BLM267

Chapter 3: Arrays

Data Structures Using C, Second Edition

Reema Thareja

- Introduction
- Declaration of Arrays
- Accessing the Elements of Array
- Storing Values in Arrays
- Operations on Arrays
- Passing Arrays to Functions
- Pointers and Arrays
- Arrays of Pointers
- Two-dimensional Arrays
- Operations on Two-dimensional Arrays
- Passing Two-dimensional Arrays to Functions
- Pointers and Two-dimensional Arrays

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/index is an ordinal number which is used to identify an element of the array.

Declaration of Arrays

- We have already seen that every variable must be declared before it is used.
- The same concept holds true for array variables. An array must be declared before being used.
- Declaring an array means specifying the following:
  - 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.
- Arrays are declared using the following syntax:

  type name[size];
- The type can be either int, float, double, char, or any other valid data type.
- The number within brackets indicates the size of the array, i.e., the maximum number of elements that can be stored in the array.

Declaration of Arrays

- For example, if we write, int marks[10]; then the statement declares marks to be an array containing 10 elements.
- In C, the array index starts from zero.
- The first element will be stored in marks[0], second element in marks[1], and so on.
- Therefore, the last element, that is the 10th element, will be stored in marks[9].
- Note that 0, 1, 2, 3 written within square brackets are the subscripts. In the memory, the array will be stored as shown in Fig. 3.2.

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]

Figure 3.2 Memory representation of an array of 10 elements

Accessing the Elements of an Array

- Storing related data items in a single array enables the programmers to develop concise and efficient programs.
- But there is no single function that can operate on all the elements of an array.
- To access all the elements, 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.

Accessing the Elements of an Array

- Now to process all the elements of the array, we use a loop as shown in Fig. 3.4.
- Figure 3.5 shows the result of the code shown in Fig. 3.4.
- The code accesses every individual element of the array and sets its value to –1.
- In the for loop, first the value of marks[0] is set to –1, then the value of the index (i) is incremented and the next value, that is, marks[1] is set to –1.
- The procedure continues until all the 10 elements of the array are set to –1.

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

Figure 3.4 Code to initialize each element of the array to –1

–1	–1	–1	–1	–1	–1	–1	–1	–1	–1
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

Figure 3.5 Array `marks` after executing the code given in Fig. 3.4

Calculating the Address of Array Elements

- You must be wondering how C gets to know where an individual element of an array is located in the memory.
- The answer is that the array name is a symbolic reference to the address of the first byte of the array.
- When we use the array name, we are actually referring to the first byte of the array.
- The subscript or the index represents the offset from the beginning of the array to the element being referenced.
- That is, with just the array name and the index, C can calculate the address of any element in the array.
- Since an array stores all its data elements in consecutive memory locations, storing just the base address, that is the address of the first element in the array, is sufficient.

Calculating the Address of Array Elements

- The address of other data elements can simply be calculated using the base address.
- The formula to perform this calculation is, Address of data element, A[k] = BA(A) + w(k – lower_bound)
- Here, A is the array, k is the index of the element of which we have to calculate the address, BA is the base address of the array A, and w is the size of one element in memory, for example, size of int is 2.

Example 3.1 Given an array `int marks[]={99,67,78,56,88,90,34,85}`, calculate the address of `marks[4]` if the `base address = 1000`.

Solution

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

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

```
marks[4] = 1000 + 2(4 - 0)
         = 1000 + 2(4) = 1008
```

Calculating the Length of an Array

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

Example 3.2 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
```

Show the memory representation of the array and calculate its length.

Solution

The memory representation of the array `Age[5]` is given as below.

2	5	3	1	7
Age[0]	Age[1]	Age[2]	Age[3]	Age[4]

```
Length = upper_bound - lower_bound + 1
```

Here, `lower_bound = 0, upper_bound = 4`

Therefore, `length = 4 - 0 + 1 = 5`

STORING VALUES IN ARRAYS

- When we declare an array, we are just allocating space for its elements; no values are stored in the array.
- There are three ways to store values in an array.
- First, to initialize the array elements during declaration; second, to input values for individual elements from the keyboard; third, to assign values to individual elements.
- This is shown in Fig. 3.6.

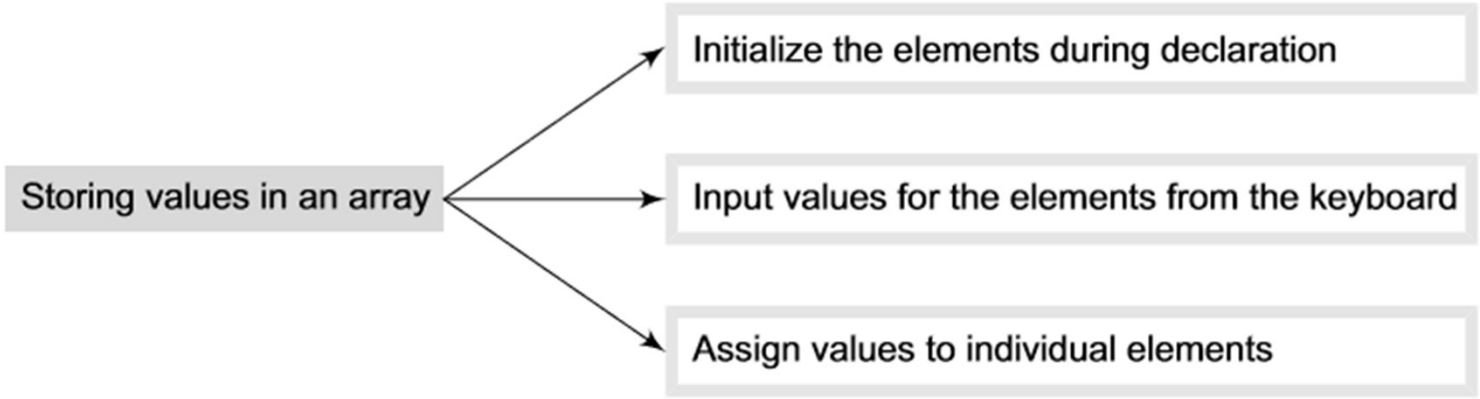

Figure 3.6 Storing values in an array

Initializing Arrays during Declaration

- The elements of an array can be initialized at the time of declaration, just as any other variable.
- When an array is initialized, we need to provide a value for every element in the array.
- Arrays are initialized by writing, type array_name[size]={list of values};
- Note that the values are written within curly brackets and every value is separated by a comma.
- It is a compiler error to specify more values than there are elements in the array.
- When we write, int marks[5]={90, 82, 78, 95, 88}; An array with the name marks is declared that has enough space to store five elements.
- The first element, that is, marks[0] is assigned value 90.
- Similarly, the second element of the array, that is marks[1], is assigned 82, and so on. This is shown in Fig. 3.7.

marks[0]	90
marks[1]	82
marks[2]	78
marks[3]	95
marks[4]	88

Figure 3.7 Initialization of array `marks[5]`

Initializing Arrays during Declaration

- While initializing the array at the time of declaration, the programmer may omit the size of the array.
- For example, int marks[]= {98, 97, 90}; The above statement is absolutely legal.
- Here, the compiler will allocate enough space for all the initialized elements.
- Note that if the number of values provided is less than the number of elements in the array, the un-assigned elements are filled with zeros.
- Figure 3.8 shows the initialization of arrays.

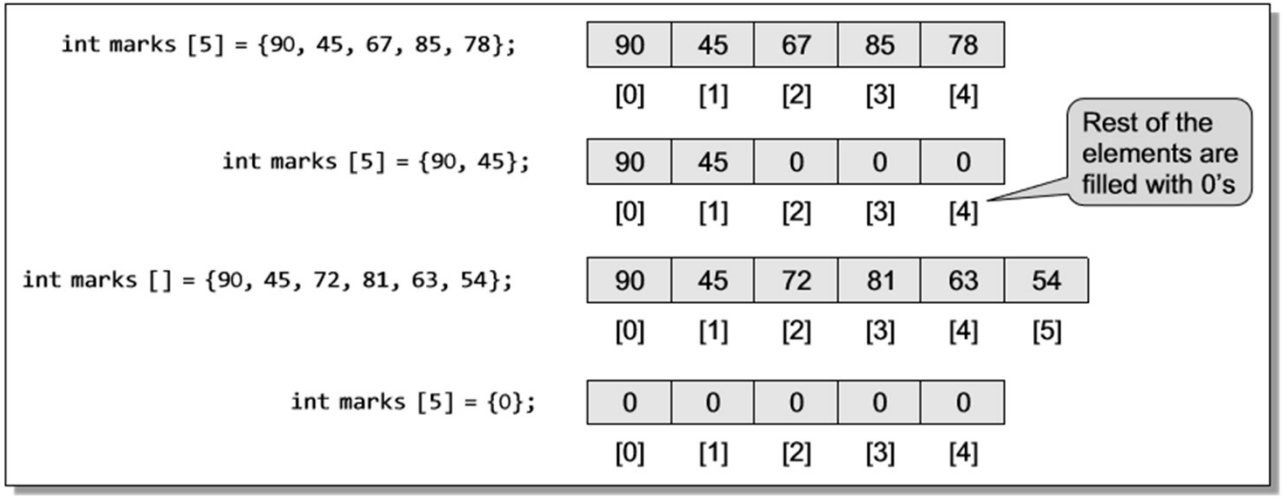

Figure 3.8 Initialization of array elements

Inputting Values from the Keyboard

- An array can be initialized by inputting values from the keyboard.
- In this method, a while/do–while or a for loop is executed to input the value for each element of the array.
- For example, look at the code shown in Fig. 3.9.
- In the code, we start at the index i at 0 and input the value for the first element of the array.
- Since the array has 10 elements, we must input values for elements whose index varies from 0 to 9.

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

Figure 3.9 Code for inputting each element of the array

Assigning Values to Individual Elements

- The third way is to assign values to individual elements of the array by using the assignment operator.
- 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; Here, 100 is assigned to the fourth element of the array which is specified as marks[3].
- Note that 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. Figure 3.10 illustrates the code to copy an 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];
```

Figure 3.10 Code to copy an array at the individual element level

OPERATIONS ON ARRAYS

- There are a number of operations that can be performed on arrays.
- These operations include:
  - Traversing an array
  - Inserting an element in an array
  - Searching an element in an array
  - Deleting an element from an array
  - Merging two arrays
  - Sorting an array in ascending or descending order
- We will discuss all these operations in detail in this section, except searching and sorting, which will be discussed in Chapter 14.

OPERATIONS ON ARRAYS

- **Traversing an array**
  - Traversing an array means accessing each and every element of the array for a specific purpose.
  - Traversing the data elements of an array A can include printing every element, counting the total number of elements, or performing any process on these elements.
  - Since, array is a linear data structure (because all its elements form a sequence), traversing its elements is very simple and straightforward.
  - The algorithm for array traversal is given in Fig. 3.12.

OPERATIONS ON ARRAYS

- **Traversing an array**
  - In Step 1, we initialize the index to the lower bound of the array.
  - In Step 2, a while loop is executed.
  - Step 3 processes the individual array element as specified by the array name and index value.
  - Step 4 increments the index value so that the next array element could be processed.
  - The while loop in Step 2 is executed until all the elements in the array are processed, i.e., until I is less than or equal to the upper bound of the array.

```
Step 1: [INITIALIZATION] SET I = lower_bound
Step 2: Repeat Steps 3 to 4 while I <= upper_bound
Step 3:       Apply Process to A[I]
Step 4:       SET  I = I + 1
        [END OF LOOP]
Step 5: EXIT
```

Figure 3.12 Algorithm for array traversal

OPERATIONS ON ARRAYS

Write a program to print the position of the smallest number of *n* numbers using arrays.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, arr[20], small, pos;
        clrscr();
        printf("\n Enter the number of elements in the array : ");
        scanf("%d", &n);
        printf("\n Enter the elements : ");
        for(i=0;i<n;i++)
                scanf("%d",&arr[i]);
        small = arr[0]
        pos =0;
        for(i=1;i<n;i++)
        {
                if(arr[i]<small)
                {
                        small = arr[i];
                        pos = i;
                }
        }
        printf("\n The smallest element is : %d", small);
        printf("\n The position of the smallest element in the array is : %d", pos);
        return 0;
}
```

Output

```
Enter the number of elements in the array : 5
Enter the elements : 7 6 5 14 3

The smallest element is : 3
The position of the smallest element in the array is : 4
```

OPERATIONS ON ARRAYS

Inserting an element in an array

- If an element has to be inserted at the end of an existing array, then the task of insertion is quite simple.
- We just have to add 1 to the upper_ bound and assign the value.
- Here, we 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.
- Figure 3.13 shows an algorithm to insert a new element to the end of an array.
- In Step 1, we increment the value of the upper bound.
- In Step 2, the new value is stored at the position pointed by the upper bound.

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

Figure 3.13 Algorithm to append a new element to an existing array

OPERATIONS ON ARRAYS

- **Inserting an element in an array**
- For example, let us assume an array has been declared as int marks[60];
- The array is declared to store the marks of all the students in a class.
- Now, suppose there are 54 students and a new student comes and is asked to take the same test.
- The marks of this new student would be stored in marks[55]. Assuming that the student secured 68 marks, we will assign the value as marks[55] = 68;
- However, if we have to insert an element in the middle of the array, then this is not a trivial task.
- On an average, we might have to move as much as half of the elements from their positions in order to accommodate space for the new element.
- For example, consider an array whose elements are arranged in ascending order.
- Now, if a new element has to be added, it will have to be added probably somewhere in the middle of the array.
- To do this, we must first find the location where the new element will be inserted and 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.

OPERATIONS ON ARRAYS

- Algorithm to Insert an Element in the Middle of an Array
- The algorithm INSERT will be declared as INSERT (A, N, POS, VAL).
- The arguments are
  - (a) A, the array in which the element has to be inserted
  - (b) N, the number of elements in the array
  - (c) POS, the position at which the element has to be inserted
  - (d) VAL, the value that has to be inserted

```
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
```

Figure 3.14 Algorithm to insert an element in the middle of an array.

OPERATIONS ON ARRAYS

- **Algorithm to Insert an Element in the Middle of an Array**
  - Now, let us visualize this algorithm by taking an example. Initial Data[] is given as below.

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

  - Calling INSERT (Data, 6, 3, 100) will lead to the following processing in the array:

45	23	34	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

45	23	34	100	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]	Data[6]

OPERATIONS ON ARRAYS

Deleting an element from an array

- Deleting an element from an array means removing a data element from an already existing array.
- If the element has to be deleted from the end of the existing array, then the task of deletion is quite simple.
- We just have to subtract 1 from the upper bound.
- Figure 3.15 shows an algorithm to delete an element from the end of an array.
- For example, if we have an array that is declared as int marks[60]; The array is declared to store the marks of all the students in the class.
- Now, suppose there are 54 students and the student with roll number 54 leaves the course.
- The score of this student was stored in marks[54].
- We just have to decrement the upper bound. Subtracting 1 from the upper bound will indicate that there are 53 valid data in the array.

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

Figure 3.15 Algorithm to delete the last element of an array

OPERATIONS ON ARRAYS

- Deleting an element from an array
- However, if we have to delete an element from the middle of an array, then it is not a trivial task.
- On an average, we might have to move as much as half of the elements from their positions in order to occupy the space of the deleted element.
- For example, consider an array whose elements are arranged in ascending order.
- Now, suppose an element has to be deleted, probably from somewhere in the middle of the array.
- To do this, we must first find the location from where the element has to be deleted and 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.

OPERATIONS ON ARRAYS

- **Algorithm to Delete an Element from the Middle of an Array**

DELETE(A, N, 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
```

Figure 3.16 Algorithm to delete an element from the middle of an array

DELETE (Data, 6, 2)

45	23	34	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]	Data[5]

45	23	12	56	20
Data[0]	Data[1]	Data[2]	Data[3]	Data[4]

Figure 3.17 Deleting elements from an array

Write a program to delete a number from a given location in an array.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int i, n, pos, arr[10];
        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("\nEnter the position from which the number has to be deleted : ");
        scanf("%d", &pos);
        for(i=pos; i<n-1;i++)
                arr[i] = arr[i+1];
        n--;
        printf("\n The array after deletion is : ");
        for(i=0;i<n;i++)
                printf("\n arr[%d] = %d", i, arr[i]);
        getch();
        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
Enter the position from which the number has to be deleted : 3
The array after deletion is :
arr[0] = 1
arr[1] = 2
arr[2] = 3
arr[3] = 5
```

OPERATIONS ON ARRAYS

Merging Two Arrays

- Merging two arrays in a third array means first copying the contents of the first array into the third array and then copying the contents of the second array into the third array.
- Hence, the merged array contains the contents of the first array followed by the contents of the second array.
- 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.
- But merging is not a trivial task when the two arrays are sorted and the merged array also needs to be sorted.
- Let us first discuss the merge operation on unsorted arrays. This operation is shown in Fig 3.18.

Array 1-	90	56	89	77	69							
Array 2-	45	88	76	99	12	58	81					
Array 3-	90	56	89	77	69	45	88	76	99	12	58	81

Figure 3.18 Merging of two unsorted arrays

OPERATIONS ON ARRAYS

Merging Two Arrays

- If we have two sorted arrays and the resultant merged array also needs to be a sorted one, then the task of merging the arrays becomes a little difficult.
- The task of merging can be explained using Fig. 3.19.
- Figure 3.19 shows how the merged array is formed using two sorted arrays.
- Here, we first compare the 1st element of array1 with the 1st element of array2, and then put the smaller element in the merged array.
- Since 20 > 15, we put 15 as the first element in the merged array.

Array 1-	20	30	40	50	60							
Array 2-	15	22	31	45	56	62	78					
Array 3-	15	20	22	30	31	40	45	50	56	60	62	78

Figure 3.19 Merging of two sorted arrays

OPERATIONS ON ARRAYS

Merging Two Arrays

- We then compare the 2nd element of the second array with the 1st element of the first array.
- Since 20 < 22, now 20 is stored as the second element of the merged array.
- Next, the 2nd element of the first array is compared with the 2nd element of the second array.
- Since 30 > 22, we store 22 as the third element of the merged array.
- Now, we will compare the 2nd element of the first array with the 3rd element of the second array.
- Because 30 < 31, we store 30 as the 4th element of the merged array. This procedure will be repeated until elements of both the arrays are placed in the right location in the merged array.

Array 1-	20	30	40	50	60							
Array 2-	15	22	31	45	56	62	78					
Array 3-	15	20	22	30	31	40	45	50	56	60	62	78

Figure 3.19 Merging of two sorted arrays

PASSING ARRAYS TO FUNCTIONS

- Like variables of other data types, we can also pass an array to a function.
- In some situations, you may want to pass individual elements of the array; while in other situations, you may want to pass the entire array.
- In this section, we will discuss both the cases.
- Look at Fig. 3.20 which will help you understand the concept.

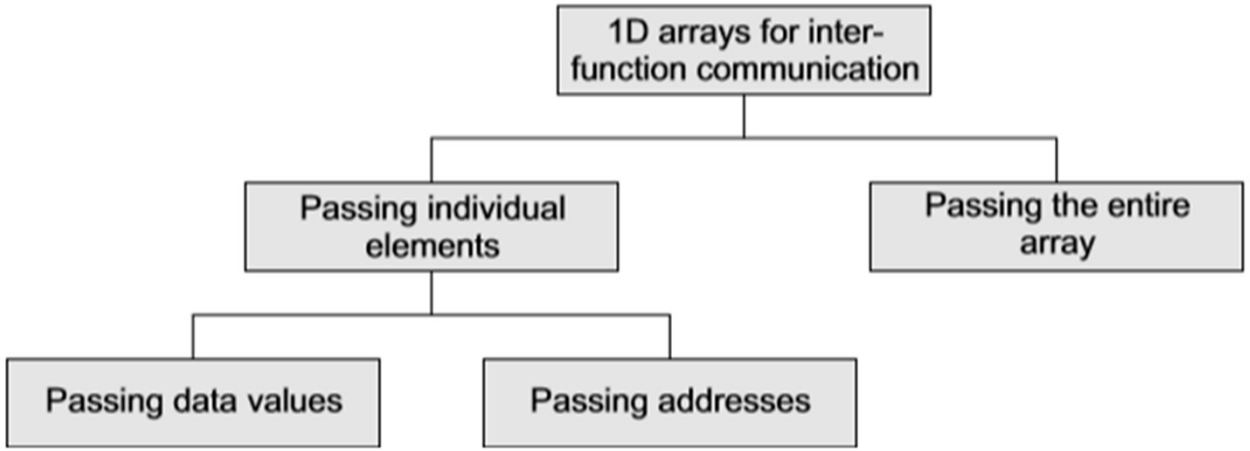

Figure 3.20 One dimensional arrays for inter-function communication

PASSING ARRAYS TO FUNCTIONS

Calling function	Called function

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

Figure 3.21(a) Passing values of individual array elements to a function

- **Passing Data Values**
- 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 does not know whether a normal integer variable is passed to it or an array value is passed.

PASSING ARRAYS TO FUNCTIONS

- 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.
- Look at the code shown in Fig. 3.21(b).

Calling function	Called function

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

Figure 3.21(b) Passing addresses of individual array elements to a function

PASSING ARRAYS TO FUNCTIONS

Passing the Entire Array

- We have discussed that 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.
- Figure 3.22 illustrates the code which passes the entire array to the called function.

Calling function

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

Called function

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

Figure 3.22 Passing entire array to a function

PASSING ARRAYS TO FUNCTIONS

Passing the Entire 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);

- 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.
- 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);

PASSING ARRAYS TO FUNCTIONS

- **Passing the Entire Array**
- It is not necessary to pass the whole array to a function.
- 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);
- Note that 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 const before the data type of the array.

POINTERS AND ARRAYS

- Execute the code given below and observe the output

```
main()
{
        int arr[]={1,2,3,4,5};
        int *ptr;
        ptr = &arr[0];
        printf("\n Address of array = %p %p %p %p", arr,  &arr[0], &arr, ptr);
}
```

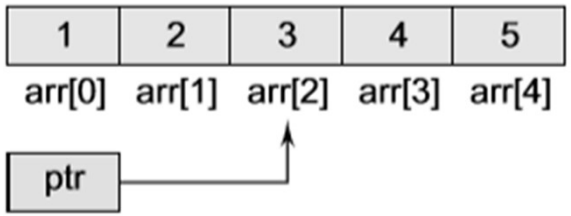

Figure 3.24 Pointer pointing to the third element of the array

Programming Tip

An error is generated if an attempt is made to change the address of the array.

POINTERS AND ARRAYS

- 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++.

```
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.

POINTERS AND ARRAYS

- 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).
- Had this been a character array, every byte in the memory would have been used to store an individual character. ptr++ would then add only 1 byte to the address of ptr.

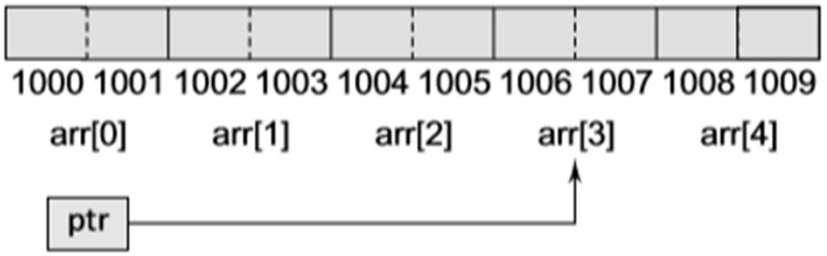

Figure 3.25 Pointer (ptr) pointing to the fourth element of the array

Programming Tip

When an array is passed to a function, we are actually passing a pointer to the function. Therefore, in the function declaration you must declare a pointer to receive the array name.

POINTERS AND ARRAYS

- When using pointers, an expression like arr[i] is equivalent to writing *(arr+i).
- Many beginners get confused by thinking of array name as a pointer.
- 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.

POINTERS AND ARRAYS

- 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.

POINTERS AND ARRAYS

- Look at the following code which modifies the contents of an array using a pointer to an array.

```
int main()
{
        int arr[]={1,2,3,4,5};
        int *ptr, i;
        ptr=&arr[2];
        *ptr = –1;
        *(ptr+1) = 0;
        *(ptr–1) = 1;
        printf("\n Array is: ");
        for(i=0;i<5;i++)
                printf(" %d", *(arr+i));
        return 0;
}
```

Output

Array is: 1 1 –1 0 5

POINTERS AND ARRAYS

- In C 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.
- For example, look at the code given below.

```
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

- Like with other variables, relational operators (>, <, >=, etc.) can also be applied to pointer variables.

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;
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

- Now look at another code in which we store the address of three individual arrays in the array of pointers:

```
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

TWO-DIMENSIONAL ARRAYS

- Till now, we have only discussed one-dimensional arrays.
- One-dimensional arrays are organized linearly in only one direction.
- But at times, we need to store data in the form of grids or tables.
- Here, the concept of single-dimension arrays is extended to incorporate two-dimensional data structures.
- A two-dimensional array is specified using two subscripts where the first subscript denotes the row and the second denotes the column.
- The C compiler treats a two-dimensional array as an array of one-dimensional arrays.
- Figure 3.26 shows a two-dimensional array which can be viewed as an array of arrays.

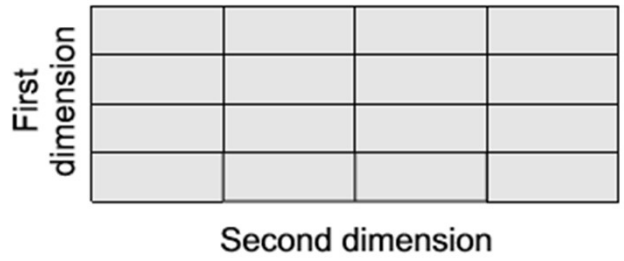

Figure 3.26 Two-dimensional array

TWO-DIMENSIONAL ARRAYS

- Declaring Two-dimensional arrays
- Any array must be declared before being used.
- The declaration statement tells the compiler the name of the array, the data type of each element in the array, and the size of each dimension.
- A two-dimensional array is declared as:
- data_type array_name[row_size][column_size];
- Therefore, a two-dimensional m X n array is an array that contains m X n data elements and each element is accessed using two subscripts, i and j, where i < m and j < n.
- For example, if we want to store the marks obtained by three students in five different subjects, we can declare a two dimensional array as:
- int marks[3][5];

TWO-DIMENSIONAL ARRAYS

- **Declaring Two-dimensional arrays**
- **The first element of the array is denoted by marks[0][0], the second element as marks[0][1], and so on.**
- **Here, marks[0][0] stores the marks obtained by the first student in the first subject, marks[1][0] stores the marks obtained by the second student in the first subject.**
- **The pictorial form of a two-dimensional array is shown in Fig. 3.27.**

Rows / Columns	Col 0	Col 1	Col 2	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]

Figure 3.27 Two-dimensional array

TWO-DIMENSIONAL ARRAYS

- **Declaring Two-dimensional arrays**
- Hence, we see that 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.
- To understand this, we can also see the representation of a two-dimensional array as shown in Fig. 3.28.

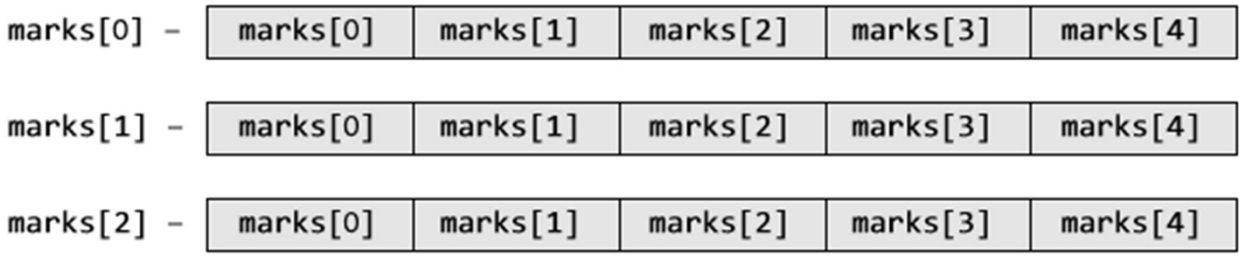

Figure 3.28 Representation of two-dimensional array `marks[3][5]`

TWO-DIMENSIONAL ARRAYS

Declaring Two-dimensional arrays

- Although we have shown a rectangular picture of a two-dimensional array, in the memory, these elements actually 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 column major order.
- Let us see how the elements of a 2D array are stored in a row major order.
- Here, 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 locations. This is illustrated in Fig. 3.29.

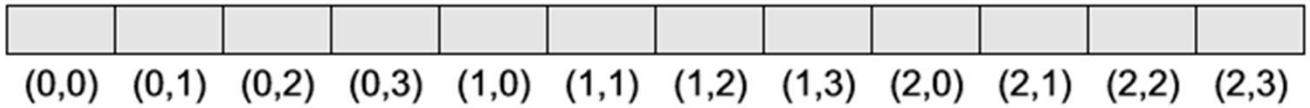

Figure 3.29 Elements of a 3 × 4 2D array in row major order

TWO-DIMENSIONAL ARRAYS

- Declaring Two-dimensional arrays
- 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 column.
- That is, the elements of the array are stored column by column where m elements of the first column will occupy the first m locations.
- This is illustrated in Fig. 3.30

Figure 3.30 Elements of a 4 × 3 2D array in column major order

TWO-DIMENSIONAL ARRAYS

- **Declaring Two-dimensional arrays**
- In one-dimensional arrays, we have seen that the computer does not keep track of the address of every element in the array.
- It stores only the address of the first element and calculates the address of other elements from the base address (address of the first element).
- Same is the case with a two-dimensional array.
- Here also, the computer stores the base address, and the address of the other elements is calculated using the following formula.
- If the array elements are stored in column major order,

Address(A[i][j]) = Base_Address + w(M * j + i)

- And if the array elements are stored in row major order,

Address(A[i][j]) = Base_Address + w(N * i + j)

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.

1	2	3	4
5	6	7	8
9	10	11	12

0,0	1,0	2,0	0,1	1,1	2,1	0,2	1,2	2,2	0,3	**1,3**	2,3
1	5	9	2	6	10	3	7	11	4	**8**	12
1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	**1020**	1022

Address(A[i][j]) = Base_Address + w(M * j + i)

Address(A[i][j]) = Base_Address + w(3 * 3 + i)

Address(A[i][j]) = Base_Address + w(9+ 1)

Address(A[i][j]) = Base_Address + 2*10

Address(A[i][j]) = Base_Address + 2*10

TWO-DIMENSIONAL ARRAYS

- Initializing Two-dimensional arrays
- Like in the case of other variables, declaring a two-dimensional array only reserves space for the array in the memory.
- No values are stored in it. 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};
- Note that the initialization of a two-dimensional array is done row by row.
- The above statement can also be written as:

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

TWO-DIMENSIONAL ARRAYS

- Initializing Two-dimensional arrays
- In case of one-dimensional arrays, we have discussed that if the array is completely initialized, we may omit the size of the array.

```
int marks[]={1,2,3,4,5}
```

- The same concept can be applied to a two-dimensional array, except that only the size of the first dimension can be omitted.

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

- In order to initialize the entire two-dimensional array to zeros, simply specify the first value as zero.
- That is, int marks[2][3] = {0};

TWO-DIMENSIONAL ARRAYS

- Initializing Two-dimensional arrays
- The individual elements of a two-dimensional array can be initialized using the assignment operator as shown here.
- marks[1][2] = 79;
- or marks[1][2] = marks[1][1] + 10;

TWO-DIMENSIONAL ARRAYS

- Accessing the elements of Two-dimensional array
- The elements of a 2D array are stored in contiguous memory locations.
- In case of one-dimensional arrays, we used a single for loop to vary the index i in every pass, so that all the elements could be scanned.
- 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.
- Look at the programs which use two for loops to access the elements of a 2D array.

PASSING TWO-DIMENSIONAL ARRAYS TO FUNCTIONS

- There are three ways of passing a two-dimensional array to a function.
- First, we can pass individual elements of the array.
- This is exactly the same as passing an element of a one-dimensional array.
- Second, we can pass a single row of the two-dimensional array.
- This is equivalent to passing the entire one-dimensional array to a function that has already been discussed in a previous section.
- Third, we can pass the entire two-dimensional array to the function.

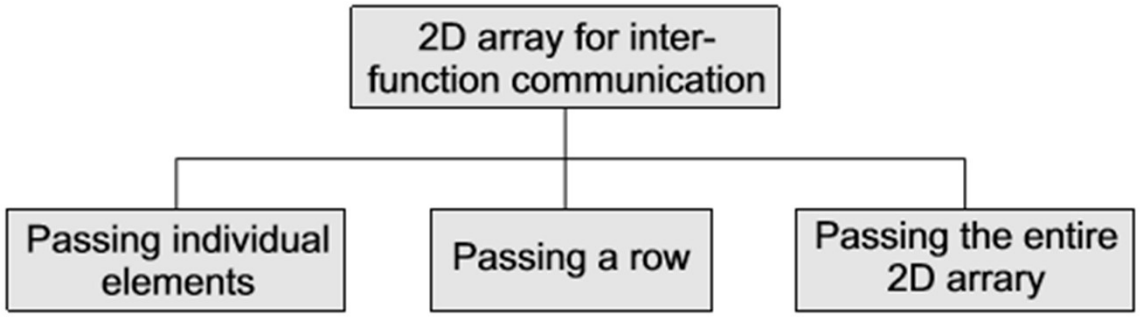

Figure 3.31 2D arrays for inter-function communication

PASSING TWO-DIMENSIONAL ARRAYS TO FUNCTIONS

- Passing a Row
- A row of a two-dimensional array can be passed by indexing the array name with the row number.
- Look at Fig. 3.32 which illustrates how a single row of a two-dimensional array can be passed to the called function.

Calling function	Called function

`main()`
`{`
`int arr[2][3] = ({1, 2, 3}, {4, 5, 6});`
`func(arr[1]);`
`}`
	`void func(int arr[])`
	`{`
	`int i;`
	`for(i=0;i<3;i++)`
	`printf("%d", arr[i] * 10);`
	`}`

Figure 3.32 Passing a row of a 2D array to a function

PASSING TWO-DIMENSIONAL ARRAYS TO FUNCTIONS

- 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.

Calling function	Called function

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

POINTERS AND TWO-DIMENSIONAL ARRAYS

- To declare a pointer to a two-dimensional array, you may write int **ptr
- Consider a two-dimensional array declared as int mat[5][5];
- Individual elements of the array mat can be accessed using either: mat[i][j] or *(*(mat + i) + j) or *(mat[i]+j);

POINTERS AND TWO-DIMENSIONAL ARRAYS

- To understand more fully the concept of pointers, let us replace *(mat+ i) with X so the expression *(*(mat + i) + j) becomes *(X + j)
- Using pointer arithmetic, we know that the address pointed to by (i.e., value of) X + j+ 1 must be greater than the address X + jby an amount equal to sizeof(int).
- Since mat is a two-dimensional array, we know that in the expression mat+ i as used above, mat+ i+ 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).

POINTERS AND TWO-DIMENSIONAL ARRAYS

- Thus, 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.

POINTERS AND TWO-DIMENSIONAL ARRAYS

- 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 + 1 is address 1010. To summarize, &arr[0][0], arr[0], arr, and &arr[0] point to the base address.

- 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.

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

POINTERS AND TWO-DIMENSIONAL ARRAYS

```
#include <stdio.h>
int main()
{
        int arr[2][3]={{1,2,3},{4,5,6}};
        printf("%d\n", arr[0][2]);
        printf("%d\n", (*(arr+1))[1]);
        printf("%d\n", *((*(arr+1))+2));
        printf("%d\n", *(arr[0]+2));
}
```

POINTERS AND TWO-DIMENSIONAL ARRAYS

```
#include <stdio.h>
int main()
{
        int arr[2][3]={{1,2,3},{4,5,6}};
        printf("%d\n", arr[0][2]);
        printf("%d\n", (*(arr+1))[1]);
        printf("%d\n", *((*(arr+1))+2));
        printf("%d\n", *(arr[0]+2));
}
```

Output:

3

5

6

3

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)
```

POINTERS AND TWO-DIMENSIONAL ARRAYS

```
#include<stdio.h>
int main()
{
        int arr[2][3]={{1,2,3},{4,5,6}};
        int i, j, (*parr)[3];
        parr=arr;
        for(i=0; i<2; i++)     {
                for(j=0; j<3; j++)
                        printf(" %d", (*(parr+i))[j]);
        }
        printf("\n");
        return 0;
}
```

POINTERS AND TWO-DIMENSIONAL ARRAYS

```
#include<stdio.h>
int main()
{
        int arr[2][3]={{1,2,3},{4,5,6}};
        int i, j, (*parr)[3];
        parr=arr;
        for(i=0; i<2; i++)     {
                for(j=0; j<3; j++)
                        printf(" %d", (*(parr+i))[j]);
        }
        printf("\n");
        return 0;
}
```

Output:

1 2 3 4 5 6