Array Pointers & Structures

- DS are classified as linear or non linear.
- A DS is said to be linear if its elements form a sequence, or, in other words 'a linear list'.
- There are 2 basic ways of representing linear structures in memory:
  1) To have the linear relationship b/w the elements represented by means of sequential memory location. These are called arrays.
  2) To have the linear relationship between the elements represented by means of pointers or links. These are called linked lists.

Linear Arrays

- A linear array is a list of a finite number n of homogeneous data elements (i.e data elements of the same type) such that:
  a) The elements of the array are referenced respectively by an index set consisting of n consecutive numbers.
  b) The elements of the array are stored respectively in successive memory locations.
- The no. n of elements is called the length or size of the array.

$$\text{Length} = UB - LB + 1$$

where UB (largest index) = Upper bound
LB (smallest index) = Lower bound.

- Representation of elements of an array A can be done in following notations:
  1) Subscript notation — $A_1, A_2, A_3 \ldots A_n$

Representation of linear Arrays in Memory

- Let LA be a linear array in the memory of the computer.
- The notation:

LOC (LA[K]) = address of the element LA[K] of the array LA.

- The elements of LA are stored in successive memory cells.
- The computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, which is denoted by

Base (LA).

- By using this address Base (LA), the computer calculates the address of any element of LA by using the following formula:

$$LOC(LA[K]) = Base(LA) + w(K - lower\ bound) \rightarrow (1)$$

where LOC (LA[K]) = Address of the element LA[K] of the array LA

Base (LA) = Base address of LA

w = The no. of words per memory cell for the array.

1 One Dimensional Array

1.1 Declaration of 1-D Array

- Like other simple variables, arrays should also be declared before they are used in the prog.
- Syntax: data-type array-name [size];
- array-name denotes the name of the array & it can be any valid C identifier.
- data-type is the data type of the elements of array.
- size of the array specifies the no. of elements that can be stored in the array. It can be a positive integer constant or constant integer expression.

- Examples: int age [100];
  float salary [15];
  char grade [20];

  Individual elements of the above arrays are:

  age[0], age[1], age[2], ------ age[99].

  salary[0], salary[1], ------ salary[14]

  grade[0], grade[1], ------ grade[19].

- When the array is declared, the compiler allocates space in memory sufficient to hold all the elements of the array, so the size of array should be known at the compile time.
- There can't be use of variables for specifying the size of the array in the declaration.
- The symbolic constants can be used to specify the size of the array.
- Example

```
#define SIZE 10
main ()
    int size = 15;
    float sal [size];
    int marks [size];
```

Traversing Linear Arrays

- 'A' be a collection of data elements stored in the memory of the computer
- If we want to print the contents of each element of A or we want to count the no. of elements of A.
- This can be accomplished by traversing A, i.e by accessing & processing each element of A exactly once.
- Linear structure like array or linked list can easily be traversed.
- Where as traversal of non linear structures like trees & graphs is more complicated.

Algo: 4.1 Here LA is a linear array with LB & UB. This algo traverses LA applying an operation Process to each element of LA

1. [Initialize counter] Set K = LB:
2. Repeat steps 3 & 4 while K ≤ UB
3. [Visit Element]. Apply Process to LA[K].
   [Increase counter] Set K := K+1
4. [End of step 2 loop]
5. Exit

Example:

(a) Find the number NOM of yrs during which more than 300 autos were sold.

1. Set Num = 0 [Initialization step]
2. Repeat for K = 1932 to 1984:
   if Auto[K] > 300, then: Set NUM := NOM + 1
   [End of loop]
3. Return.

1.3 Processing 1-D Arrays

- For processing arrays we generally use a for loop & the loop variable is used at the place of subscript.
- The initial value of loop variable is taken as '0' as array.
- The loop variable is increased by 1 each time so that we can access & process the next element in the array.
- The total no. of passes in the loop will be equal to the no. of elements in the array & in each pass we will process one element.

I Reading values in 'a'

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

II Displaying values of 'a'

```
for (i=0; i<10; i++)
    printf ("%d", a[i]);
```

III Adding all the elements of 'a'

```
sum=0;
for (i=0; i<10; i++)
    sum + a[i];
```

Prog: WAP to I/P values into an array & display them.

```
#include <stdio.h>
#include <conio.h>
main()
{
int a[5], i;
for (i=0; i<5; i++)
{
    printf("Enter the value of a[%d]: ", i);
    scanf("%d", &a[i]);
}
printf("The array elements are: \n");
```

- The traversal of LA by modifying above algo. in order to compute the average of elements of the array.

```
float Avg. (int *a, int n)
{
    int total = 0, i;
    float avg;
    for (i=0; i<n; i++)
        total += a[i];
    avg. = (float) total / n;
    return avg.
}
```

2. Search Operation

- Searching is the proces of finding the location of given elements in the linear array.
- Two approaches to search operation:
  a). Linear search
  b) Binary search.

(a) When the elements are in random order, then we use linear search.

(b) When the elements are ~~sto~~ sorted, then we use binary seach technique

nent iteration

a[mid] = 20715, end = mid - 1 = 3 - 1 = 2, & beg.

remains unchanged.

$$mid = (0 + 2)/2 = 1$$

a[mid] @ a[1] ≠ 15 & beg < end.

nent iteration.

a[mid] 10 < 15, beg = mid + 1 = 1 + 1 = 2. &

end remains unchanged.

$$mid = (2 + 2)/2 = 2$$

a[mid] a[2] = 15 search terminates.

Algo

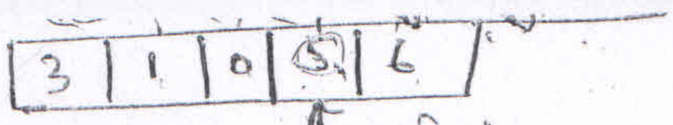
```
Begin
    set beg = 0
    set end = n - 1
    set mid = ( beg + end ) / 2
    while ( (beg ≤ end) && (a[mid] ≠ item) ) do
        if (item < a[mid]) then
            set end = mid - 1
        else
            set beg = mid + 1
        endif
        set mid = ( beg + end ) / 2
    endwhile
    if ( beg > end ) then
        set loc = -1
    else set loc = mid
    endif
End.
```

(a) Inserting at a Given Position.

```
void insert (int *a, int *n, int item, int k)
 {
  int j;
 j = *n - 1;
  while (j >= k)
  {
    a[j+1] = a[j];
     j--;

  }
 a[k] = item;
   (*n)++;

}
```

(b) Inserting in a Sorted Array

```
void insert (int *a, int * n, int item)
 {
   int k;
    k = n* -1;
    while ( (item < a[k]) && (k >=0))
      {
        a[k+] = a[k]; &
        k--;
      }
    a[k+1] = item;
     (*n)++;

}
```



a[j+1] = a[j]

```
j = k+1
while (j <= *n)
{
    a[j-1] = a[j];
    j++;
}
(*n)--;
}
```

(5) Sort Operation.

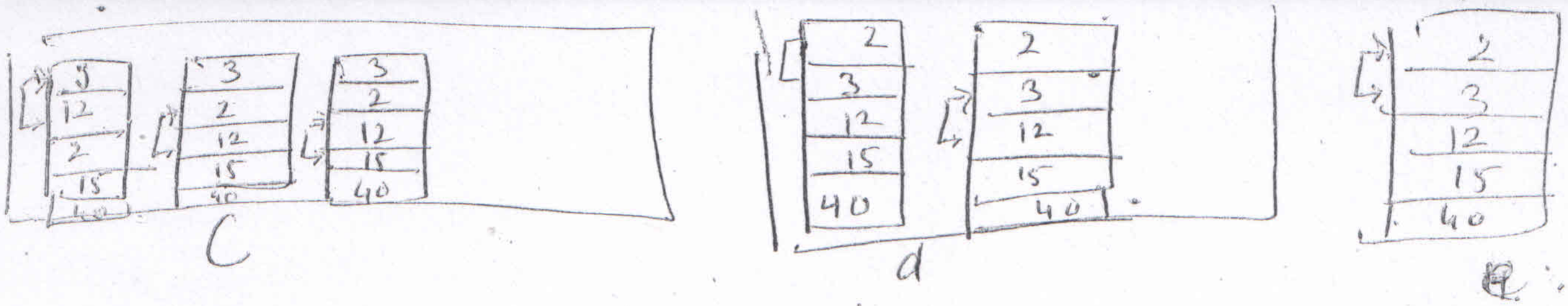
- Sorting is the process of arranging the elements of the array in some logical order.

(a) Bubble Sort Method.

- It requires (n-1) pass to sort an array.
- In each pass every element a[i] is compared with a[i+1], for i = 0 to (n-k), where k is the pass no & if they are out of order i.e if a[i] > a[i+1], then swap.

- Important property.

"Once there is no swapping of elements in a particular pass, there will be no further swapping of elements in the subsequent passes.




```
void Sort (int *a, int n)
{
  int j, k, temp;
  for (k=1; k<n; k++)
  {
    for (j=0; j<n-k; j++)
      if (a[j] > a[j+1])
      {
        temp = a[j];
        a[j] = a[j+1];
        a[j+1] = temp;
      }
  }
}
```

(6) Merge Operation.

- Meging is the process of combining the elements of 2. similar arrays into a single array.

- Example:

Array a

12	40	-3	2

Array b

15	5	7	10	60

Array c

12	40	-3	2	15	5	7	10	60

- One ~~method~~ approach is to join them end to end & then sort the combined array, but this approach is not efficient & economical

- Best approach is to compare the elements of the given array & based on this comparision, decide which element should come first in the third array.

Code

```
void merge(int *a, int m, int *b, int n, int *c)
{
  int na, nb, nc;
  na = nb = nc = 0;
  while ((na<m) && (nb<n))
  {
    if (a[na] < b[nb])
      c[nc] = a[na++];
    else
      c[nc] = b[nb++];
    nc++;
  }
```

```
if (na == m)
  {
   while (nb < n)
       c[nc++] = b[nb++];
  }
else if (nb == n)
  {
    while (na < m)
        c[nc++] = a[na++];
  }
}
```

Applications of LD:

Limitations of linear Arrays

(1) Prior knowledge of no. of elements in the linear array is necessary.

(2) These are static structures. Static in the sense that whether memory is allocated at compilation time or runtime, the memory used by them cannot be reduced or extended.

(3) As the elements of these arrays are stored in consecutive locations, the insertions & deletions in these array are time consuming, as moving elements down to create a space for new element or moving elements up to occupy the space vacated by the deleted element.

Application

1.) Array addition
2.) Array subtraction, multiplication
3.) Transpose of an array
4.) Addition of rows & col

Or

```
void bubble sort (int *a, int n)
{
int j, k, temp, flag = 1;
k = 1
while ((k < n) && (flag))
  {  flag = 0;
  for (j = 0, j < n - k, j++)
  {
  if (a[j] > a[j+1])
    {
      flag = 1;
    temp = a[j];
    a[j] = a[j+1];
    a[j+1] = temp;

  }
}
k++;

}
}
```

Pass 1

Step 1 if a[0] > a[1] then swap a[0] & a[1].
2 if a[1] > a[2] then " a[1] & a[2].

(n-1) if a[n-2] > a[n-1] then swap a[n-2] & a[n-1].

Pass 2

Step 1 if a[0] > a[1] then swap a[0] & a[1].
2 if a[1] > a[2] then swap a[1] & a[2].

step (n-2) iff a[n-3] > a[n-3] & a[n-2].

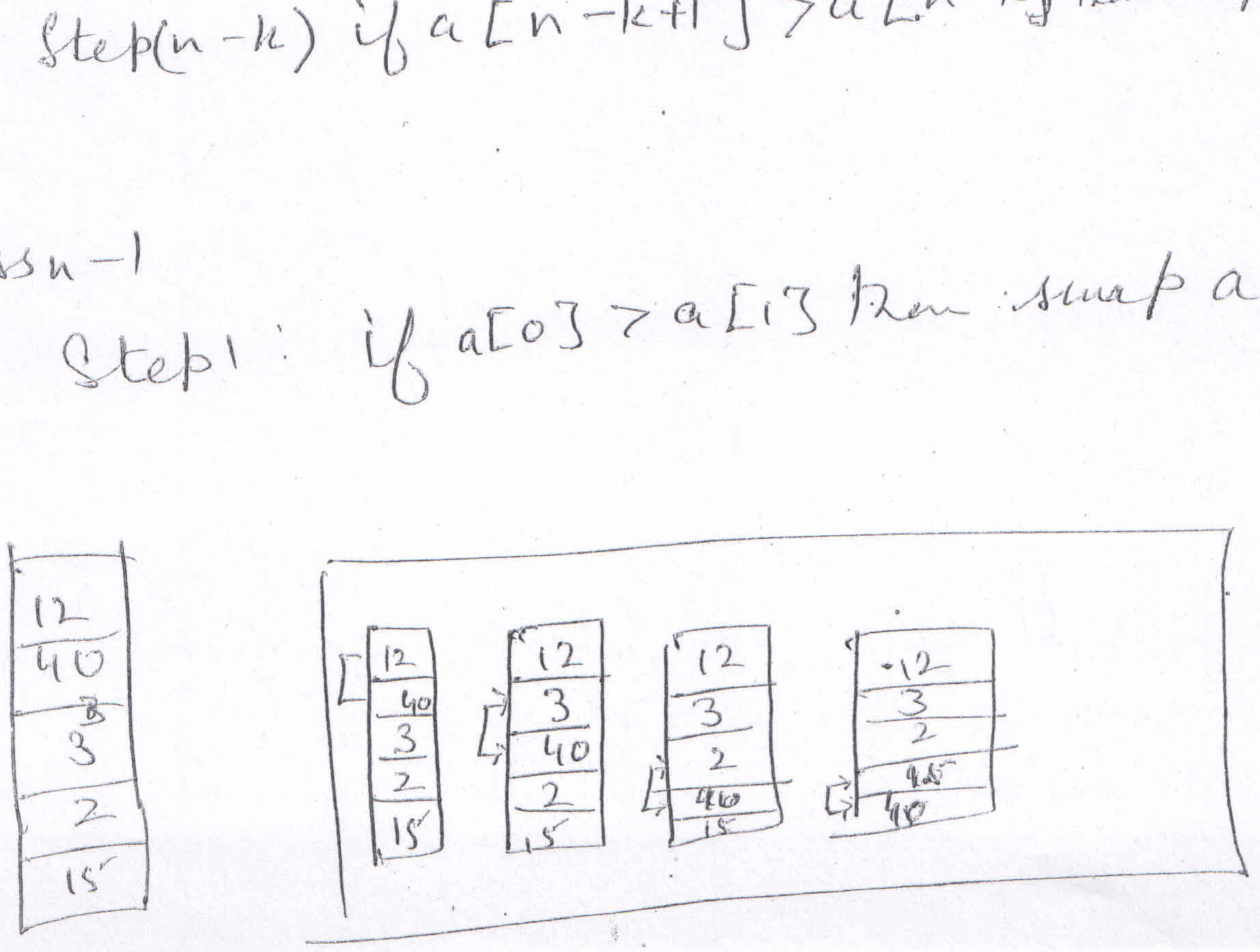
Pass k

Step 1 if a[0] > a[1] then swap a[0] & a[1]
2 if a[1] > a[2] then swap a[1] & a[2].

step(n-k) if a[n-k+1] > a[n-k] then swap

Pass n-1

Step 1 : if a[0] > a[1] then swap a[0] & a[1].



Pass 1

4 Deletion Operation.

- Deletion refers to the operation of removing an element from existing list of elements.
- After deletion the size of the LA is decreased by factor one.

(a) Deletion from a Given Position.

```
void delete (int *a, int *n, int k)
{
int j;
j = k+1;
while (j <= *n)
  {
    a[j-1] = a[j];
    j++;
  }
(n*)--;
}
```

(b) Deletion from a Sorted Array.

- First we find the position k from where the given element is to be deleted & then start from (k+1)th position i.e. the elements are moved up by one position in order to fill the hole created at kth position.

```
void delete (int a[10], int *n, int item)
{
int j, k;
k = Binary search Recursive (a, 0, *n-1, item);
  if (k == -1) {
  printf ("\n Element %d is not in the array \n", item);
    return;
```

Code.

```
int bSI (int *a, int n, int item)
{
    int beg, end, mid;
    beg=0, end =n-1
    mid= (beg + end) /2;
    while ( (beg <=end) && (a[mid]!=item))
    {
        if(element < a [mid])
        end = mid-1;
    else
        beg = mid+1;
    mid = (beg+end)/2;
}
if (beg > end);
    return -1;
else
    return mid;
}
```

(3) Insertion operation -

- Insertion refers to the operation of adding an element to existing list of elements.
- After insertion the size of the linear array is increased by factor of one
- Insertion impossible only if the memory space allocated is large enough to accomodate the additional element.
- To insert an element at any other location, the elements are to be moved downward to new ~~beg~~ location to accomodate

(a) LS.

Algo

```
Begin
   for i = 0 to (n-1) by 1 do
      if (a[i] = item) Then
         set loc = 1
         Exit
      end if
   end for
   Set loc = -1
End.
```

Code

```
int ls (int *a, int n, int item)
{
  int k;
  for (k=0; k<n; k++)
  {
    if (a[k] == item;
       return k
  }
  return -1;
}
```

(b) Binary Search - Sorted array A with 7 elements.

0	1	2	3	4	5	6
3	10	15	20	35	40	60

search 15

Step 1 begin with 0 & 6 & compute location of middle element as :— mid = (beg + end)/2
= (0+6)/2 = 3

i.e a[3] ≠ 15 & beg ≤ end,

```
for (i=0; i<5; i++)
        printf ("%d\t", a[i]);
    printf ("\n");
}
```

Operations on Linear Arrays

- Operations that can be performed on LA are!

(1) Traversal
(2) Search
(3) Insertion
(4) Deletion
(5) Sorting
(6) Muging

1, Traversal Operation

- Traversing is the process of visiting each element of the array exactly once.
- As element of LA can be accessed directly, but we have to vary an index from LB to UB in steps of 1 to access individual elements in order.

- Algo

```
Begin
    for i = 0 to (n-1) by 1 do
        Apply procen (a[i])
    end for
End.
```

- The use of symbolic constant to specify the size of array makes it convenient to modify the prog. if the size of array is to be changed later, coz the size has to be changed only at one place, in the #define directive.

1.2 Accessing 1-D Array Elements

- The elements of an array can be accessed by specifying the array name followed by subscript in brackets.
- In C, the array subscripts starts from 0. Example: If there is an array of size then the valid subscripts will be from 0 to 4.
- The last valid subscript is one less than the size of the array.
~~A is last~~
- The subscript can be any expression that yields an integer value.
- It can be any integer constant, integer variable, integer expression or return value (int) from a function call.
- Example: If 'i' & 'j' are integer variables, then these are valid subscripted array elements:-
arr[3], arr[i], arr[i+j], arr[2*j], arr[i++]

- Example: If arr & sal are two arrays of sizes 5 & 10.

```
int arr[5];
float sal[10];
int i;
scanf("%d", &arr[1]);        * I/P value into arr[1] *
printf("%f", sal[3]);        * print value of sal[3] *
arr[4] = 25;                 * assign a value to arr[4] *
arr[4]++;                    * increment the value of arr[4] by 1 *
sal[5] += 200;               * Add 200 to sal[5] *
sum = arr[0] + arr[1] + arr[2] + arr[3] + arr[4];   * Add all the values of array arr[5] *
i = 2
scanf("%f", &sal[i]);        * I/P value into sal[2] *
printf("%f", sal[i]);        * Print value of sal[2] *
printf("%f", sal[i++]);      * Print value of sal[2] & increment the value of i *
```

- Example: The array AUTO, which records the no. of automobile sold each yr. from 1932 to 1964. Suppose AUTO appears as the fig. given below, i.e Base (Auto) = 200 & w = 4 words per memory cell for Auto.

Loc (AUTO[1932]) = 200, Loc(AUTO[1933]) = 204,
Loc(AUTO[1934]) = 208 - - - -

The address of the array element for the yr K = 1965

Sol:- $LOC(AUTO[K]) = Base(LA) + w(K - lowerbound)$

Base (LA) = 200
w = 4 words per memory
lowerbound = 1932
K = 1965

$LOC(AUTO[1965]) = 200 + 4(1965 - 1932)$
Loc (AUTO[1965]) = ~~200~~ 332

2.) Parentheses notation (used in FORTRAN, PL/1 & BASIC)

A(1), A(2), A(3) ---- -, A(N)

3.) Bracket Notation (used in Pascal).

A[1], A[2], A[3], ..., A[N].

- Generally subscript notation or the bracket notation
- The no K in A[K] is called a subscript or an index & A[K] is called a subscripted variable

- Let data be a 6 element single linear array of integers D[1] = 12, D[2] = 14, D[3] = 6, D[4] = 2, D[5] = 20 D[6] = 1.

Or D: 12, 14, 6, 2, 20, 1

Data

1	12
2	14
3	6
4	2
5	20
6	1

(a)

Or

Data

12	14	6	2	20	1
1	2	3	4	5	6

(b)

- An automobile company uses an array Auto to record to no. of automobiles sold each year from 1932 to 1984.
  Auto[K] = no. of automobiles sold in the yr K.

Sol:

LB = 1932

UB = 1984

Length = 1984 − 1932 + 1

= 52 + 1

= 53

∴ AUTO array contains 53 elements & its index set consists of all integers from 1932 to 1984.

Two - Dimensional Arrays.

- 2-D array is a list of a finite no. m*n of homogeneous data elements such that

(1) Elements of the array are referenced by 2 index sets consists of m & n consecutive integer nos.

(2) Elements of the array are stored in consecutive memory location.

- size of 2-D array is denoted by mxn.

b[i,j] - in mathematical notation

b(i,j) - in BASIC & FORTRAN languages

b[i,j] - in Pascal lang.

b[i][j] - in C/C++ & Java languages.

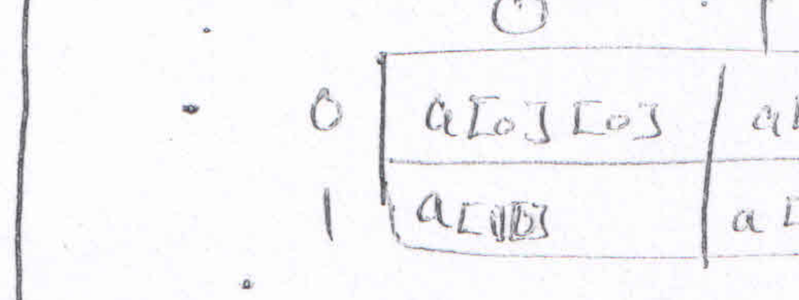
- 2-D array are called matrices in mathematics & tables in business applications.

$$\begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{bmatrix} a_{00} & a_{01} & a_{02} & a_{03} \\ a_{10} & a_{11} & a_{12} & a_{13} \\ a_{20} & a_{21} & a_{22} & a_{23} \\ a_{30} & a_{31} & a_{32} & a_{33} \end{bmatrix} \end{array}$$

	0	1
0	a[0][0]	a[0][1]
1	a[1][0]	a[1][1]

Syntax: <data type> <array name>[Rows][col];

Int A[50][50];



- Using base address, the computer computes the address of the element in the ith row & jth col i.e loc(a[i][j]), by using following formula:

Row Major Order.

Eg. loc(a[i][j]) = base(a) + w [no. of (i - lowerbound of row) + (j - lowerbound of col)]

Col. Major Order

$$loc(a[i][j]) = base(a) + w\left(\text{rows}\left(\left(i - \text{lower bound of col.}\right) + \left(j - \text{lower bound of row}\right)\right)\right]$$

- WAP to I/P & display a matrix

```
#define ROW 3
#define COL 4
#include <stdio.h>
main()
{
int i, j, a[ROW][COL];
printf("Enter the elements of the matrix (%d x %d) row-wise:\n",
                                                    ROW, COL);
for(i=0, i<Row; j++)
        scanf("%d", &a[i][j]);
printf("The Matrix that we have entered is:\n");
for(i=0, i<Row, i++)
    {
        for(j=0; j<col; j++)
            printf("%5d", a[i][j]);
        printf("\n");
    }
```