p=0 to n)
scanf ("%d", &b[i]);
while (i<m) && (j<n))
{
if (a[i] < = b[j])
{
 c[k] = a[i]; // Copy element a to c
     i++;
}
else
 {
  c[k] = b[j]; // Copy element b to c
     j++;
}
```

```
k++;   // move to next position in array c
}
if (i==m) // When array 'a' has
                been finished.
{
for (u=j; u<n; u++)
{
c[k] = b[u];
k++;
}
}
else
{
   for (u=i ! u<m ! u++)
   {
   c[k] = a[u];
     k++;
   }
} for (i=0; i<(m+n); i++)
   Pf("%d", c[i]);
```

NAP to append next array into previous array.

```
void main ()
{
int a[20][20], i, j, n, m;
Printf (" Enter size of a & b ");
Scanf (" %d %d ", &m, &n);
Printf (" Enter array elements ");
   for (i=0; i<m; i++)
Scanf (" %d ", &a[i]);
Printf (" Enter Second array elements ");
   for (i=0; i<n)
Scanf (" %d ", &b[i]);
if ( (m+n) <= 20))
{
for (i=m, j=0; j<m+n; i++, j++)
   a[i] = b[j];
Printf (" Append list is \n");
   for (i=0; i<(m+n); i++);
Printf ("\n %d", a[i]);
}
else
   Printf (" Append & Concatenation is
                not possible");
getch ();
}
```

Sorting :→ Sorting means arranging the elements in some specific order, that is, either ascending or descending order. The various sorting techniques available are :-

1) Selection sort ii) Bubble sort iii) Insertion sort 4) Quick sort v) Shell sort 6) Merge sort vii) Heap sort & etc.

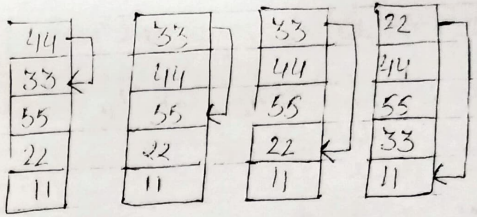
ii) Selection sort :→ In this method we perform a search in the array, starting from the first element, to find the position of element with the smallest value. If found, it is swapped with the first element in the array. As a result of this interchange, the smallest element is placed in the first position of the array and so on.

N-1 passes are required in the sorting technique as each pass places one element properly.

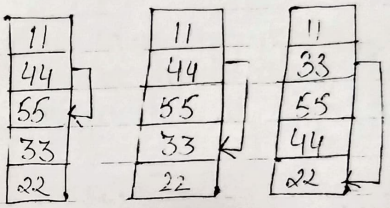
for ex :- Consider the following array A having 5 elements.

44	33	55	22	11

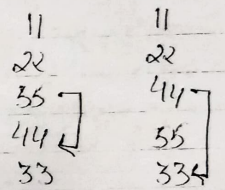
Iteration 1



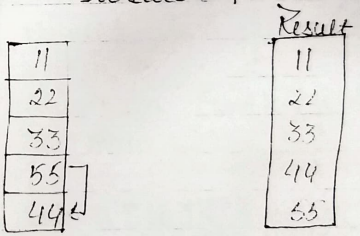
Iteration 2



Iteration 3



Iteration 4



Q WAP to sort an array using selection sort.

```
void main ( )
{
int a[20], n, i, temp, j, m, n, loc;
```

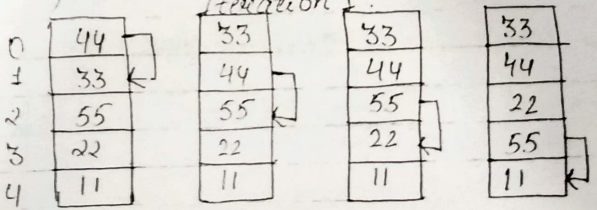
```
printf("enter array size");
scanf("%d", &n);
printf("enter array elements");
for(i=0; i<n; i++);
scanf("%d", &a[i]);
for(i=0; i<n; i++)
{
    min = a[i];
    loc = i;
for(j=i+1; j<n; j++)
{
    if(a[j] < min)
    {
        min = a[j]
        loc = j;
    }
}
    temp = a[i];
    a[i] = a[loc];
    a[loc] = temp;
}
printf("\n Sorted array is");
    for(i=0 to n)
printf("\n %d", a[i]);
}
    or
for(i=0; i<n; i++)
{
    for(j=i+1; j<n; j++)
    {
        if(a[j] < a[i])
        {
            temp = a[j];
            a[j] = a[i];
            a[i] = temp;
        }
    }
}
```

ii) Bubble Sort / Exchange Sort :→ In bubble sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the position of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array.

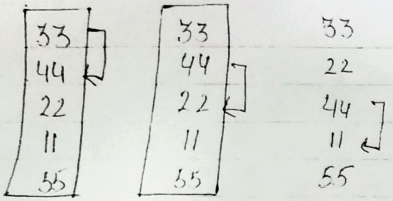
The first pass places the largest element in the array at the last location.

for ex:-

Iteration 1.



Iteration 2.



(3)

33	22
22	33
11	11
44	44
55	55

(4) Result

22	11
11	22
33	33
44	44
55	55

// WAP to implement Bubble Sort

```
void main ( )
{
int a[20], n, i, temp, j;
Printf ("Enter Size");
Scanf ("%d", &n);
Printf ("enter elements");
for (i=0; i<n-1; i++)
{
  for (j=0; j<n-i-1; j++)
  {
    if (a[j] > a[j+1])
    {
      temp = a[j];
      a[j] = a[j+1];
      a[j+1] = a[temp];
    }
  }
}
Printf ("Sorted array is \n");
for (i=0 to n)
Printf ("\n %d", a[i]);
}
```


Passing array elements as Arguments :-

like other arguments, an array element can also be passed to a function by calling the function by value or by reference.

for ex :-

Ques WAP to demonstrate the passing of an array element to a function by value.

```
void even (int k);
void main ( )
{
int a[10] = {10,20,30,40,50,60,70,80,90,100};
int i;
for (i=0; i<10; i++)
{
  if (even (a[i])) == even
  Printf ("Number %d is even\n", a[i]);
  else
  Printf ("Number %d is odd \n", a[i]);
```

```
printf("enter array size");
scanf("%d",&n);
printf("enter array elements");
for(i=0; i<n; i++);
scanf("%d", &a[i]);
for(i=0; i<n; i++)
{
   min = a[i];
   loc = i;
   for(j=i+1; j<n; j++)
   {
      if(a[j] < min)
      {
         min = a[j]
         loc = j;
      }
   }
      temp = a[i];
      a[i] = a[loc];
      a[loc] = temp;
}
printf("In Sorted array is");
   for(i=0 to n)
printf("In %d", a(i));
}
```

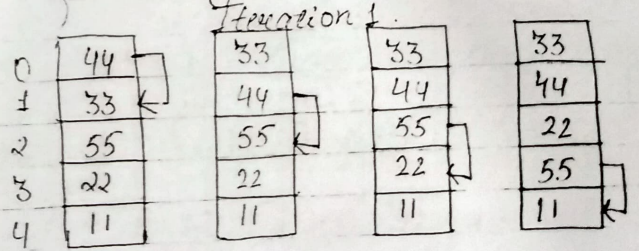
or

```
for(i=0; i<n; i++)
{
   if(a[j] < a[i])
   {
      temp = a[j];
      a[j] = a[i];
      a[i] = temp;
   }
}
}
```

ii) Bubble Sort / Exchange Sort :→ In Bubble Sort, each element is compared with its adjacent element. If the first element is larger than the second one, then the position of the elements are interchanged, otherwise it is not changed. Then next element is compared with its adjacent element and the same process is repeated for all the elements in the array.

The first pass places the largest element in the array at the last location.

for ex:-



Iteration 2:

```
}
}

// function Definition.
unit even (int k)
{
  if (k%2 == 0)
     return 1;
  else
     return 0;
}
```

Passing entire array as an argument :-

Q) WAP to demonstrate the passing of an entire array using subscript notation.

```
# define Num 10
unit max (int b[ ], unit size);
void main ( )
{
unit a [num] = { 10,20,30,40,50,5,4,3,2,1};
        unit i;
printf (" Largest of the array element is
                %d\n", max(a,num));
}
  int max (int b[ ], unit size);
{
unit n,x,i;
big = b[0];
for (i=1; i< size; i++)
{
   if (b[i] > big)
      big = b[i];
}
   return big;
}
```

②

```
void main ( )
{
int num [ ] = { 2,4,6,8 };
 display (& num [0],4);
}
 void display (int * j , int n)
{
   int i;
   for (i=0; i<n; i++)
{
  printf ("In element = %d", * j);
   j++;     // increment pointer to point to
                next location.
      }
}
```

Note that when array is passed as an argument, the address of the array address of the first element is passed. the addresses of the remaining elements are computed at executed time. Hence, by default array is always passed by address.

9 WAP Input, output, linear search, insertion deletion, selection sort, bubble sort.

```
void main ( )
{
 int a [20], n;
 clrscr ( );
 printf ("Enter size of an array");
 scanf (" %d", &n);
     input (a,n);
     display (a,n);
     getch ( );
}
  void input ( int b [ ], int size)
{
   int i;
   for (i=0; i<size; i++)
   {
   printf ("enter [%d] value \n", i+1);
   scanf (" %d", &b [i]);
   }
   return;
}
void display (int b[ ], int size)
{
   int i;
printf (" In the array is ");
   for (i=0; i<n; i++)
printf (" %d In", a[i]);
   return;
}
```

* Two dimensional Arrays :-

Two dimensional array, the most common multi dimensional arrays are used to store information that are normally represent table form,

Ex :-
$$\begin{bmatrix} 2 & 4 & 3 \\ 6 & 4 & 8 \end{bmatrix}$$

Declaration of 2-d arrays :- The general syntax for declaration of 2-d - array is as follows -

data type <array name> [size of row] [size of column];

Ex :- int a[3] [5];

Array Initialization :- We can declare & initialize an array as follows -

```
int A[3][4] = {{ 0,2,6,5 } int a[2][2]
               { 1,2,3,4 }, or {1,2,3,4};
               { 6,8,10,12}};
```

Memory for the array may be visualized as follows :-

	Col 0	Col 1	Col 2	Col 3
row 0	0	2	6	5
row 1	1	2	3	4
row 2	6	8	10	12

It is important to number that, while initializing a 2-D array, it is necessary to mention the second (column) dimension whereas the first dimension (row) is optional.

for ex:- int an [ ][3]= { 12,34,55,45, 46, 45};

Accessing Of Array :-

int a[2][1] = 8;

Memory Map of a 2-D Array :-

The array arrangement shown in figure is only conceptually. this is because memory does not contain rows & columns. In memory, whether it is a one-dimensional or a 2-D - array, the array elements are stored in one continuous chain. the arrangement of array elements of a 2-D array in memory is shown below.

a[3][0]

a[0][0]	a[0][1]	a[1][2]	a[1][1]	a[2][0]	a[2][1]	a[3][2]
15	16	18	20	22	24	26	28

1. WAP to input a 3*3 matrix and print the elements in matrix form
2. Addition, Subtraction
3. Multiplication
4. Transpose
5. Addition of upper diagonal, diagonal, below diagonal

```
int a[3][3], i, j;
clrscr();
printf("enter rows & col. size");
scanf("%d %d", &r, &c);
for(i=0, i<3, i++)
{
```

```
        printf("%2d", b[i][j]);
    }
    printf("\n");
}
```

Address calculation in one-D array.

The array elements are stored in contiguous memory locations by sequential allocation technique. The address of ith element of the array can be obtained if we know

1. The starting address, that is the address of the first element called Base address denoted by B.
2. The size of the element in the array denoted by S.

Consider the array A in which LB <= i <= UB, that is A[LB:UB] where LB denotes the lower bound of the index and UB upper bound of index. The address of the ith element is given by.

Formula → address of A[i] = B + (i - LB) * S

for ex:- (1) given an array A[0:15] if B = 1000 & S = 2 then. Calculate the address of A[10].

Ex: Base address B = 1000
S = 2

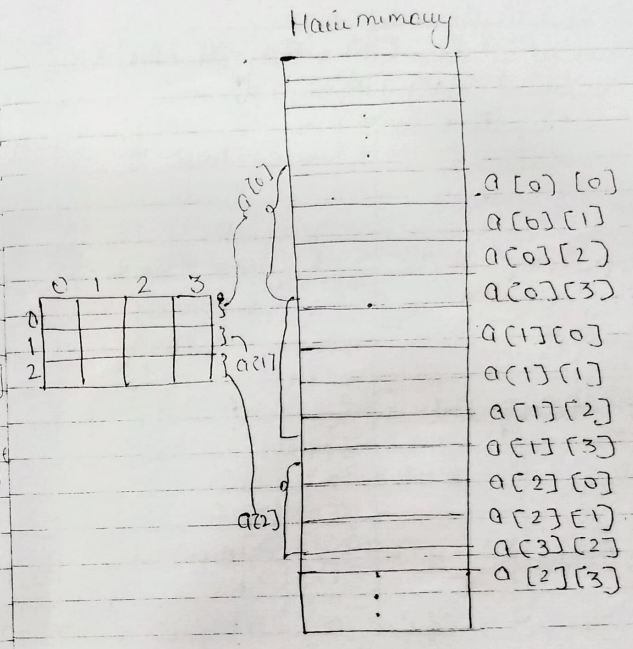
A[ii] = B + (ii - LB) * S
A[10] = 1000 + (10 - 0) * 2
= 1020.

Two-dimensional array -

Sequential allocation for 2-D array -
Suppose we have a 2-D array a[0:2, 0:3] of type int.
An integer requires two bytes of storage.
Since the main memory of a computer is linear, two dimensional elements array cannot be stored in its natural grid form. The array elements are stored linearly using one of the following methods.

i) Row major storage.
ii) Column major storage.

The Row major storage is shown in below figure. Using this method an 2-D array is stored with all the elements of first row 2-D array in sequence followed by the elements of second row & so on.



Row major storage

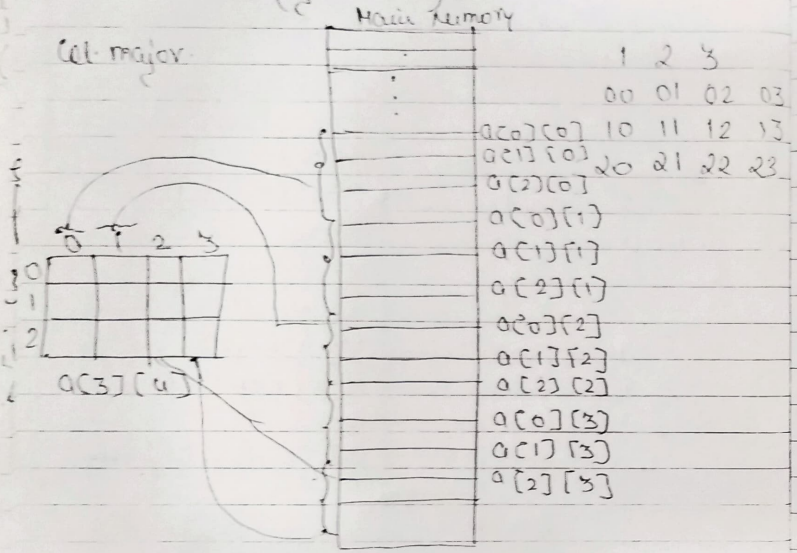
Address Calculation of elements of Array
a[LB1 : UB1, LB2 : UB2]

Let i, j denote the index of row & column
Where LB1 <= i <= UB1 & LB2 <= j <= UB2
Address of an element = B + (number of elements before ii) * S
Where B is base address & S denotes size of each element.

* Also the no. of rows = M = (UB1 - LB1 +
* & the no. of col = N = (UB2 - LB2 +

Using row major order the address of a[i][j] is given by

address of a[i][j] = B + [(i - LB1) * N + (j - LB2)] * S



Using col major order the address of a[i][j] is given by,

Address of a[i][j] = B + [(i - LB1) + [(j - LB2) * H]] * S

for ex:- An array a[7][20] is stored

(32)

in the memory with col. major order with each element requiring 2 bytes of storage if the base address of array is 2000, calculate the location of X[3][5] when the array X is stored in column major order.

Sol Here, B = 2000

S = 2 bytes.

No. of rows M = 6 - 0 + 1 = 7 ∴ M = UB1 - LB1 + 1

No. of col. N = 19 - 0 + 1 = 20 i.e N UB2 - LB2 + 1

The array is stored in col. major order.

Address of X[i][j] = B + [(i - LB1) + (j - LB2) * M] * S

X[3][5] = 2000 + [(3 - 0) + (5 - 0) * 7] * 2

= 2076. Row major answer is 2150.

Using col major order the address of a[i][j] is given by,

Address of a[i][j] = B + [(i - LB1) + [(j - LB2) * H]] * S

for ex:- An array a[7][20] is stored

Ex 2 The array A[20][10] is stored in the memory with row major order with each element requiring one byte of storage if the base address of A is 50 Determine the location of A[10][5] is zero.

~~Note~~ - Note- A[20][10] means Valid row are 0 to 19 & Valid col. are 0 to 9.

Sol
B = C0
S = 1 byte
H = 19-0+1 = 20
N = 9-0+1 = 10

The array is stored in row major order

$A[i][j] = B\{[i-(B_1)]*N+(j-(B_2))]*S\}$

$A[10][5] = C0 + [(10-0)*10 + (5-0)]*1$

$2000 = C0 + [100+5]*1 = C0 + 105$
$C0 = 2000 - 105 = 1895.$

Array of Pointers.
The array element can be an array of ints or an array of floats, similarly it can be an array of pointers. Since a pointer variable always contains an address, an array of pointers would be nothing but a collection of addresses.

for ex:- Void main()
{ int * arr[4]; // array of

```
int i = 31, j = 5, k = 19, l = 71, m;
arr[0] = &i
arr[1] = &j
arr[2] = &k;
arr[3] = &l;

for (m=0; m<=3; m++)
printf("%d\n", *arr[m]));
getch();
}
```

Passing 2-D array to fun^n -

```
Void display(int *q, int, int);
Void show(int (*q)[4], int, int);
Void main()
{ int a[3][4] = {1,2,3,4,5,6,7,8,9,0,16,
                 00 01 02 03 10 11 12 13 20 21 22 23};
display(a,3,4);
show(a,3,4);
print(a,3,4);
}
Void display(int *q, int k, int l)
{ int i,j;
for(i=0; i<k; i++)
{
for(j=0; j<l; j++)
printf("%8d", *(q+i*col+j));
printf("\n");
}
```

// general formula
base add + row * no of col + col no.

Pointers

Definition

A pointer is a memory variable that stores a memory address. Pointer can have any name that is legal for other variable & it is declared in the same fashion like other variables but it is always denoted by '*' operator.

There are two new operators you will need to know to work with pointer. The "address of operator '&' and the "indirection" or "dereferencing" operator '*'. Both are prefix unary operators.

When you place an ampersand in front of a variable you will get its address, this can be stored in a pointer variable. When you place an asterisk in front of a pointer you will get the value at the memory address pointed to.

for ex:-

a	Ptr. 1	Ptr. 2
5	1025	5000
1025	5000	6025

features :-

1. Saves memory space
2. Access memory efficiently. Dynamic memory is located.
3. handle arrays and data table efficiently
4. Support dynamic memory management
5. Pointers helps to return multiple values from a function through function argument
6. Increase prog. execution speed. Pointers is an effecient tool for manipula-ting structures linked lists, queries, stacks etc.

Advantages.

1. function can't return more than one value. But when the same function can modify many pointer variables & function as it is returning more than one variable. returning returning
2. In the case of arrays, we can decide the size of the array at run time by allocating the necessary space.
3. In the case of pointers to classes, we can polymorphism & virtual classes to change the behaviour of pointers to various types of classes at run time

Disadvantage

1. If sufficient memory is not available during runtime for the storage of pointers, the prog. may crash. (least possible)

2. If the programmer is not careful and consistent with the use of pointers, the prog. may crash (very possible).

Declaring a pointer variable -

Syntax - data type * pointer variable

We can declare pointers like this -

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

Prog 1. WAP to illustrate the concept of accessing the address of pointer.

```
void main ( )
{
int *ptr1, a;
  a=5
  ptr1 = &a;
printf ("In value %d is stored at addr
        -ess %u", a, ptr1);
}
```

Output - Value 5 is stored at address 1025.

Pointer Operators (* and &)

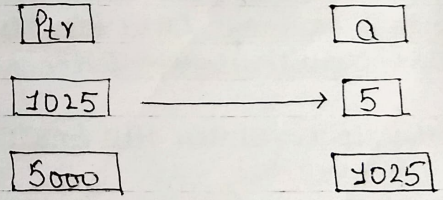
int a = 5

a → Memory location name
5 → Value at memory loc.
1025 → Memory location address.

For ex:- Simple pointer declaration & def.

```
int a = 5;
int *ptr;
ptr = &a;
```

Pictorial representation -



Explanation -

About variable a;

1. Name of variable - a
2. Value of variable which it keeps - 5

memory - 1025 (assume)

-Same as ptr

1. Name of variable a;
2. Value of variable which it keeps - 5
3. Address where it has stored in memory - 1025 (assume).

Pwog of Pointer

Pointer to Pointers -

A pointer is variable that holds the address of another variable. this concept can be further extended. we can have a variable that hold an address of a variable that in turn holds an address of another variable. This type of variable will known as pointer to a pointer or it is also known as double indirection.

A pointer to a pointer will be declare -d as

int ** x, ptr;

The value pointed to by a person pointer to another pointer when be accessed as

** ptr;

Q HAP to illustrate the concept of double pointe -r.

Concatenation rule of above two pointers - * and &.

* and & operators always cancel to each other i.e

* & p = p

But it is not right to write:

& * p = p

for ex:- explain use of pointer to pointer -

```
void main ()
{
int x = 25;
int * ptr = &x;   // stmt one
int **temp = & ptr;  // stmt two
printf (" %d %d %d ", x, * ptr, ** temp);
}
```

Output

25 25 25

Pointer Arithmetic -

The following operations are permitted on pointers.

1. Addition of a number to a pointer variable -

Suppose P is a pointer variable pointing to an element of integer type, then the stmt.

P++; or ++P;

increments the value of P by a factor of 2, so that it points to the next location that holds another value of integer type. this increment factor will be 1 for character, 4 for long integer & float & 8 for long float.

The Statement

P+ = i

Where i is either a positive integer Constant or an integer variable

2. Subtraction of a no. from a pointer

Suppose P is a pointer variable pointing to an element of integer type, then the stmt

P--; or --P;

Decrements the value of P by a factor of 2, so that it now points to the location preceding the current location.

The statement

P- = i;

Where i is either a positive integer constant or an integer variable.

3. Subtraction of one pointer variable from another:

One ptr variable can be subtracted from another provided both point to the same data type. The difference of the two indicates the number of bytes separating the corresponding elements.

The following arithmetic operations on pointers are not permitted.

- Addition of two ptr. variables
- Multiplication of a pointer variable by a number.
- Division of a pointer variable by a number.

8. WAP to illustrate the concept of displaying the address of a ptr variable

```
int a, * Ptr ;
  a = 5;
Ptr = &a;
Printf ("In value %d is stored at address %u", a, &a);
Printf ("In value %d is stored at address %u", a Ptr);
Printf ("In value %d is stored at address %u", * Ptr, Ptr);
Printf ("In value %d is stored at address %u", * (&a), &a);
```

Output - value 5 is stored at address 1025
"
"
"

9. WAP to display the use of pointers.

```
Char c;
int a;
float b;
c = 'u'; a = 412; b = 4.12;
Printf ("In value %c is stored at address %u", c, &c);
Printf ("In value %d is stored at address %u", a, &a);
Printf ("In value %f is stored at address %u", b, &b);
}
```

Output
value d is stored at address 5000
"
"

// Pointers Arithmetic

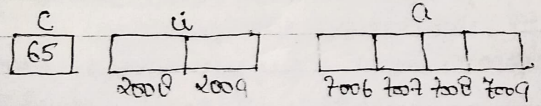
```
Void main ( )
{
int i = 3, * x;
float j = 1.5, * y;
Char k = 'c', * z;

Printf ("In value of i = %d", i);
Printf ("In value of j = %f", j);
Printf ("In value of k = %c", k);

x = &i;
y = &j;
z = &k;

Printf ("In original value in x = %u", x);
Printf ("In original value in y = %u", y);
Printf ("In original value in z = %u", z);

x++;
y++;
z++;
Printf ("New value in x = %u", x);
```

```
printf("New value in y = %u", y);
printf("New value in z = %u", z);
}
```

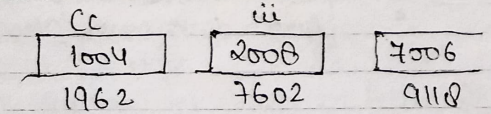
Output- Value of ii = 3
" " i = 4.500000
" " k = C

Original Value in x = 1002
" " " y = 2004
" " " z = 5006

New value in x = 1004
" " " y = 2008
" " " z = 5007



Pointer




```
Char c, * cc
int ii, * ii
```

Debugging Pointers.

Common bugs related to pointers and memory management is dangling pointers, memory leaks and allocation failures.

1) Problem of dangling/wild Points:-
- The Programmer fails to initialize a pointer with a valid address.
- If the programmer assign a value to such pointer, the value will then write the prog. or os instructions, and the computer program may show undesirable behaviour i.e stop responding.

2) Problem of Null Pointer Assignment.
- One particular situation that happens is when the pointers points to address 0, which is called NULL.
- for ex - if the pointer variable is declared as global since global variables are initialized to 0.
- When it happens, then the system will display a message "Null pointer assignment" on termination of the prog.

3) Problem of Memory Leaks :-

Memory leak is a situation where the programmer fails to release the memory allocated at run time in a module.

4) Problem of Allocation failure.

- An allocation failure is a situation when the prog. through malloc ( ), calloc ( ), or realloc ( ) fun^n request you a block of memory, & the OS could not fulfill the request of the prog. bcoz the sufficient memory may not be available in the free pool.

Dynamic Memory Management -

C language provides a set of library functions called dynamic Memory Management functions to allocate and deallocate dynamic Memory at execution time as dynamically.

By allocation, we mean that your prog. can obtain as much memory as required by your prog. even during execution of your prog. By de-allocation we mean that the memory acquired dynamically can be released at any time during your prog. execution.

Memory Management functions:

functions Name.	Description
Malloc	Allocates memory requests size of bytes and returns a pointer to the 1st byte of allocated space.
Calloc	Allocates space for an array of elements initializes them to zero & returns a pointer to the memory.
free -	free previously allocated space.
realloc -	Modifies the size of previously allocated space.