ARRAYS

INTRODUCTION

Divyashikha Sethia
Department of CSE, DTU
divyashikha@dtu.ac.in

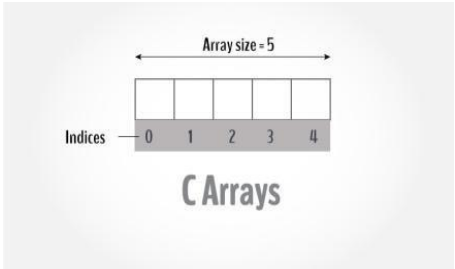
1

3.1 INTRODUCTION

•An array is a collection of similar data elements. These data elements have the same data type.

•The elements of the array are stored in consecutive memory locations and are referenced by an index (also known as the *subscript*).

•The subscript is an ordinal number which is used to identify an element of the array



2

3.2 DECLARATION OF ARRAY

- Array can be declared by specifying its type and size or by initializing it or by both
  - *Data type*—the kind of values it can store, for example, int, char, float, double.
  - *Name*—to identify the array.
  - *Size*—the maximum number of values that the array can hold.

type name[size];

- The array-size must be an integer constant and greater than zero.
- In C, the array index starts from zero.

3

```
data_type array_name[array_size];

//Array declaration by specifying size
int arr[10];

//Array declaration by initializing elements
int arr[] = {10, 20, 30, 40}

//Array declaration by specifying size and initializing elements
int arr[4] = {10, 20, 30, 40}
```

1st element	2nd element	3rd element	4th element
arr[0]	arr[1]	arr[2]	arr[3]

Memory representation of an array of 4 elements

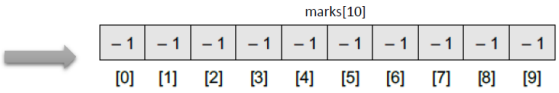
4

3.3 ACCESSING ARRAY ELEMENTS

S

- Array elements are accessed by using an integer index.
- It has 0 as the first index and not 1.
- If the size of an array is n(say), to access the last element, (n-1) index is used.
- To access all the elements, we must use a loop.
- Example:

```
// Set each element of the array to -1
int i, marks[10];
for(i=0;i<10;i++)
        marks[i] = -1;
```



- The code accesses every individual element of the array and sets its value to–1.

5

3.3.1 Address Calculation

An array stores all its data elements in consecutive memory locations, storing just the base address (address of the first element in the array)

Array[k] = BaseAddress + w(k –lower_bound) // Address of data element

Here, Array is an array, k is the index of the element of which we have to calculate the address, and w is the size of one element in memory.

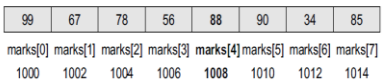
- **Example:**

Given an array int marks[] = {99,67,78,56,88,90,34,85}, calculate address of marks[4] if base address = 1000.

Solution:

We know that storing an integer value requires 2 bytes, therefore, its size is 2 bytes.

marks[4] = 1000 + 2(4 –0)



6

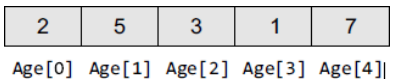
3.3.2 Length Calculation

The length of an array is given by the number of elements stored in it. The general formula to calculate the length of an array is

Length = upper_bound –lower_bound + 1 // Size of array

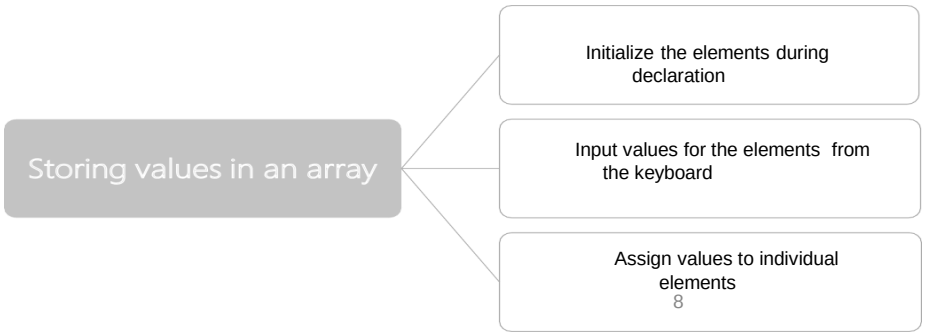
where upper_bound is the index of the last element and lower_bound is the index of the first element in the array.

- Example:
  Let Age[5] be an array of integers such that
  Age[0] = 2, Age[1] = 5, Age[2] = 3, Age[3] = 1, Age[4] = 7
  Length = upper_bound–lower_bound + 1 Here, lower_bound = 0, upper_bound = 4
  Therefore, length = 4–0 + 1 = 5



7

3.4 STORING VALUES IN ARRAY

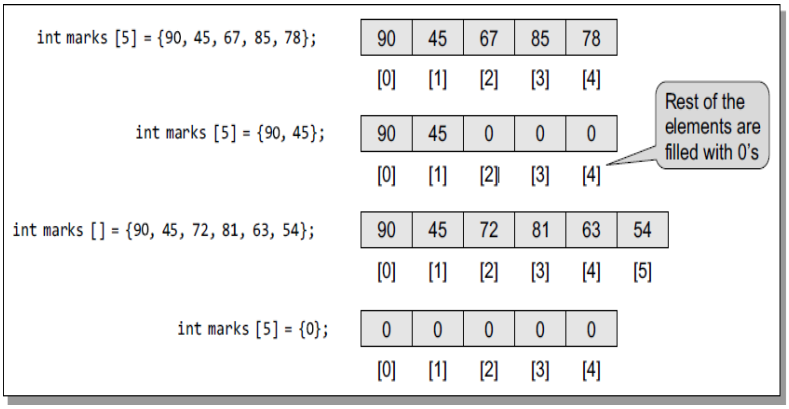


8

Initialize the elements during declaration

Arrays are initialized by writing,

type array_name[size]={list of values};



9

Input values for the elements from the keyboard

•An array can be initialized by inputting values from the keyboard.

•A while/do–while or a for loop is executed to input the value for each element of the array.

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

10

1
0

Assigning Values to Individual Elements

- Any value that evaluates to the data type as that of the array can be assigned to the individual array element.
- A simple assignment statement can be written as

  marks[3] = 100;

- We cannot assign one array to another array, even if the two arrays have the same type and size.
- To copy an array, you must copy the value of every element of the first array into the elements of the second array.

```
int i, arr1[10], arr2[10];
arr1[10] = {0,1,2,3,4,5,6,7,8,9};
for(i=0;i<10;i++)
      arr2[i] = arr1[i];
```

Code to copy an array at the individual element level

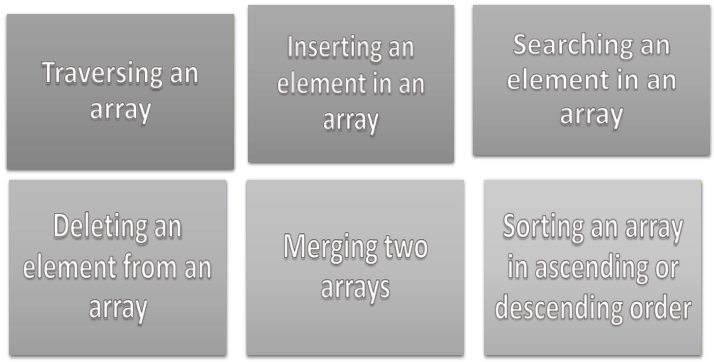
- We can also use a loop to assign a pattern of values to the array elements. For example, if we want to fill an array with even integers (starting from 0), then we will write the code as

```
// Fill an array with even numbers
int i,arr[10];
for(i=0;i<10;i++)
      arr[i] = i*2;
```

- In the code, we assign to each element a value equal to twice of its index, where the index starts from 0.
- So after executing this code, we will have

  arr[0] = 0, arr[1] = 2, arr[2] = 4, and so on.

3.5 OPERATIONS ON ARRAY



3.5.1 Traversing An Array

Traversing an array means accessing each and every element of the array for a specific purpose which can include printing every element, counting the total number of elements, or performing any process on these elements.

Pseudo-code :-

```
Step 1: (initialization) Set counter = lower_bound
Step 2: Repeat steps 3 to 4 while counter <=
upper_bound  Step 3:    Your algorithm
Step 4: Set counter++;  (end of
   loop)
Step 5: Exit
```

Example

Write a program to read and display *n* numbers using an array.

```
#include <stdio.h>  #include
<conio.h>
int main()
{
int i, n, arr[20];
clrscr();
printf("\n Enter the number of
elements in the  array : ");
scanf("%d", &n);
for(i=0;i<n;i++)
{
printf("\n arr[%d] = ", i);
scanf("%d",&arr[i]);
}
printf("\n The array elements are
");  for(i=0;i<n;i++)
printf("\t %d", arr[i]);
return 0;
}
```

Output

```
Enter the number of elements in
the array : 5
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
The array elements are 1 2 3 4 5
```

15

3.5.2 Insertion

We assume that contiguous memory space is available for allocation. Insertion can be performed in two positions, at the end or in the middle of the array.

Insertion at the end

- We just have to add 1 to the upper_ bound and assign the value.
- Assume that the memory space allocated for the array is still available.
- For example, if an array is declared to contain 10 elements, but currently it has only 8 elements, then obviously there is space to accommodate two more elements.
- But if it already has 10 elements, then we will not be able to add another element to it.

Algorithm :-

```
Insertion at the end
Step 1: Set upper_bound =
upper_bound + 1  Step 2:
Set A[upper_bound] =
value;
Step 3: Exit
```

16

Insertion in the middle

•We first find the location where the new element will be inserted

•Then move all the elements (that have a value greater than that of the new element) one position to the right so that space can be created to store the new value.

Algorithm :-

```
Insertion in the middle
Step 1: (initialization) Set counter = upper_bound
Step 2: Repeat steps 3 to 4 while counter >= position
Step 3:     Set A[counter +1] = A[counter]
Step 4:     Set counter--;  (end of loop)
Step 5: Set upper_bound = upper_bound + 1
Step 6: Set A[position] = value;
Step 7: Exit
```

Example :

Data[] is an array that is declared as int Data[20]; and contains the following values:

Data[] = {12, 23, 34, 45, 56, 67, 78, 89, 90, 100};

(a)Calculate the length of the array.

(b)Find the upper_bound and lower_bound.

(c)Show the memory representation of the array.

(d)If a new data element with the value 75 has to be inserted, find its position.

(e)Insert a new data element 75 and show the memory representation after the insertion.

Solution

(a)Length of the array = number of elements. Therefore, length of the array = 10

(b)By default, lower_bound = 0 and upper_bound = 9

(c)

12	23	34	45	56	67	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]

(d) Since the elements of the array are stored in ascending order, the new data element will be stored after 67, i.e., at the 6th location. So, all the array elements from the 6thposition will be moved one position towards the right to accommodate the new value.

(e)

12	23	34	45	56	67	75	78	89	90	100
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]	Data[7]	Data[8]	Data[9]	Data[10]

3.5.3 Deletion

Removing data element from an already existing array. Deletion also can be performed in two positions, at the end or in the middle of the array

Deletion at the end

•We just have to subtract 1 to the upper_ bound and assign the value.

Algorithm :-

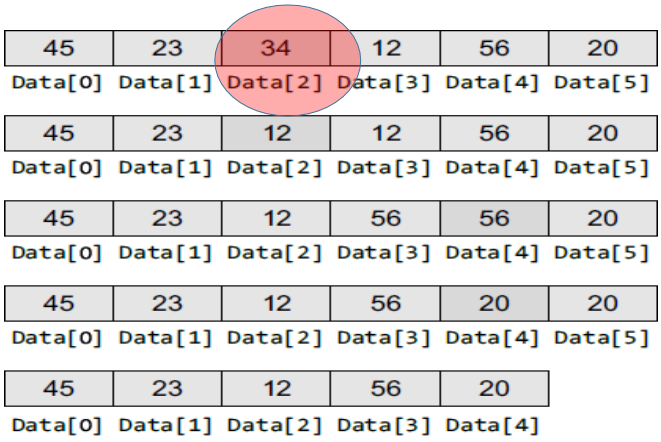
```
Deletion at the end
Step 1: Set upper_bound = upper_bound - 1
Step 2: Exit
```

Deletion in the middle

- We must first find the location from where the element has to be deleted
- Then move all the elements (having a value greater than that of the element) one position towards left so that the space vacated by the deleted element can be occupied by rest of the elements.

Algorithm :-

```
Deletion in the middle
Step 1: (initialization) Set counter = position
Step 2: Repeat steps 3 to 4 while counter <= upper_bound
Step 3:     Set A[counter] = A[counter + 1]
Step 4:     Set counter++;
      (end of loop)
Step 5: Set upper_bound = upper_bound - 1
Step 6: Exit
```

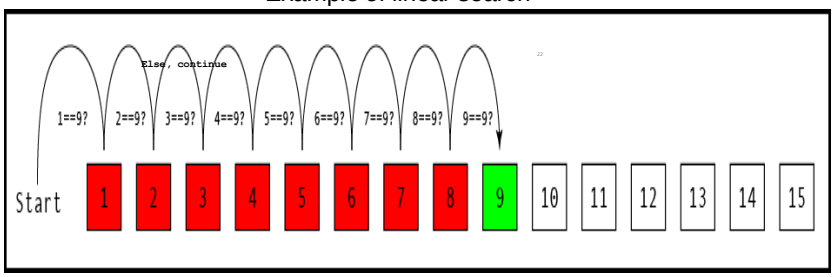


Deleting elements from an array

Searching In An Array

Linear search: It works by traversing all the elements one by one in an orderly manner, until the requested item is found.

Example of linear search



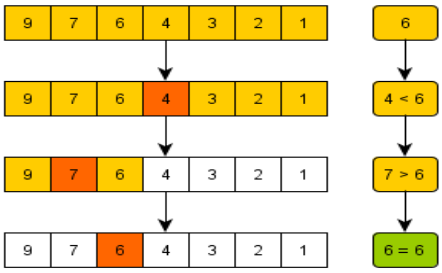
Pseudo-code

```
Linear search Algorithm

For all elements
Check if it is equal to the element being searched for
If it is, return the position
Else, continue
```

22

Binary search: It works by repeatedly dividing in half the portion of the list that could contain the item, until one narrows down the possible locations to just one.



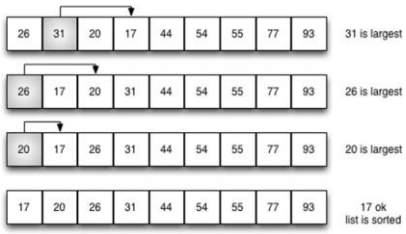

```
Binary Search Algorithm

Step 1: Let min = 1 and max = n.
Step 2: Guess the average of max and min rounded down so that it
is an integer.
Step 3: If you guessed the number, stop. You found it!
Step 4: If the guess was too low, set min to be one larger than the guess.
Step 5: If the guess was too high, set max to be one smaller than the guess.
Step 6: Go back to step two.
```

Sorting

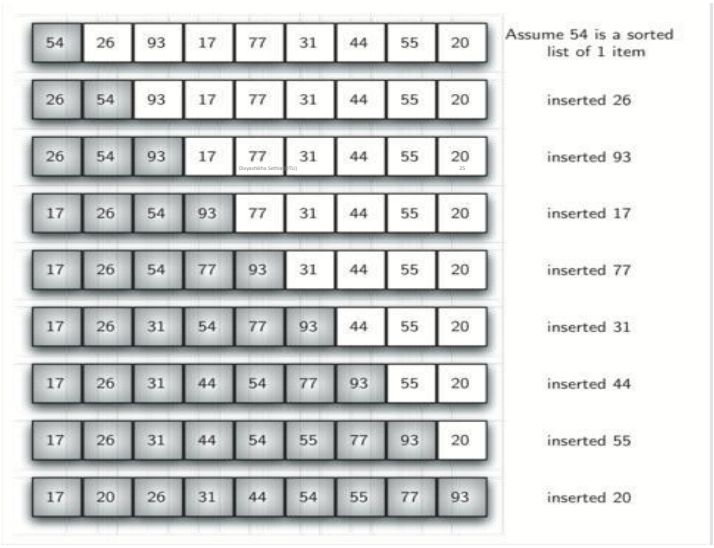
Sorting a list of items into ascending or descending order can help find items on that list quickly. There are many different ways to sort data, the simplest one is selection sort.

Selection Sort : it repeatedly*selects*the next-smallest element and swaps it into place.




```
Step 1: Find the smallest card. Swap it with the first card.
Step 2: Find the second-smallest card. Swap it with the second card.
Step 3: Find the third-smallest card. Swap it with the third card.
Step 4: Repeat finding the next-smallest card, and swapping it into
the correct position until the array is sorted.
```

Insertion Sort : Loop over positions in the array, starting with index 1. Each new position is like the new card handed to you by the dealer, and you need to insert it into the correct place in the sorted subarray to the left of that position.



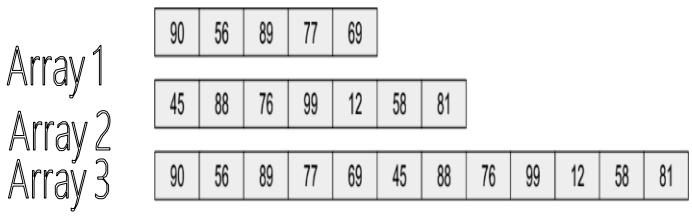
3.5.4 Merging Arrays

Merging two arrays in a third array means copying the contents of the first array and second array into the third array.

- Merging sorted arrays
- Merging unsorted arrays

Merging unsorted arrays

If the arrays are unsorted, then merging the arrays is very simple, as one just needs to copy the contents of one array into another array.



26

Example

Write a program to merge two unsorted arrays.

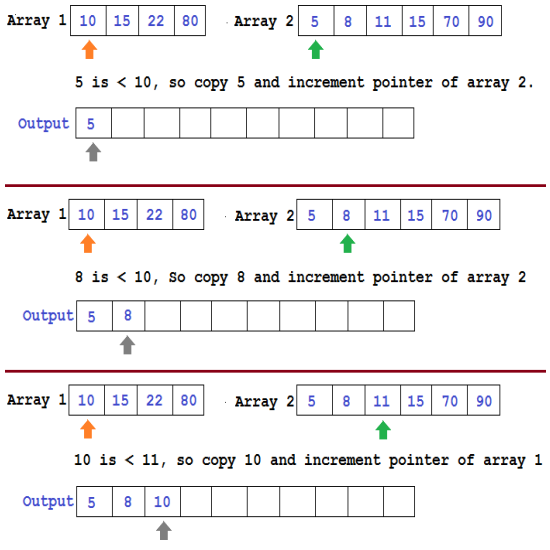
```
#include <stdio.h>
#include <conio.h>
int main() {
int arr1[10], arr2[10], arr3[20];
int i, n1, n2, m, index=0;
clrscr();
printf("\n Enter the number of elements in array1 : ");
scanf("%d", &n1);
printf("\n\n Enter the elements of the first array");
for(i=0;i<n1;i++) {
        printf("\n arr1[%d] = ", i);
        scanf("%d", &arr1[i]); }
printf("\n Enter the number of elements in array2 : ");
scanf("%d", &n2);
printf("\n\n Enter the elements of the second array");
for(i=0;i<n2;i++) {
        printf("\n arr2[%d] = ", i);
        scanf("%d", &arr2[i]); }
m = n1+n2;
for(i=0;i<n1;i++) {
        arr3[index] = arr1[i];
        index++; }
for(i=0;i<n2;i++) {
        arr3[index] = arr2[i];
        index++; }
printf("\n\n The merged array is");
for(i=0;i<m;i++)
printf("\n arr[%d] = %d", i, arr3[i]);
getch();
return 0;
}
```

Output

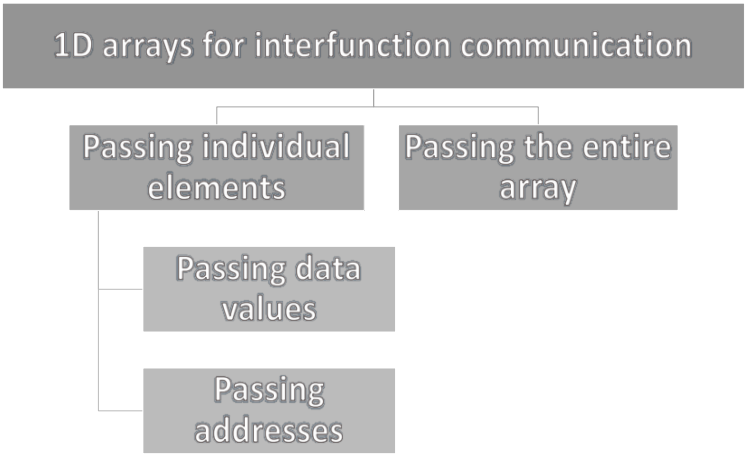
```
Enter the number of elements in array1 : 3
Enter the elements of the first array
arr1[0] = 1
arr1[1] = 2
arr1[2] = 3
Enter the number of elements in array2 : 3
Enter the elements of the second array
arr2[0] = 4
arr2[1] = 5
arr2[2] = 6
The merged array is
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 4
arr[4] = 5
arr[5] = 6
```

Merging sorted arrays

- We first compare the 1st element of array1 with the 1st element of array2, and then put the smaller element in the merged array.
- Compare the 2nd element of the second array with the 1st element of the first array and and then put the smaller element in the merged array.
- Now, we will compare the 1st element of the first array with the 3rd element of the second array.
- This procedure will be repeated until elements of both the arrays are placed in the right location in the merged array.



3.6 PASSING ARRAY TO FUNTIONS



3.6.1 Passing individual element

Passing Data Values

•Individual elements can be passed in the same manner as we pass variables of any other data type.

•The condition is just that the data type of the array element must match with the type of the function parameter.

Calling function	Called function

main()
{
int arr[5] ={1, 2, 3, 4, 5};
func(arr[3]);
}
	void func(int num)
	{
	printf("%d", num);
	}

•In the above example, only one element of the array is passed to the called function. This is done by using the index expression. Here, arr[3] evaluates to a single integer value.

•The called function hardly bothers whether a normal integer variable is passed to it or an array value is passed.

30

Passing individual element

Passing Addresses

•Like ordinary variables, we can pass the address of an individual array element by preceding the indexed array element with the address operator.

•Therefore, to pass the address of the fourth element of the array to the called function, we will write &arr[3].

•However, in the called function, the value of the array element must be accessed using the indirection (*) operator.

Calling function

```
main()
{
      int arr[5] ={1, 2, 3, 4, 5};
      func(&arr[3]);
}
```

Called function

```
void func(int *num)
{
      printf("%d", *num);
}
```

3.6.2 Passing the entire array

•In C the array name refers to the first byte of the array in the memory.

•The address of the remaining elements in the array can be calculated using the array name and the index value of the element.

•Therefore, when we need to pass an entire array to a function, we can simply pass the name of the array.

•A function that accepts an array can declare the formal parameter in either of the two following ways

```
func(int arr[]); or func(int *arr);
```

•You can also pass the size of the array as another parameter to the function.So for a function that accepts an array as parameter, the declaration should be as follows.

```
func(int arr[], int n); or func(int *arr, int n);
```

•When we pass the name of an array to a function, the address of the zeroth element of the array is copied to the local pointer variable in the function. When a formal parameter is declared in a function header as an array, it is interpreted as a pointer to a variable and not as an array.

•With this pointer variable you can access all the elements of the array by using the expression:

```
array_name + index.
```

•We can also pass a part of the array known as a sub-array. A pointer to a sub-array is also an array pointer. For example, if we want to send the array starting from the third element then we can pass the address of the third element and the size of the sub-array, i.e., if there are 10 elements in the array, and we want to pass the array starting from the third element, then only eight elements would be part of the sub-array. So the function call can be written as:

```
func(&arr[2], 8);
```

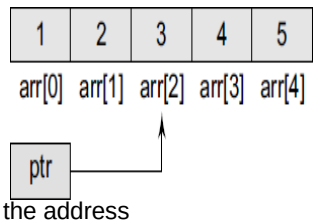
•In case we want the called function to make no changes to the array, the array must be received as a constant array by the called function. This prevents any type of unintentional modifications of the array elements. To declare an array as a constant array, simply add the keyword*c onst* before the data type of the array.

3.7 POINTERS AND ARRAYS

•Array notation is a form of pointer notation.

•The name of the array is the starting address of the array in memory. It is also known as the base address.

```
int *ptr;
ptr = &arr[0];
```



•Here, ptr is made to point to the first element

```
ptr = &arr[2]
```

•It makes ptr to point to the third element of the array that has index 2.

•If pointer variable ptr holds the address of the first element in the array, then the address of successive elements can be calculated by writing *ptr*++.

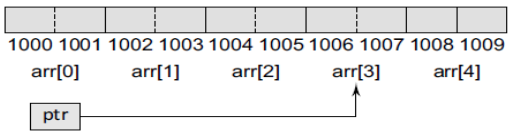
3.8 POINTERS AND ARRAYS

```
int *ptr = &arr[0];
ptr++;
printf("\n The value of the second element of the array is %d",
*ptr);
```

•The **printf()** function will print the value 2 because after being incremented **ptr** points to the next location. One point to note here is that if x is an integer variable, then **x++; adds 1 to the value of x**. But **ptr** is a pointer variable, so when we write **ptr+i**, then adding i gives a pointer that points i elements further along an array than the original pointer.

•Since **++ptr and ptr++** are both equivalent to **ptr+1,** incrementing a pointer using the unary ++ operator, increments the address it stores by the amount given by sizeof(type) where type is the data type of the variable it points to (i.e., 2 for an integer).

•If ptr originally points to arr[2], then ptr++ will make it to point to the next element, i.e., arr[3].



Pointer (ptr) pointing to the fourth element of the array

•When using pointers, an expression like arr[i] is equivalent to writing *(arr+i).

•For example, while we can write

```
ptr = arr; // ptr = &arr[0]
```

we cannot write `arr = ptr;`

•This is because while ptr is a variable, arr is a constant. the location at which the first element of arr will be stored cannot be changed once arr[] has been declared. Therefore, an array name is often known to be a constant pointer.

•To summarize, the name of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the element that it points to. Therefore, arrays and pointers use the same concept.

•We can add or subtract an integer from a pointer to get a new pointer, pointing somewhere other than the original position. C also permits addition and subtraction of two pointer variables.

•Example:

```
int main()
{
int arr[]={1,2,3,4,5,6,7,8,9};
int *ptr1, *ptr2;  ptr1 = arr;  ptr2 = arr+2;
printf("%d", ptr2—ptr1);  return 0;
}
```

•Output: 2

ARRAYS OF POINTERS

•An array of pointers can be declared as

```
int *ptr[10];
```

•The above statement declares an array of 10 pointers where each of the pointer points to an integer variable.

Example:

```
int *ptr[10];
int p = 1, q = 2, r =
3, s = 4, t = 5;
ptr[0] = &p;
ptr[1]    =    &q;
ptr[2]    =    &r;
ptr[3]    =    &s;
ptr[4] = &t;
printf("\n %d", *ptr[3]);
```

•The output will be 4 because ptr[3] stores the address of integer variable s and *ptr[3] will therefore print the value of s that is 4.

•Another code in which we store the address of three individual arrays in the array of pointers:

```
Example:
int main() {
int arr1[]={1,2,3,4,5};
int arr2[]={0,2,4,6,8};
int arr3[]={1,3,5,7,9};
int *parr[3] =
{arr1, arr2,
arr3};  int i;
for(i = 0;i<3;i++)
 printf(«%d», *parr[i]);
return 0; }
Output
1 0 1
```

•In the for loop, parr[0] stores the base address of arr1 (or, &arr1[0]). So writing *parr[0] will print the value stored at &arr1[0]. Same is the case with *parr[1] and *parr[2].

3.9 2DIMENSIONAL ARRAY

•The simplest form of multidimensional array is the two-dimensional array.

•A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column

•In simple terms, an array of one-dimensional arrays is a two-dimensional array.

```
data_type array_name[row_size][column_size]; //declaration of 2-D array
int A[3][4]; // example
```

•A 2D array is treated as a collection of 1D arrays. Each row of a 2D array corresponds to a 1D array consisting of n elements, where n is the number of columns.

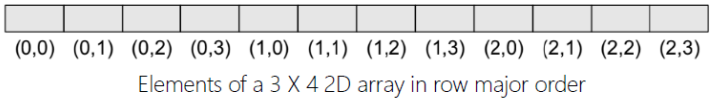
		Column 1	Column 2	Column 3	Column 4
First dimension	Row 1	A[0][0]	A[0][1]	A[0][2]	A[0][3]
	Row 2	A[1][0]	A[1][1]	A[1][2]	A[1][3]
	Row 3	A[2][0]	A[2][1]	A[2][2]	A[2][3]

Second dimension

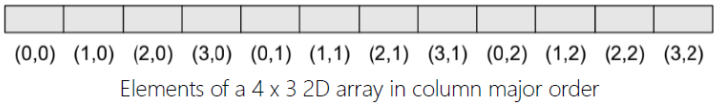
•A two-dimensional m x n array is an array that contains mxn data elements and each element is accessed using two subscripts, i and j, where i <= m and j <= n.

•A rectangular picture of a two-dimensional array, in the memory, will be stored sequentially.

•There are two ways of storing a two-dimensional array in the memory. The first way is the*row major order*and the second is the c*olumn major order.*



Elements of a 3 X 4 2D array in row major order



Elements of a 4 x 3 2D array in column major order

•2D array that are stored in a row major order: elements of first row stored before elements of second and third rows. (array stored row by row)

•column major order: elements of first column are stored before elements of second and third column. (array stored column by column)

Address Calculation

If the array elements are stored in column major order,

```
Address(A[i][j]) = Base_Address + w { M(j-1) + (i-1)}
```

If the array elements are stored in row major order,

```
Address(A[i][j]) = Base_Address + w { N(i-1) + (j-1)}
```

where w is the number of bytes required to store one element, N is the number of columns, M is the number of rows, and I and J are the subscripts of the array element.

Example:

Consider a 20x5 two-dimensional array marks which has its base address = 1000 and the size of an element = 2. Now compute the address of the element, marks[18][4] assuming that the elements are stored in row major order.

Solution

```
Address(A[I][J]) = Base_Address + w{N (I— 1) + (J— 1)}
Address(marks[18][4]) = 1000 + 2 {5(18— 1) + (4— 1)}
        = 1000 + 2 {5(17) + 3}
        = 1000 + 2 (88)
        = 1000 + 176 = 1176
```

3.9.2 INITIALIZING 2-D ARRAY

- A two-dimensional array is initialized in the same way as a one-dimensional array is initialized. For example

```
int marks[2][3] = {90, 87, 78, 68, 62, 71};
```

- Since, the initialization of a two-dimensional array is done row by row. The above statement can also be written

```
int marks[2][3] = {{90, 87, 78}, {68, 62, 71}};
```

- In the above example, each row is defined as a one-dimensional array of three elements that are enclosed in braces.

```
int marks[][3] = {{90, 87, 78}, {68, 62, 71}};
```

- If the array is completely initialized, we may omit the size of the array, except that only the size of the first dimension can be omitted. Therefore, the declaration statement given below is valid

```
int marks[2][3] = {0};
```

- In order to initialize the entire two-dimensional array to zeros, simply specify the first value as zero. That is,

```
marks[1][2] = 79;  or    marks[1][2] = marks[1][1] + 10;
```

- The individual elements of a two-dimensional array can be initialized using the assignment operator as shown here.

3.9.3 ACCESSING ELEMENTS OF 2D ARRAY

The elements of a 2D array are stored in contiguous memory locations. Since the two-dimensional array contains two subscripts, we will use two for loops to scan the elements. The first for loop will scan each row in the 2D array and the second for loop will scan individual columns for every row in the array.

```
Example:
int arr[2][2] = {12, 34, 56,32};
int i, j;  for(i=0;i<2;i++) {
 printf("\n");

 for(j=0;j<2;j++)  printf("%d\t",
  arr[i][j]);
}

Output:  12 34
56 32
```

3.10 OPERATIONS ON 2D ARRAYS

Two-dimensional arrays can be used to implement the mathematical concept of *matrices*.

Transpose : Transpose of an m x n matrix A is given as a n x m matrix B, where $B_{i,j} = A_{j,i}$.

Sum : Two matrices that are compatible with each other can be added together, storing the result in the third matrix. The elements of two matrices can be added by writing:

$C_{ij} = A_{ij} + B_{ij}$

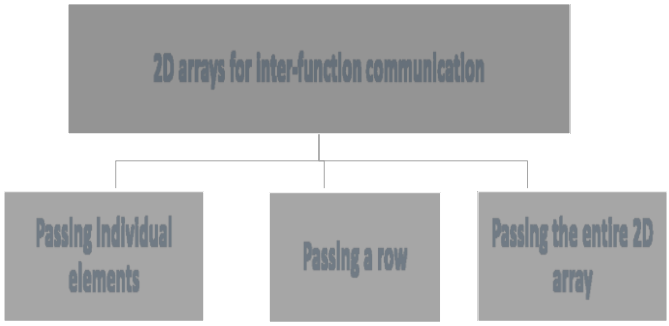
Difference : Two matrices that are compatible with each other can be subtracted, storing the result in the third matrix. The elements of two matrices can be subtracted by writing:

$C_{ij} = A_{ij} - B_{ij}$

Product : The dimension of the product matrix is m x q. The elements of two matrices can be multiplied by writing:

C_{ij} = Summation (A_{ik} B_{kj}) for k 1 to n

3.11 PASSING TWO DIMENSIONAL ARRAY TO FUNCTION



Passing A Row

•A row of a two-dimensional array can be passed by indexing the array name with the row number.

Calling function

```
main()
{
      int arr[2][3] = {{1, 2, 3}, {4, 5, 6}};
      func(arr[1]);
}
```

Called function

```
void func(int arr[])
{
      int i;
      for(i=0;i<3;i++)
        printf("%d", arr[i] * 10);
}
```

Passing the Entire 2D Array

- To pass a two-dimensional array to a function, we use the array name as the actual parameter (the way we did in case of a 1D array).
- However, the parameter in the called function must indicate that the array has two dimensions

49

Write a program to fill a square matrix with value zero on the diagonals, 1 on the upper right triangle, and −1 on the lower left triangle.

```
#include <stdio.h>
#include <conio.h>
void read_matrix(int mat[5][5], int);
void display_matrix(int mat[5][5], int);
int main()
{
        int row;
        int mat1[5][5];
        clrscr();
        printf("\n Enter the number of rows and columns of the matrix:");
        scanf("%d", &row);
        read_matrix(mat1, row);
        display_matrix(mat1, row);
        getch();
        return 0;
}

void read_matrix(int mat[5][5], int r)
{
        int i, j;
        for(i=0; i<r; i++)
        {
              for(j=0; j<r; j++)
              {
                      if(i==j)
                            mat[i][j] = 0;
                      else if(i>j)
                            mat[i][j] = -1;
                      else
                            mat[i][j] = 1;
              }
        }
}
void display_matrix(int mat[5][5], int r)
{
```


50

```
            for(i=0; i<r; i++)
            {
            printf("\n");
            for(j=0; j<r; j++)
                    printf("\t %d", mat[i][j]);
            }
    }
```

Output

```
Enter the number of rows and columns of the matrix: 2
 0              1
-1              0
```

3.12 POINTERS AND TWODIMENSIONAL ARRAYS

•Consider a two-dimensional array declared as

**ptr

int mat[5][5];

•To declare a pointer to a two-dimensional array, you may write

int **ptr

•Here int ** ptr is an array of pointers (to one-dimensional arrays), while int mat[5][5] is a 2D array.

•They are not the same type and are not interchangeable.

•Individual elements of the array mat can be accessed using either:

mat[i][j] or

((mat + i) + j) or

*(mat[i]+j);

52

- To understand more fully the concept of pointers, let us replace

  *(multi + row) with X so the expression

  ((mat + i) + j) becomes *(X + col)

- Using pointer arithmetic, we know that the address pointed to by (i.e., value of) X + col + 1 must be greater than the address X + col by an amount equal to sizeof(int).

- Since *mat* is a two-dimensional array, we know that in the expression multi + row as used above, multi + row + 1 must increase in value by an amount equal to that needed to point to the next row, which in this case would be an amount equal to COLS * sizeof(int).

- In case of a two-dimensional array, in order to evaluate expression (for a row major 2D array), we must know a total of 4 values:
  1. The address of the first element of the array, which is given by the name of the array, i.e., mat in our case.
  2. The size of the type of the elements of the array, i.e., size of integers in our case.
  3. The specific index value for the row.
  4. The specific index value for the column.

- int (*ptr)[10];

  declares ptr to be a pointer to an array of 10 integers. This is different from

- int *ptr[10];

  which would make ptr the name of an array of 10 pointers to type int

Pointer Arithmetic

```
int * arr[10] ;
int ** ptr = arr ;
```

- In this case, arr has type int **. Since all pointers have the same size, the address of ptr + i can be calculated as:

```
addr(ptr + i) = addr(ptr) + [sizeof(int *) * i]
              = addr(ptr) + [2 * i]
```

- Since arr has type int **,

arr[0] = &arr[0][0], arr[1] = &arr[1][0], and in general, arr[i] = &arr[i][0].

- According to pointer arithmetic, arr + i = & arr[i], yet this skips an entire row of 5 elements, i.e., it skips complete 10 bytes (5 elements each of 2 bytes size). Therefore, if arr is address 1000, then arr + 2 is address 1010. To summarize, &arr[0][0], arr[0], arr, and &arr[0] point to the base address.

```
&arr[0][0] + 1 points to arr[0][1]
arr[0] + 1 points to arr[0][1]
arr + 1 points to arr[1][0]
&arr[0] + 1 points to arr[1][0]
```

- A two-dimensional array is not the same as an array of pointers to 1D arrays. Actually a two-dimensional array is declared as:

```
int (*ptr)[10] ;
```

- Here ptr is a pointer to an array of 10 elements. The parentheses are not optional. In the absence of these parentheses, ptr becomes an array of 10 pointers, not a pointer to an array of 10 ints.

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

Output

```
1  2  3  4
```

- The golden rule to access an element of a two-dimensional array can be given as

$$arr[i][j] = (*(arr+i))[j] = *((*arr+i))+j) = *(arr[i]+j)$$

- Therefore,

$$arr[0][0] = *(arr)[0] = *((*arr)+0) = *(arr[0]+0)$$
$$arr[1][2] = (*(arr+1))[2] = *((*(arr+1))+2) = *(arr[1]+2)$$

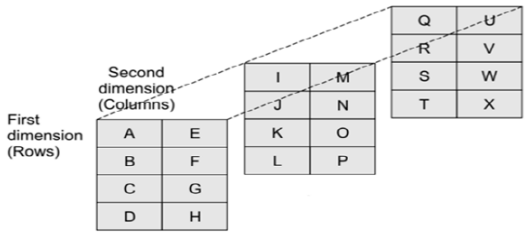
If we declare an array of pointers using, `data_type *array_name[SIZE];` Here `SIZE` represents the number of rows and the space for columns that can be dynamically allocated.	If we declare a pointer to an array using, `data_type (*array_name)[SIZE];` Here `SIZE` represents the number of columns and the space for rows that may be dynamically allocated (refer Appendix A to see how memory is dynamically allocated).

3.13 MULTI-DIMENSIONAL ARRAY

A multi-dimensional array in simple terms is an array of arrays. We have n indices in an n-dimensional array or multi-dimensional array.

An n-dimensional $m_1 \times m_2 \times m_3 \times \ldots \times m_n$ array is a collection of $m_1 \times m_2 \times m_3 \times \ldots \times m_n$ elements. In a multi-dimensional array, a particular element is specified by using n subscripts as $A[I_1][I_2][I_3]...[I_n]$, where $I_1 <= M_1$, $I_2 <= M_2$, $I_3 <= M_3$, $...I_n <= M_n$.

Following is an example of 3-D array.



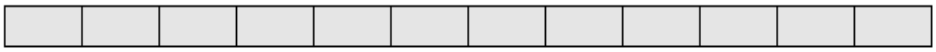
Example 3.6 Consider a three-dimensional array defined as `int A[2][2][3]`. Calculate the number of elements in the array. Also, show the memory representation of the array in the row major order and the column major order.

Solution

A three-dimensional array consists of pages. Each page, in turn, contains `m` rows and `n` columns.

(0,0,0) (0,0,1) (0,0,2) (0,1,0) (0,1,1) (0,1,2) (1,0,0) (1,0,1) (1,0,2) (1,1,0) (1,1,1) (1,1,2)

(a) Row major order



(0,0,0) (0,1,0) (0,0,1) (0,1,1) (0,0,2) (0,1,2) (1,0,0) (1,1,0) (1,0,1) (1,1,1) (1,0,2) (1,1,2)

(b) Column major order

The three-dimensional array will contain $2 \times 2 \times 3 = 12$ elements.

3.14 POINTER AND THREE DIMENSIONAL ARRAYS

- A pointer to a three-dimensional array can be declared as,

```
int arr[2][2][2]={1,2,3,4,5,6,7,8};
int (*parr)[2][2];
parr = arr;
```

- We can access an element of a three-dimensional array by writing,

```
arr[i][j][k] = *(*(*(arr+i)+j)+k)
```

3.15 SPARSE MATRICES

•Sparse matrix is a matrix that has large number of elements with a zero value.

•In order to efficiently utilize the memory, specialized algorithms and data structures that take advantage of the sparse structure should be used.

•Sparse data can be easily compressed, which in turn can significantly reduce memory usage.

Types of Sparse matrices

•There are t wo types of sparse matrices.

•In the first type of sparse matrix, all elements above the main diagonal have a zero value.

•This type of sparse matrix is also called a (*lower*) *triagonal matrix*

•In a lower triangular matrix, Ai,j = 0 where i < j.

•An nxn lower-triangular matrix A has one non- zero element in the first row, two non-zero elements in the second row and likewise n non- zero elements in the*n*th row.

(lower)

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

Lower-triangular matrix

- To store a lower-triangular matrix efficiently in the memory, we can use a one-dimensional array which stores only non-zero elements.
- The mapping between a two-dimensional matrix and a one- dimensional array can be done in any one of the following ways:

(a) Row-wise mapping—Here the contents of array A[] will be

{1, 5, 3, 2, 7,–1, 3, 1, 4, 2,–9, 2,–8, 1, 7}

(b) Column-wise mapping—Here the contents of array A[] will be

{1, 5, 2, 3,–9, 3, 7, 1, 2,–1, 4,–8, 2, 1, 7}

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

Lower-triangular matrix

- In an *upper-triangular matrix*, $A_{i,j} = 0$ where $i > j$.
- An nxn upper-triangular matrix A has n non-zero elements in the first row, n–1 non-zero elements in the second row and likewise one non-zero element in the *n*th row.

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

Upper-triangular matrix

•A *tri-diagonal matrix*, is a variant of sparse matrix.

•In a tridiagonal matrix, if elements are present on

(a)the main diagonal, it contains non-zero elements for i=j. In all, there will be n elements.

(b)below the main diagonal, it contains non-zero elements for i=j+1. In all, there will be n–1 elements.

(c)above the main diagonal, it contains non-zero elements for i=j–1. In all, there will be n–1 elements.

•To store a tri-diagonal matrix efficiently in the memory, we can use a one-dimensional array that stores only non-zero elements.

•The mapping between two-dimensional matrix and a one-dimensional array can be done in any one of the following ways:

(a)Row-wise mapping—Here the contents of array A[] will be

{4, 1, 5, 1, 2, 9, 3, 1, 4, 2, 2, 5, 1, 9, 8, 7}

(b)Column-wise mapping—Here the contents of array A[] will be

{4, 5, 1, 1, 9, 2, 3, 4, 1, 2, 5, 2, 1, 8, 9, 7}

(c)Diagonal-wise mapping—Here the contents of array A[] will be

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

Tri-diagonal matrix

3.16 APPLICATION OF ARRAYS

•Arrays are widely used to implement mathematical vectors, matrices, and other kinds of rectangular tables.

•Many databases include one-dimensional arrays whose elements are records.

•Arrays are also used to implement other data structures such as strings, stacks, queues, heaps, and hash tables.

•Arrays can be used for sorting elements in ascending or descending order.

THANKS

67