Data Structures Using C, 2e

Reema Thareja

Chapter 3

Arrays

Introduction

- An array is a collection of similar data elements.

- These data elements have the same data type.

- Elements of arrays are stored in consecutive memory locations and are referenced by an index (also known as the subscript).

- Declaring an array means specifying three things:

  Data type - what kind of values it can store. For example, int, char, float

  Name - to identify the array

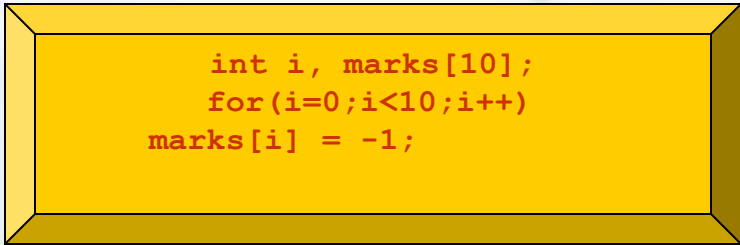 *Size* - the maximum number of values that the array can hold

- Array

1st element	2nd element	3rd element	4th element	5th element	6th element	7th element	8th element	9th element	10th element
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]	marks[8]	marks[9]

type name[size];

Accessing Elements of an Array

- To access all the elements of an array, we must use a loop.

- That is, we can access all the elements of an array by varying the value of the subscript into the array.

- But note that the subscript must be an integral value or an expression that evaluates to an integral value.

```
int i, marks[10];
for(i=0;i<10;i++)
marks[i] = -1;
```
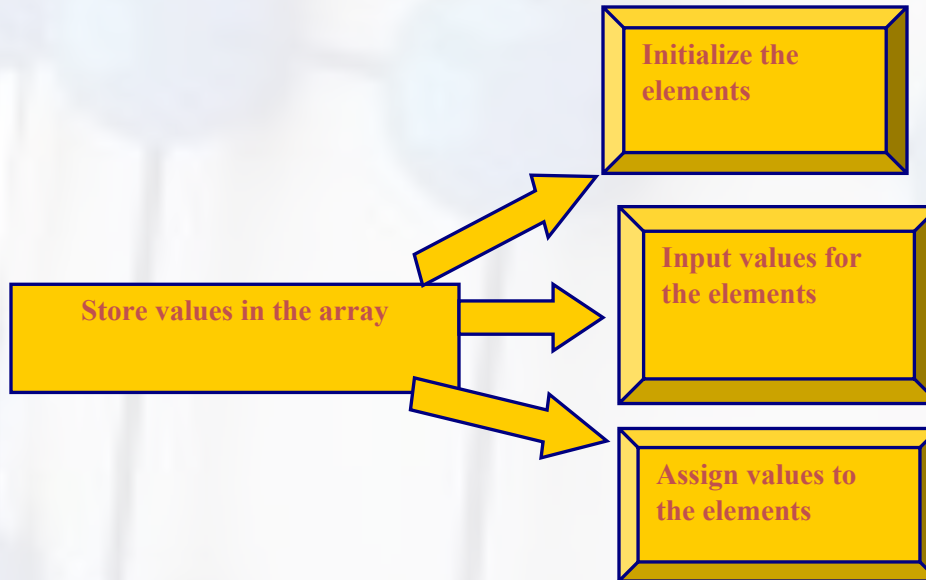
Calculating the Address of Array Elements

- Address of data element, A[k] = BA(A) + w(k – lower_bound)

  where

  A is the array

  k is the index of the element whose address we have to calculate

  BA is the base address of the array A

  w is the word size of one element in memory. For example, size of int is 2

99	67	78	56	88	90	34	85

marks[0]   marks[1]   marks[2]   marks[3]   **marks[4]**   marks[5]   marks[6]   marks[7]

1000   1002 1004 1006 **1008**   1010   1012   1014

$$marks[4] = 1000 + 2(4 – 0)$$
$$= 1000 + 2(4) = 1008$$

Storing Values in Arrays

Initialize the elements

Store values in the array

Input values for the elements

Assign values to the elements

Initializing Arrays during declaration

int marks [5] = {90, 98, 78, 56, 23};

Inputting Values from Keyboard

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

Assigning Values to Individual Elements

```
int i, arr1[10], arr2[10];
for(i=0;i<10;i++)
    arr2[i] = arr1[i];
```

Calculating the Length of an Array

Length = upper_bound – lower_bound + 1

where

upper_bound is the index of the last element

lower_bound is the index of the first element in the array

99	67	78	56	88	90	34	85
marks[0]	marks[1]	marks[2]	marks[3]	marks[4]	marks[5]	marks[6]	marks[7]]

Here, lower_bound = 0, upper_bound = 7

Therefore, length = 7 – 0 + 1 = 8

WAP to Read and Display *N* Numbers using an Array

```c
#include<stdio.h>
#include<conio.h>
int main()
{
    int i=0, n, arr[20];
    clrscr();
    printf("\n Enter the number of elements : ");
    scanf("%d", &n);
    printf("\n Enter the elements : ");
```

WAP to Read and Display *N* Numbers using an Array

```c
for(i=0;i<n;i++)
{
    printf("\n arr[%d] = ", i);
    scanf("%d", &num[i]);
}
printf("\n The array elements are ");
for(i=0;i<n;i++)
    printf("arr[%d] = %d\t", i, arr[i]);
return 0;
}
```

Inserting an Element in an Array

Algorithm to insert a new element to the end of an array

```
Step 1: Set upper_bound = upper_bound + 1
Step 2: Set A[upper_bound] = VAL
Step 3; EXIT
```

Algorithm INSERT(A, N, POS, VAL) to insert an element VAL at position POS

```
Step 1: [INITIALIZATION] SET I = N
Step 2: Repeat Steps 3 and 4 while I >= POS
Step 3:    SET A[I + 1] = A[I]
Step 4:    SET I = I – 1
        [End of Loop]
Step 5: SET N = N + 1
Step 6: SET A[POS] = VAL
Step 7: EXIT
```
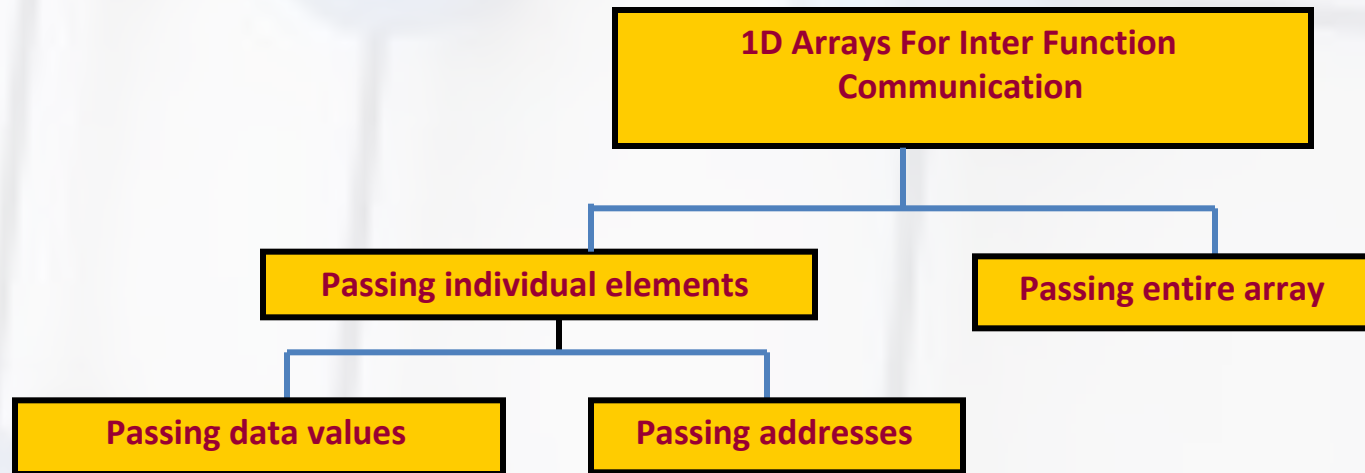
Deleting an Element from an Array

Algorithm to delete an element from the end of the array

```
Step 1: Set upper_bound = upper_bound - 1
Step 2: EXIT
```

Algorithm DELETE(A, N, POS) to delete an element at POS

```
Step 1: [INITIALIZATION] SET I = POS
Step 2: Repeat Steps 3 and 4 while I <= N-1
Step 3:    SET A[I] = A[I + 1]
Step 4:    SET I = I + 1
        [End of Loop]
Step 5: SET N = N - 1
Step 6: EXIT
```

Passing Arrays to Functions

```
┌─────────────────────────────────┐
│   1D Arrays For Inter Function   │
│         Communication            │
└─────────────────────────────────┘
```

Passing individual elements

Passing entire array

Passing data values

Passing addresses

Passing Arrays to Functions

Passing data values

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

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

Passing addresses

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

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

Passing the entire array

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

```
void func(int arr[5])
{
    int i;
    for(i=0;i<5;i++)
    printf("%d", arr[i]);
}
```

Pointers and Arrays

- Concept of array is very much bound to the concept of pointer.

- Name of an array is actually a pointer that points to the first element of the array.

  int *ptr;

  ptr = &arr[0];

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

  int *ptr = &arr[0];

  ptr++;

  printf ("The value of the second element in the array is %d", *ptr);

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. For example, look at the code given below.

    int *ptr[10];

    int p=1, q=2, r=3, s=4, t=5;

    ptr[0]=&p;

    ptr[1]=&q;

Arrays of Pointers

ptr[2]=&r;

ptr[3]=&s;

ptr[4]=&t

Can you tell what will be the output of the following statement?

printf("\n %d", *ptr[3]);

Arrays of Pointers

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.

Two-dimensional Arrays

A two-dimensional array is specified using two subscripts where one subscript denotes row and the other denotes column.

C looks at a two-dimensional array as an array of one-dimensional arrays.

A two-dimensional array is declared as:

data_type array_name[row_size][column_size];

First Dimension

Second Dimension
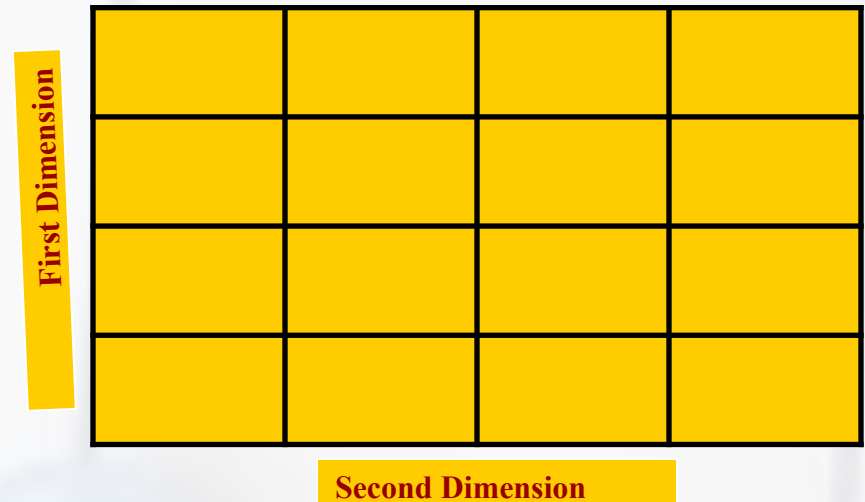
Two-dimensional Arrays

Therefore, a two dimensional m×n array is an array that contains m×n data elements and each element is accessed using two subscripts, i and j, where i<=m and j<=n

int marks[3][5];

| Rows/Columns	Col 0	Col 1	Col2	Col 3	Col 4
Row 0	Marks[0][0]	Marks[0][1]	Marks[0][2]	Marks[0][3]	Marks[0][4]
Row 1	Marks[1][0]	Marks[1][1]	Marks[1][2]	Marks[1][3]	Marks[1][4]
Row 2	Marks[2][0]	Marks[2][1]	Marks[2][2]	Marks[2][3]	Marks[2][4]

Two Dimensional Array

Memory Representation of a 2D Array

- There are two ways of storing a 2-D array in memory. The first way is **row-major order** and the second is **column-major order**.

- In the row-major order the elements of the first row are stored before the elements of the second and third rows. That is, the elements of the array are stored row by row where *n* elements of the first row will occupy the first *n*th locations.

(0,0)	(0, 1)	(0,2)	(0,3)	(1,0)	(1,1)	(1,2)	(1,3)	(2,0)	(2,1)	(2,2)	(2,3)
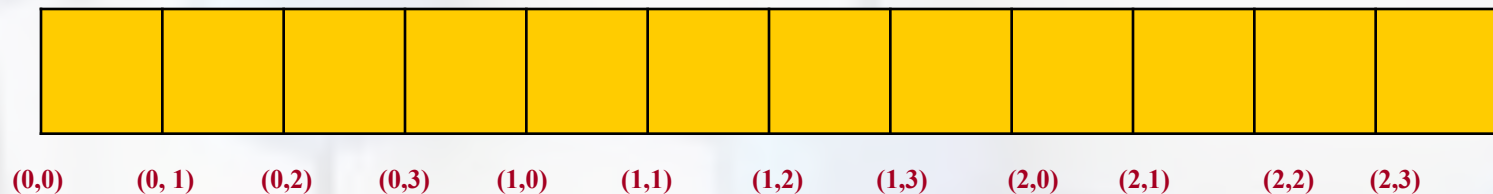
Memory Representation of a 2D Array

- However, when we store the elements in a column major order, the elements of the first column are stored before the elements of the second and third columns. That is, the elements of the array are stored column by column where n elements of the first column will occupy the first nth locations.
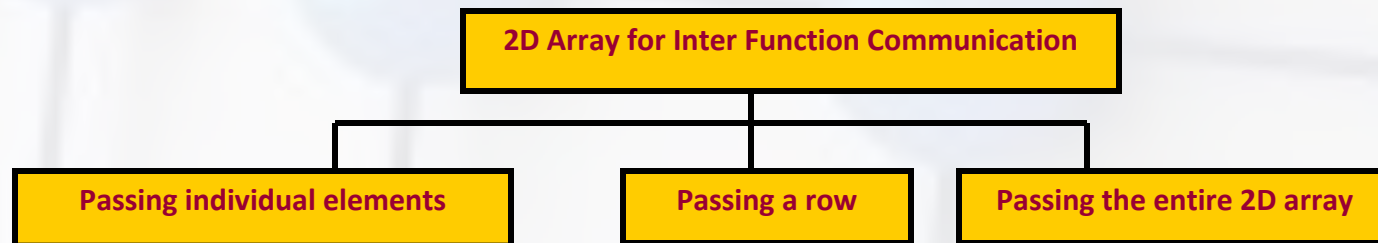
(0,0)	(1,0)	(2,0)	(3,0)	(0,1)	(1,1)	(2,1)	(3,1)	(0,2)	(1,2)	(2,2)	(3,2)
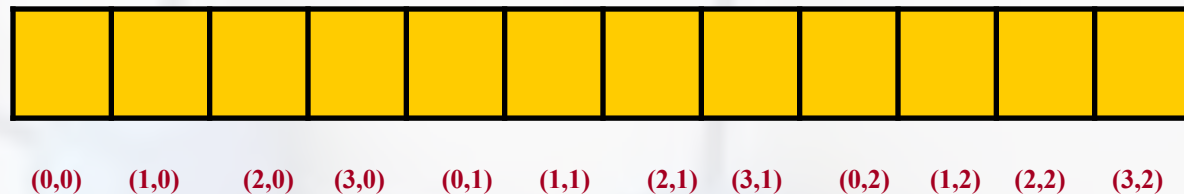
Initializing Two-dimensional Arrays

A two-dimensional array is initialized in the same was as a single

dimensional array is initialized. For example,

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

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

Passing 2D Arrays to Functions

```
2D Array for Inter Function Communication
```

```
Passing individual elements        Passing a row        Passing the entire 2D array
```

There are three ways of passing two-dimensional arrays to a function.

First, we can pass individual elements of the array. This is exactly same as we passed element of a one-dimensional array.

Passing 2D Arrays to Functions

Passing a row

```
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 same we did in case of a 1D array.) However, the parameter in the called function must indicate that the array has two dimensions.

Pointers and 2D Arrays

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

mat[i][j] or *(*(mat + i) + j) or*(mat[i]+j);

Pointer to a one-dimensional array can be declared as:

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

int *parr;

parr=arr;

Similarly, pointer to a two-dimensional array can be declared as:

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

int (*parr)[2];

parr=arr;

Multi-dimensional Arrays

- A multi-dimensional array is an array of arrays.

- Like we have one index in a single dimensional array, two indices in a two-dimensional array, in the same way we have *n* indices in a *n*-dimensional array or multi-dimensional array.

- Conversely, an *n* dimensional array is specified using *n* indices.

- An *n* dimensional m1 x m2 x m3 x ….. mn array is a collection of m1×m2×m3× ….. ×mn 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

$I_1 <= M_1$ $I_2 <= M_2$ $I_3 <= M_3$ …….. $I_n <= M_n$

Multi-dimensional Arrays
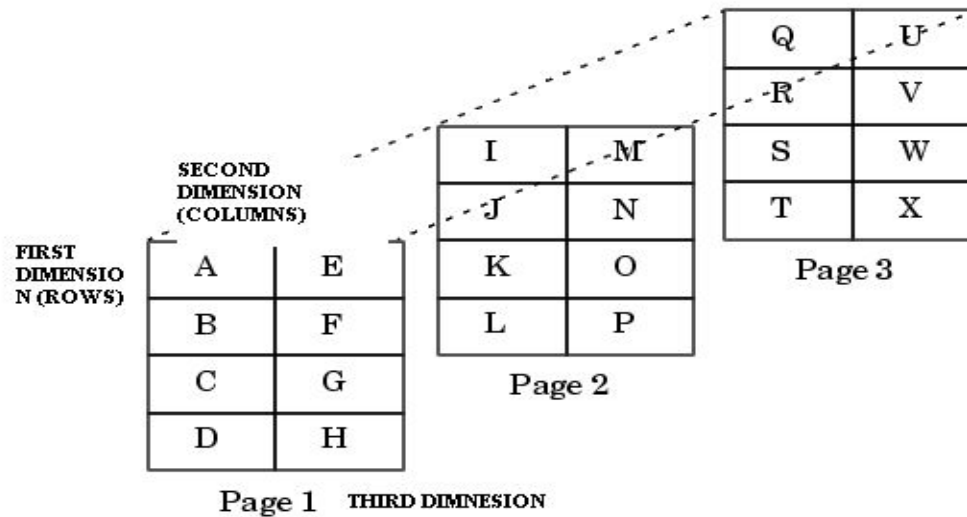
Initializing Multi-dimensional Arrays

- A multi-dimensional array is declared and initialized the same way as we declare and initialize one- and two-dimensional arrays.

Pointers 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)

Applications 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 like heaps, hash tables, deques, queues, stacks and string. We will read about these data structures in the subsequent chapters.

- Arrays can be used for dynamic memory allocation.