# Unit - 2

Chapter 8

# Arrays

## Chapter Outline

- Array - Meaning and Definition
- Single-Dimensional Arrays (1D Arrays)
  - Declaration of an Arrays
  - Array Initialization
  - Accessing the Elements of an Array
  - Memory Representation of Single Dimensional Array
  - Boundary Checking
- Two Dimensional Arrays
  - Declaration of Two Dimensional Array
  - Initializing Two Dimensional Array
  - Accessing the Elements of Two Dimensional Arrays
  - Memory Representation of Two Dimensional Array
- Advantages and Disadvantages of Arrays
- Applications of Arrays
- Sample Program
- Review Questions

## 8.1 Array - Meaning and Definition

So far we have seen a variable of fundamental data types namely char, int, float, and double. The variables of these data types can store only one value at any given time. Therefore they can be used only to handle limited amount of data. In many scenarios, we need to handle large volume of data for the purpose of reading, processing and printing. C Supports a **derived data type** known as **array** and it handles storing, accessing and manipulation of large data items efficiently.

**Why do we need Arrays?**

Imagine we have a problem that requires us to read, process and print 100 roll numbers of students. To begin, we can declare and define 100 variables, each with a different name as shown below;

int r1, r2, r3, r4, r5, r6, r7, r8, r9, r10.....r100;

This creates 100 variables . To read 100 integers from the keyboard, we need 100 read statements. To print them we need 100 print statements.

Although this approach may be acceptable for less variables it is definitely not acceptable for 100 or 1000 or 10000 variables.

Hence, we need more powerful data structure to process large amounts of data. Hence the array is used for processing large amounts of data of same type.

Arrays gives the capability to store the 100 roll numbers in the contiguous memory locations which can be accessed by single variable name.

**Definitions : Array**

- An array in C can be defined as "a collection of data items of the same data type which are stored in consecutive memory locations and referred by the common name".
- an array is a collection of homogeneous data elements. Homogeneous means all the individual data elements are of same data type.

An array is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using index of an array. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. The rules for assigning names to arrays are the same as for variable names. An array name must be unique. It can't be used for another array or for any other identifier (variable, constant, and so on). All the similar data items can be placed in an array and can be accessed by the same name. The array is the simplest data structure where each data element can be randomly accessed by using its index number.

An array can be of any variable type.

```
int regno[20];
float salary[100];
char name[40];
```

The above examples are the arrays of **integer** type , **character** type and **float** type that can store 20 100 and 40 data elements respectively. These data elements are known as **members** of the arrays.

Array is beneficial if we have to store similar elements. For example, if we want to store the marks of a student in 5 subjects, then we don't need to define different variables for the marks in the different subject. Instead of that, we can define an array which can store the marks in each subject at the contiguous memory locations. So, it is clear that arrays reduce the program complexity and helps us to develop efficient programs.

**What is Contiguous Memory?**

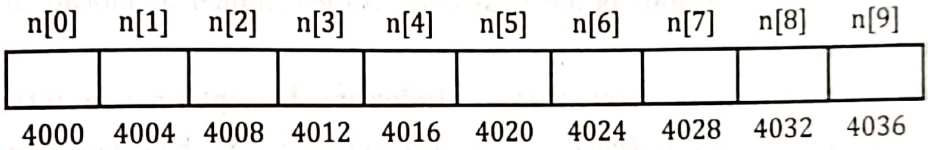
Consecutive blocks of memory allocated to user processes are called contiguous memory. For example, if a user process needs some x bytes of contiguous memory, then all the x bytes will reside in one place in the memory

**Example:**

If we create an array of 10 integers, 40 bytes of contiguous memory (for 64 bit machine) is allocated.

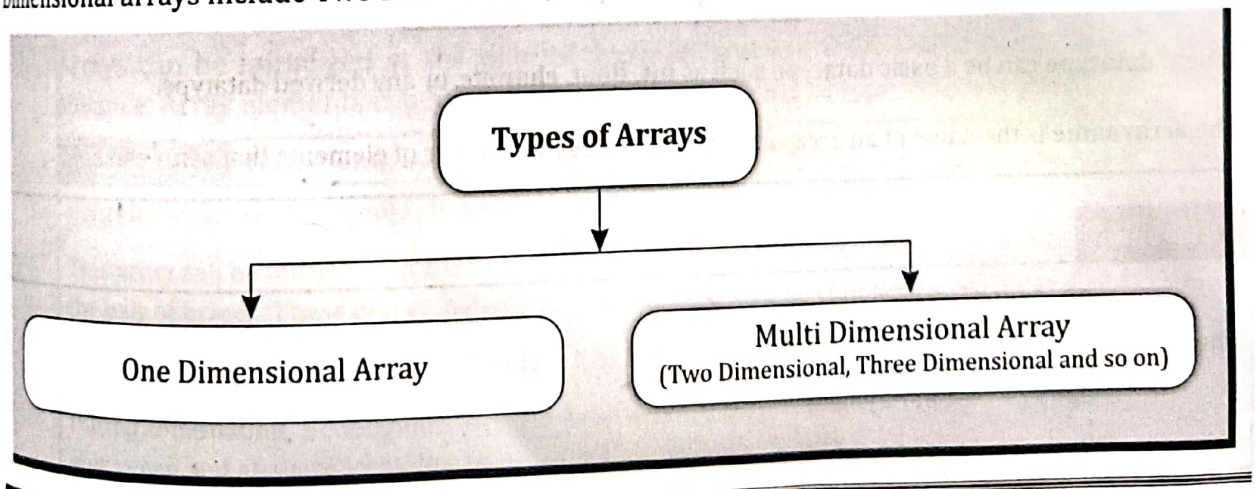
```
int n[10] ;
```

The way the integer array 'n' is stored in memory can be understood from the following figure. Lets say system allocates memory from 4000 to 4040 to store the integer elements of an array. The base address is 4000. The first element n[0] stores in the memory location 4000 to 4003.




## Types of Arrays

Arrays are categorized as **Single or One Dimensional** and **Multi Dimensional** arrays. The Multi Dimensional arrays include Two Dimensional, Three Dimensional, and so on.




## 8.2 Single-Dimensional Arrays (1D Arrays)

A single-dimensional array is the simplest form of an array that requires only one subscript to access an array element. Like an ordinary variable, an array must have been declared before it is used in the program.

**Definitions : Single Dimensional Array**

**Single-dimensional array** or **One-dimensional array** or **linear array** can be defined as an array with n elements where each element is stored in consecutive memory location. To store n elements together all the n memory locations have the same array name and are contiguous to each other. 1-D-array contains one column of n-elements. The consecutive memory locations means all the elements are stored in sequential fashion in memory (that is one-after-the other).

Thus for example **regno[10]** represent one dimensional array with 10 elements that require 10 consecutive memory locations in memory. The one-dimensional array has a single subscript and subscript must be specified in pair of square brackets. In C, arrays elements begin with subscript number 0. Therefore regno[0] refers to the element of the $0^{th}$ position or it refers to the first element of the array.

## 8.2.1 Declaration of an Arrays

We know that all the variables are declared before they are used in the program. Similarly, an array must be declared before it is used. During declaration, the size of the array has to be specified. The size used during declaration of the array informs the compiler to allocate and reserve the specified memory locations.

Each element of the array is accessed via an **index** or **subscript.** An index is the integer value specified within the square brackets. One dimensional array will always have only one subscript/index.

**Syntax**

The syntax for one-dimensional array declaration is

```
datatype arrayname[size];
```

Where **datatype** can be a basic datatype such as **int, float, char** etc. or any derived datatype.

The **arrayname** is the name of an array and **size** indicates the number of elements that array can hold

**Example**

```
        int regno[20];
where   int      → datatype
        regno    → arrayname
        20       → size of the array
```

**Note**

It is very important to note that all the elements/members of the array must be of same data type. In the above example, the array 'regno' has to contain only the integer numbers.

**Example**

```
int regno[20];          // Array of Integer data type that can store 20 integer elements
float average[10];      // Array of float data type that can store 10 float elements
char name[40];          // Array of characters data type that can store 40 characters
double salary[10];      // Array of double data type that can store 10 double elements.
```

It is also always advisable to specify the number and elements (size) with a literal constant or with a symbolic constant created with the **#define** directive. Thus, the following:

```
#define COUNT 12
int month[COUNT];
```

is equivalent to the statement:

```
int month[12];
```

Here, count value is 12 and hence 12 locations will be reserved for the variable month. Since it is an integer array, let us consider that integer takes 2 bytes if memory the it reserves 12 × 2 = 24 bytes of memory space for an array. if integer takes 4 bytes of memory, then it reserves 12 × 4 = 48 bytes of memory.

However, we can't declare an array's elements with a symbolic constant created with the **const** keyword.

```
i.e., const int COUNT = 12;
int month[COUNT]; /*WRONG*/
```

### 8.2.2 Array Initialization

Providing a value for each element of the array is called as initialization. The list of values must be enclosed in curly braces. There are different ways of initializing arrays. Lets look at each one.

1. **Initializing an array with size and initial values:**

   Arrays can be initialized at the time of declaration when their initial values are known in advance. Array elements can be initialized with data items of type int, char etc.

**Example** **Initializing an array with size and initial values**

Consider an array int regno[5] is declared.

This array can be initialized at the declaration time itself followed by an '=' sign and then followed by the pair of braces. These braces contain the values, separated by commas.

```
i.e. int regno[5] = {10, 11, 12, 13, 14};
```

During compilation, 5 contiguous memory locations are reserved by the compiler for the array variable regno and all these locations are initialized as shown in figure.

| regno[0] | regno[1] | regno[2] | regno[3] | regno[4] |
|---|---|---|---|---|
| 10 | 11 | 12 | 13 | 14 |
| 4000 | 4004 | 4008 | 4012 | 4016 |

This declaration statement initializes 5 elements of array regno as shown below

```
So    regno[0] = 10      /*First student register number*/
      regno[1] = 11      /*Second student register number*/
      regno[2] = 12      /*Third student register number*/
      regno[3] = 13      /*Fourth student register number*/
      regno[4] = 14      /*Fifth student register number*/
```

**Note:** If no of initial values are more than the size of array, then we get compiler error.

**Example :** `int a[3]={9,2,4,5,6}; //error: no. of initial vales are more than the size of array (3)`

### 2. Initialize an array without specifying size and with initial values:

**Example** **Initialize an array without specifying size and with initial values**

```
int regno[]={10,20,30,40,50}
```

In this declaration, even though we have not specified exact number of elements to be used in array, the array size will be set of the total number of initial values specified. So, the array size will be set to 5 automatically.

During array initialization, if all the elements of the array are going to be initialized, then it is not necessary to specify the array size. Because array size will be taken as the total number of initial values included within a pair of braces.

Similarly we can initialize the character array without size as shown below.

```
char name [] = {'S', 'R', 'I', 'K', 'A', 'N', 'T', 'H', '\0'};
```

This declaration statement initializes 9 elements of array **name** as shown below.

```
name[0] = 'S'  name[1] = 'R'  name[2] = 'I'  name[3] = 'K'  name[4] = 'A'
name[5] = 'N   name[6] = 'T   name[7] = 'H   name[8] = '\0'
```

The compiler terminates with the string with a '\0' or NULL character.

The compiler automatically inserts the null character at the end if we initialize the character array with string in double quotes as shown below.

```
char name[]="SRIKANTH";
```

### 3. Partial array initialization:

**Example** **Partial Array Initialization**

Partial array initialization is possible in c language. If the number of values to be initialized is less than the size of the array, then the elements will be initialized to zero automatically.

```
int regno[5] = {10, 11};
```

Even though compiler allocates 5 memory locations, using this declaration statement;

the compiler initializes first two locations with 10 and 15, the next set of memory locations are automatically initialized to 0's by compiler as shown in figure.

| regno[0] | regno[1] | regno[2] | regno[3] | regno[4] |
|---|---|---|---|---|
| 10 | 11 | 0 | 0 | 0 |
| 4000 | 4004 | 4008 | 4012 | 4016 |

4. **Runtime Initialization using input functions.**

**Example** **Runtime Initialization**

An array can be explicitly initialized at run time. This approach is usually applied for initializing large arrays. The below code uses scanf to initialize an array.

```
int x[3];
scanf("%d%d%d",&x[0],&x[1],&x[2]);
```

The above statements will initialize array elements with the values entered through the keyboard.

**Example** **Runtime Initialization**

Lets consider another code below.

```
int i,
float sum[100];
for(i=0;i<100;i=i+1)
{
    if(i<50)
        sum[i]=0.0;
    else
        sum[i]=1.0;
}
```

The first 50 elements of the array sum are initialized to 0 while the remaining 50 are initialized to 1.0 at run time.

**Example**

The following code declares and initializes the array of 5 elements. The array elements are printed using **printf** statement

```
int a[5]={10, 20, 30, 40, 50};
int i ;
for (i=0 ; i<5 ; i++)
{
    printf("%d", a[i]);
}
```

The above loop executes 5 times for the values of 'i' ranging from 0 to 4. First it prints a[0], then a[1], then a[2], and so on. The above printf statement prints 10 20 30 40 50.

### 8.2.3 Accessing the Elements of an Array

Each element of the array is accessed via an **index or subscript.** An index is the integer value specified within the square brackets. One dimensional array will always have only one subscript/index.

Array elements are always numbered starting from 0 to (n - 1) where 'n' is the size of the array. The index of the array specifies the element's position in the array but the exact element position is always one more than the index value.

**Example**

```
int regno[5] = {10, 11, 12, 13, 14};
```

The array elements will be access using the subscript/index as shown below.

regno[0] specifies the first position in the array,

```
printf("%d",regno[t0]);
```
It prints 10

regno[1] specifies the second position in the array,

```
printf("%d",regno[1]);
```
It prints 11

Hence, to access the $i^{th}$ student registration no, we should print regno[i-1] value.

The lower bound in C is always zero. i.e., the array starts from the index 0. Using an array we can access any element by specifying the index of the element. It is better to specify the size of an array using a symbolic constant rather than the fixed quantity.

**Example** **The following code declares on array called A of size 20 and stores the n elements in the array.**

```
#define SIZE 20
main( )
{
      int A[size],n,i;
      printf("Enter the number of elements:");
      scanf("%d",&n);
      for (i=0;i<n;i++)
      {
         scanf("%d", &A[i]);
      }
}
```

Here, first we get the value of 'n' through the keyboard and **for loop** will be executed 'n' times for the values of 'i' ranging from 0 to n-1. For the first time within the loop, the value of 'i' will be zero. So, the first element entered will be stored in A[0]. Then 'i' becomes 1 and second element will be stored in A[1] and. so on. Finally when 'i' is (n-1), A[n-1] contains the $n^{th}$ element. Suppose 'n' is 5, the above code accepts 5 elements and stores the elements in A[0], A[1],, A[2], A[3], and A[4] respectively.

**Program 1** Program to read size and elements for a one-dimensional array from user, and print it back as output.

```
#include<stdio.h>

void main()
{
   int arr[50], size, i;
   printf("Enter Array Size: ");
   scanf("%d", &size);
   printf("\nEnter %d Elements: ", size);

   for(i=0; i<size; i++)
      scanf("%d", &arr[i]);

   printf("\nArray Elements are :\n");
   for(i=0; i<size; i++)
      printf("%d ", arr[i]);
}
```

**Output**

```
Enter Array Size: 5

Enter 5 Elements: 10 20 30 40 50

Array Elements are :
10 20 30 40 50
```

**Explanation**

In the above program, user enters 5 as the size for array, then 5 gets initialized to size variable. Initially the value of i is 0 and 0 is less than the value of size variable. So program flow goes inside the for loop, scans a value and stores at arr[0].

Program flow increments the value of i using the third statement of for loop, and its value becomes 1. Now 1 again gets checked whether it is less than the value of size or not. Because 1 is again less than the value of size variable (that holds 5 as its value). Then program flow again goes inside the loop, and scans another value and stores this value at arr[i] or arr[1].

This process continues until the condition of the for loop evaluates to be false, that is when the value of i becomes 5, then it means, 5 is not less than 5 (value of size). So program flow exits from the first for loop.

There is another for loop, to print the array elements as shown in the output.

**Example** The following code is used to print 'n' elements of an array.

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

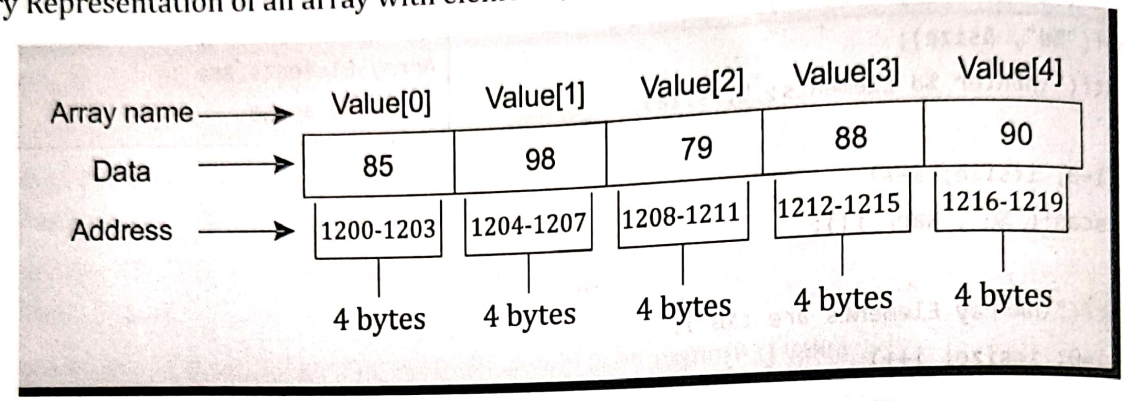
Firstly, the value of 'i' is zero and it prints a[0] value. Then 'i' value is incremented by 1 and 'i' becomes 1. Secondly, it prints a[1] value and so on. If n = 5, then it prints the values of a[0], a[1], a[2], a[3] and a[4] respectively.

## 8.2.4 Memory Representation of Single Dimensional Array

A single dimensional array is a linear list consisting of related and similar data items. In memory, all the data items are stored in contiguous memory locations one after the other. Let us consider an Array as shown below:

```
int Value[5]={ 85, 98, 79, 88, 90};
```

Memory Representation of an array with elements, address and indexes is shown below.




In the above example, the range refers to 0-4, i.e., 5 numbers. It is to be noted that 'C' arrays starts from $0^{th}$ location. So the range starts from 0 and end with 4 for 5 numbers. Hence, an array of n elements has a range from zero to (n-1). So the index/subscript always holds the integer values starting from zero.

The memory requirement for the array depends on the type of data items stored and number of items. Let us assume data items are of type int and assume that int takes 4 bytes (in majority of PCs size of integer will be 4 bytes). Since 5 elements are stored and the size of int is 4 bytes, the memory required for the array is 40 bytes. If the size of integer is 2 bytes, the array uses 20 bytes of memory and so on in general.

The type of **Value** is int, so each elements of the Value array will take 2 or 4 bytes of memory. For example, consider the starting address of first memory block is 1200. So the next elements of the array will get next contiguous 4 bytes memory block and the memory address will be 1004 (as 1000 + 4 bytes) and so on for other elements. In the above figure, the addresses are displayed here, are the starting address of each memory block assigned to individual element of the array.

**Memory for array = n * size of data type**

Where

- n is the number of data items
- data type is the type of data items

The address of the first array element is called base address. In the above figure, the base address is 1200. The base address of an array A is denoted by Base(A). i.e., Base(A)= 1200

Using the base address, we can calculate the address of any element using the below formula.

**Address(A[i])=Base(A) + i * Size of element**

Address(A[3]) = Base(A) + 3 * Size of element = 1200 + 3 * 4; = 1212

Similarly addresses of the other elements can be calculated as shown below:

Address(A[0]) = 1200 + 0 * 4 = 1200

Address(A[1]) = 1200 + 1 * 4 = 1204

Address(A[2]) = 1200 + 2 * 4 = 1208

Address(A[4]) = 1200 + 4 * 4 = 1216

**Example** To Demonstrate Contiguous Memory Locations of Single Dimensional Arrays

```
#include <stdio.h>
void main()
{
    int Value[5]={ 85, 98, 79, 88, 90};
    printf("Address of First Element  = %d \n", &Value[0]);
    printf("Address of Second Element = %d \n", &Value[1]);
    printf("Address of Third Element  = %d \n", &Value[2]);
    printf("Address of Fourth Element = %d \n", &Value[3]);
    printf("Address of Fifth Element  = %d \n", &Value[4]);
}
```

**Output**

```
Address of First Element        = 6487552
Address of Second Element       = 6487556
Address of Third Element        = 6487560
Address of Fourth Element       = 6487564
Address of Fifth Element        = 6487568
```

## 8.2.5 Boundary Checking

In C, there is no check to see if the subscript/index used for an array exceeds the size of the array. Data entered with subscript exceeding the array size will simply be placed in memory outside array. This will never give an error message but will lead to unpredictable results which is more dangerous than the errors. Sometimes the computer may also hang. So it is the programmer's job to check that the subscript doesn't reach beyond array size.

**Example**

```
int regno[5] = {10, 11, 12, 13, 14};
printf("%d", regno[15]);
```

The above printf statement may lead to junk output or abnormal termination.

In order to avoid this boundary problem, it is a better practice to initialize the array without any index value.

```
i.e., int regno[ ] = {10, 11, 12, 13, 14};
```

## Basic Properties of an Array

- The data items (called elements) stored in the array should be of the same data type.
- The data items are stored contiguously in memory with subscript of the first item always 0 (zero).
- Each data item is accessed using the same name along with different subscript.
- The subscript or index of the array is always an integer.

**Program 2** To find sum and average of 5 elements in an array.

```
#include<stdio.h>
main()
{
      int i, a[5], sum=0;
      float ave

      for (i=0; i<5; i++)
      {
            printf("\n Enter the element %d:", i+1);
            scanf("%d", &a[i];
            sum=sum+a[i];
      }

      avg=sum/5;
      printf("The sum=%d\n", sum);
      printf("The average=%f\n", avg);
}
```

**Output**

```
Enter the element 1: 10
Enter the element 2: 08
Enter the element 3: 01
Enter the element 4: 02
Enter the element 5: 09
The sum = 30
The average = 6.0
```

**Program 3** To store student's marks of a subject

```
#include<stdio.h>
main()
{
      int i, marks[6];

      for(i=0; i < 6; i++)
      {
         printf("Enter the marks of student %d : ",i+1);
         scanf ("%d'', &marks[i]);
      }

      for (i = 0; i<6;i++)
         printf("\nThe marks of student %d is %d", i+1, marks[i]);
}
```

**Output**

```
Enter the marks of student 1 : 33
Enter the marks of student 2 : 22
Enter the marks of student 3 : 91
Enter the marks of student 4 : 29
Enter the marks of student 5 : 39
Enter the marks of student 6 : 48
The marks of student 1 is 33
The marks of student 2 is 22
The marks of student 3 is 91
The marks of student 4 is 29
The marks of student 5 is 39
The marks of student 6 is 48
```

**Program 4** To find the even, odd, positive, negative numbers out of 20 numbers entered from keyboard.

```
#include<stdio.h>
main()
{
    int a[20], i, neg=0, pos=0, odd=0, even=0;

    printf("Enter 20 elements of the array\n");

    for(i=0; i < 20; i++)
    {
       scanf("%d", &a[i]);
       a[i] < 0 ? (neg++) : (pos++); /*conditional operators*/
       a[i] % 2 ? (odd++) : (even++);
    }

    printf("\n The number of positive elements=%d", pos);
    printf("\n The number of negative elements=%d", neg);
    printf("\n The number of even numbers=%d", even);
    printf("\n The number of odd numbers=%d", odd);
}
```

**Program 5** To display the elements of two arrays two separate columns and add their corresponding elements. Display result of addition in the 3rd column.

```
main()
{
    int i, num1[ ]={10, 20, 30, 40, 50, 60} ;
    int num2[ ]={11, 22 , 33, 44, 55, 66} ;

    printf("Element of 1st Array \t\t Element of  2nd Array \t\t Addition\n") ;

    for(i=0;i<=5;i++)
       printf("\n\t%d \t\t + %d\t\t=%d", num1[i], num2[i], num1[i]+num2[i]);
}
```

**Output**

| Element of 1st Array | | Element of 2nd Array | | Addition |
|---|---|---|---|---|
| 10 | + | 11 | = | 21 |
| 20 | + | 22 | = | 42 |
| 30 | + | 33 | = | 66 |
| 40 | + | 44 | = | 84 |
| 50 | + | 55 | = | 105 |
| 60 | + | 66 | = | 126 |

## 8.3 Two Dimensional Arrays

An array consisting of two subscripts is known as two-dimensional array. These are often known as array of arrays.

**Definitions : Two Dimensional Array**

**Two Dimensional (2D) array** is a fixed-length, homogeneous data type, row and column-based data structure which is used to store similar data type element in a row and column-based structure.

A two-dimensional array is referred to as a **matrix** or a **table**. A matrix has two subscripts, one denotes the row and another denotes the column. In other words, a two-dimensional array is an **array of a one-dimensional arrays.**

In two dimensional arrays the array is divided into rows and columns. These are well suited to handle a table of data. A two-dimensional array has two subscripts; a three-dimensional array has three subscripts; and so on. Three is no limit to the number of dimensions a C array can have.

### 8.3.1 Declaration of Two Dimensional Array

Two dimensional arrays are declared same as one dimensional arrays except that a separate pair of square brackets [ ] are required for each subscript. A 2-dimensional array can thus be represented with two pairs of square brackets, a 3D array with three pairs and so on.

**Syntax**

In general, the syntax for a two dimensional array is:

```
datatype  arrayname[exp1][exp2];
```

Where **datatype** is any basic data type such as **int, float, char,** etc., and **exp1, exp2** are positive integer expressions that indicate the number of array elements associated with each subscript.

**Example**

A 2-D array called **name** of type **char,** 2 rows and 3 columns can be declared as:

```
char name[2][3];
```

This array will contain 2 × 3 (6) elements.

**Example**

```
int x[3][4];
```

Here, x is a two-dimensional (2d) array. This array will contain 3× 4 (12) elements. We can think the array as a table with 3 rows and each row has 4 columns. The first index value shows the number of the rows and second index value shows the number of the columns in the array.

| | Column 1 | Column 2 | Column 3 | Column 4 |
|---|---|---|---|---|
| **Row 1** | x [0] [0] | x[0] [1] | x[0] [2] | x[0] [3] |
| **Row 2** | x [1] [0] | x[1] [1] | x[1] [2] | x[1] [3] |
| **Row 3** | x [2] [0] | x[2] [1] | x[2] [2] | x[2] [3] |

## 8.3.2 Initializing Two Dimensional Array

Like the one-dimensional arrays, two-dimensional arrays may be initialized by following their declaration with a list of initial values enclosed in braces. The below are the examples of initializing two dimensional arrays.

```
int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[][3] = {{1, 3, 0}, {-1, 5, 9}};
int c[2][3] = {1, 3, 0, -1, 5, 9};
```

**Example**

```
int checker[4][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

It results in the following assignments:

```
checker[0][0] is equal to 1
checker[0][1] is equal to 2
checker[0][2] is equal to 3
checker[1][0] is equal to 4
checker[1][1] is equal to 5
checker[1][2] is equal to 6
checker[2][0] is equal to 7
checker[2][1] is equal to 8
checker[2][2] is equal to 9
checker[3][0] is equal to 10
checker[3][1] is equal to 11
checker[3][2] is equal to 12
```

The above array has 4 rows and 3 columns. The elements in the braces from left to right are stored in the memory also from left to right. The elements will be filled in the array in order, the first 3 elements from the left in the first row, the next 3 elements in the second row, and so on.

**Note:**

Note that the first subscript ranges from 0 to 3 and the seconds subscript ranges from 0 to 2. A point to remember here also is that array elements will be stored in contiguous locations in memory.

The above array checker can be thought of as an array of 4 elements, each of which is an array of 3 integers, and will appear as

| Row 0 | | | Row 1 | | | Row 2 | | | Row 3 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Example** **Two dimensional arrays can also be initialized by forming groups of initial values enclosed within braces. Consider the following initialization.**

```
            rows columns
              ↑    ↑
int checker [4] [3] = {
                        {1,2,3},      // Row 1
                        {4,5,6},      // Row 2
                        {7,8,9},      // Row 3
                        {10,11,12}    // Row 4
                      };
is exactly same as    int checker[4][3] = {1,2,3,4,5,6,7,8,9,10,11,12};
```

Like one-dimensional arrays, it is not compulsory to initialize all members of the two-dimensional arrays. If there are less row or column values than the actual size (i.e. subscript value) then the remaining elements are initialized to zero.

**Example**

```
int checker [4][3] =     {
                              {19,5,2},
                              {2,9,9},
                              {132,10,5},
                              {0,1,2}
                         };
```

Would only initialize the first 3 elements of each row of the matrix to the given values. The remaining values will be set to zero.

Note that the inner pairs of braces are responsible for the correct initialization. Also note that the initialization values must be separated by a comma-even when there are braces { } between them.

**Example** **It is very important to remember that while initializing an array it is necessary to mention the second (column) dimension, where as first dimension (row) is optional.**

```
int arr[2][3] = {10, 20,30,40,50,60};
```

is same as

```
int arr[][3] = {10, 20,30,40,50,60};
```

whereas

| | | |
|---|---|---|
| `int arr[2][ ] = {10, 20,30,40,50,60};`<br>`int arr[ ][ ] = {10, 20,30,40,50,60};` | } | illegal |

**Example** **Two dimensional array of characters.**

```
char name[2][10]={"Skyward","Publishers"};
```

It means that name is two dimensional character array. It can store two strings. Each string can contain maximum of 10 characters. So, first string is "Skyward" and second string is "Publishers".

```
printf("%s",name[0]);  prints "Skyward"

printf("%s",name[1]);  prints "Publishers"

printf("%c",name[0][0]); prints  'S' -> First Character of the First String

printf("%c",name[1][0]); prints  'P' -> First Character of the Second String
```

## 8.3.3 Accessing the Elements of Two Dimensional Arrays

An element in 2-dimensional array is accessed by using the subscripts. That is, row index and column index of the array. Let us see how a 2–D array can be accessed.

**Example** **Two dimensional array of integers.**

Let us consider the below two dimensional array. The array contains 3 rows and 4 columns.

```
int x[3][4] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
```

x[2][1] represents the element present in the third row and second column.

To access the first row and first column we can use x[0][0]

To access the first row and third column we can use x[0][2]

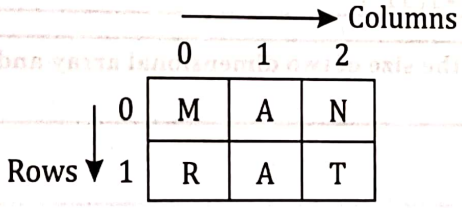
To access the second row and second column we can use x[1][1] and so on.

**Note**: In arrays, if the size of an array is N. Its index will be from 0 to N-1. Therefore, for row index 2 means row number is 2+1 = 3.

**Example**

```
char name[2][3]={"MAN","RAT"};
```

This array will contain 2 × 3 (6) characters and can be represented as:




The two subscripts 2 and 3 are numbered starting from 0. The first subscript ranges from 0 to 1 and second subscript ranges from 0 to 2 respectively.

The character 'N' is stored at row 0 and column 2. To access this element, the representation will be name[0][2];

Similarly to access the element 'T', the representation will be name[1][2];

To access the element 'R', the representation will be name[1][0];

To print first string, we can just mention name[0]

```
print("%s",name[0]);
```

To print second string, we can mention name[1]

```
printf("%s,name[1]);
```

**Example** **How to store user input data into Two Dimensional Array? How to print elements of Two Dimensional Array?**

Lets consider the below two dimensional array declaration.

int marks[3][2];

To store the elements entered by user we should two for loops, one of them is a nested loop. The outer loop runs from 0 to 2(first subscript -1) and the inner for loops runs from 0 to 1 (second subscript -1).

```
for(i=0;i<3;i++)
      for (j=0;j<2;j++)
         scanf("%d",&marks[i][j]);
```

When i=0 in outer for loop; The inner loop executes two times and reads the value for marks[0][0] and marks[0][1];

When i=1 in outer for loop; The inner loop executes two times and reads the value for marks[1][0] and marks[1][1];

When i=2 in outer for loop; The inner loop executes two times and reads the value for marks[2][0] and marks[2][1];

This way the the order in which user enters the elements would be stored in marks[0][0], marks[0][1], marks[1][0].marks[1][1], marks[2][0] and marks[2][1].

Similarly, we should use two for loops to print the elements of two dimensional array as shown below.

```
for(i=0;i<3;i++)
      for (j=0;j<2;j++)
          print("%d",marks[i][j]);
```

**Program 6** **Program to read the size of two dimensional array and elements from user and print the elements.**

```
#include<stdio.h>

void main()
{
      int marks[10][10], m, n,i,j;

      printf("\nEnter Number of Students:  ");
      scanf("%d", &m);
      printf("\nEnter Number of Subjects :  ");
      scanf("%d", &n);
      for(i=0; i<m; i++)
         for(j=0;j<n;j++)
            {
               printf("\nEnter Marks %d of Student %d : ",j+1,i+1);
               scanf("%d", &marks[i][j]);
            }
```

```
    printf("\n Marks Details : ");
    printf("\n*****************");
    for(i=0; i<m; i++)
       for(j=0;j<n;j++)
            printf("\n Marks %d of Student %d is %d",j+1,i+1,marks[i][j]);
}
```

**Output**

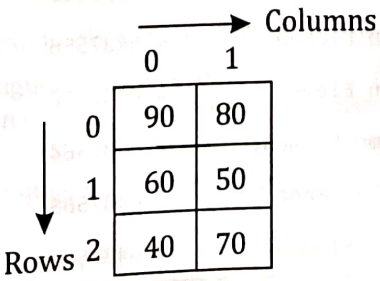
```
Enter Number of Students:  2
Enter Number of Subjects :  4
Enter Marks 1 of Student 1 : 99
Enter Marks 2 of Student 1 : 87
Enter Marks 3 of Student 1 : 82
Enter Marks 4 of Student 1 : 54
Enter Marks 1 of Student 2 : 76
Enter Marks 2 of Student 2 : 83
Enter Marks 3 of Student 2 : 69
Enter Marks 4 of Student 3 : 59
 Marks Details :
****************
 Marks 1 of Student 1 is 99
 Marks 2 of Student 1 is 87
 Marks 3 of Student 1 is 82
 Marks 4 of Student 1 is 54
 Marks 1 of Student 2 is 76
 Marks 2 of Student 2 is 83
 Marks 3 of Student 2 is 69
 Marks 4 of Student 2 is 59
```

### 8.3.4 Memory Representation of Two Dimensional Array

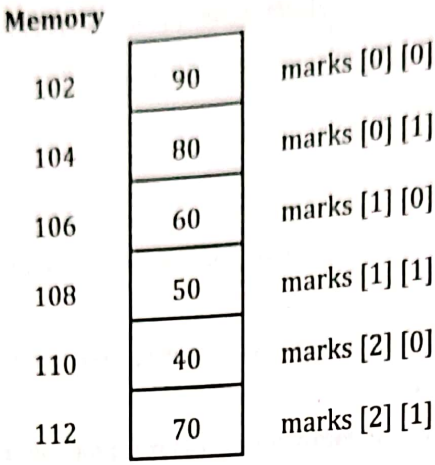
A two dimensional array of m rows and n column is represented in memory by a block of m*n sequential memory locations. The elements of two dimensional arrays are stored in the memory in terms of the row design, i.e. the first row of the array is stored in the memory then second and so on.

```
int marks [3][2] = {90, 80, 60, 50, 40, 70};
```

In this case, the memory map will look like



This diagrammatic arrangement is conceptually true. This is because, in memory there are no rows & columns. In memory, irrespective of dimensions the array elements are stored in one continuous chain. So the actual memory map will look like




It is clear from the output of the below program that elements of two dimensional array is stored in contiguous memory locations in row wise. We can observe the difference of 4 bytes between the addresses.

**Program 7** **Program to demonstrate the memory representation of two dimensional array.**

```
#include <stdio.h>
void main()
{
      int marks[3][2]={ 85, 98, 79, 88, 90,65};
      printf("Address of First Row First Column Element    = %d \n", &marks[0][0]);
      printf("Address of First Row Second Column Element   = %d \n", &marks[0][1]);
      printf("Address of Second Row First Column Element   = %d \n", &marks[1][0]);
      printf("Address of Second Row Second Column Element  = %d \n", &marks[1][1]);
      printf("Address of Third Row First Column Element    = %d \n", &marks[2][0]);
      printf("Address of Third Row Second Column Element   = %d \n", &marks[2][1]);
}
```

**Output:**

```
Address of First Row First Column Element       = 6487552
Address of First Row Second Column Element      = 6487556
Address of Second Row First Column Element      = 6487560
Address of Second Row Second Column Element     = 6487564
Address of Third Row First Column Element       = 6487568
Address of Third Row Second Column Element      = 6487572
```

### Understanding Memory Representation of 2-D array

Let us consider below two dimensional array. `int abc[4][5];`

The two dimensional conceptual major representation of abc[4][5] is as shown below.

2D array conceptual memory representation

Second subscript →

first subscript ↓

| | | | |
|---|---|---|---|
| abc [0][0] | abc [0][1] | abc [0][2] | abc [0][3] |
| abc [1][0] | abc [1][1] | abc [1][2] | abc [1][3] |
| abc [2][0] | abc [2][1] | abc [2][2] | abc [2][3] |
| abc [3][0] | abc [3][1] | abc [3][2] | abc [3][3] |
| abc [4][0] | abc [4][1] | abc [4][2] | abc [4][3] |

Here array is abc [4][5] which can be conceptually viewed as a matrix of 4 rows and 5 columns. Point to note here is that subscript starts with zero, which means abc[0][0] would be the first value of the array.

However the actual representation of this array in memory would be something like –

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| abc[0][1] | abc[0][2] | abc[0][3] | abc[1][0] | abc[1][1] | .... .... | abc[4][2] | abc[4][3] |
| 82206 | 82210 | 82214 | 82218 | 82222 | | 82274 | 82278 |

Memory locations for the array elements

Array is of integer type so each element would use 4 bytes that's the reason there is a difference of 4 in element's addresses.

Actual memory representation of a 2D array

## 8.4 Advantages and Disadvantages of Arrays

### Advantages of Arrays

- Arrays represent multiple data items of the same type using a single name.
- The elements can be accessed randomly by using the index number or subscript.
- Arrays allocate memory in contiguous memory locations for all its elements. Hence there is no chance of extra memory being allocated in case of arrays. This avoids memory overflow or shortage of memory in arrays.
- The other data structures like linked lists, stacks, queues, trees, graphs etc can be implemented using arrays.
- Two-dimensional arrays are used to represent matrices.

### Disadvantages of Arrays

- The number of elements to be stored in an array should be known in advance.
- An array is a static structure (which means the array is of fixed size). Once declared the size of the array cannot be modified. The memory which is allocated to it cannot be increased or decreased..
- Insertion and deletion are quite difficult in an array as the elements are stored in consecutive memory locations and the shifting operation is costly.
- Allocating more memory than the requirement leads to wastage of memory space and less allocation of memory also leads to a problem.

## 8.5 Applications of Arrays

### Applications of Arrays

- Arrays are used to Store List of values.
- Arrays are used to Perform Matrix Operations
- Arrays are used to implement Searching and Sorting Algorithms.
- Arrays are used to implement other Data Structures like Stacks and Queues.
- Arrays are used to implement CPU Scheduling Algorithms
- Arrays help to maintain large data under a single variable name. This avoid the confusion of using multiple variables.

**What is static and dynamic arrays?**

An array created at compile time by specifying size in the source code has a fixed size and cannot be modified at run time. The process of allocating memory at compile time is known as static memory allocation and the arrays that receives static memory allocation are called **static arrays.**

This method works fine as long as we know the array's exact size. If we don't know the exact size, we can use dynamic arrays instead.

In C, it is possible to allocate memory to arrays at run time. This feature is known as dynamic memory allocation and the arrays created at run time are called **dynamic arrays**. Dynamic arrays are created using pointer variables and memory management functions malloc(), calloc() and realloc().

**What is a derived data type? Give an example.**

Fundamental data types are basic built-in data types of C programming language. There are three fundamental data types in C programming. They are an integer data type, floating data type and character data type.

Data types that are derived from fundamental data types are called derived data types. Derived data types don't create a new data type but,instead they add some functionality to the basic data types. The derived data type can be used to represent a single value or multiple values. Arrays and Pointers are examples of derived data type.

**Example**: An array is derived data type because it contains the similar types of fundamental data types and acts as a new data type for C.

## 8.6 Sample Program

**Program 8** **Write a C program to find second largest element in a one dimensional array.**

```
#include <stdio.h>

int second_largest_element(int arr[], int size)
{
    int first_large, second_large, i;

    if (size < 2)
    {
       printf("Error: Array has less than two elements !!!");
       return;
    }

    for (i = 0; i < size; i++)
    {
       if (first_large < arr[i])
       {
         second_large = first_large;
         first_large = arr[i];
       }
      else if (arr[i] != first_large && second_large < arr[i] &&  second_large < first_large)
           second_large = arr[i];
    }
    return second_large;
}

void main()
{
   int size, index;

   printf("Enter the size of array: ");
   scanf("%d", &size);

   int arr[size];

   printf("\nEnter the array elements: ");
   for (index = 0; index < size; index++)
       scanf("%d", &arr[index]);

   printf("Second largest element: %d ", second_largest_element(arr, size));
}
```

**Output:**

```
Enter the size of array: 10
Enter the array elements: 80 40 45 78 54 36 29 99 28 12
Second largest element: 80
```

## 8.7 Review Questions

1. Define an array.
2. Write the general syntax of declaring an array.
3. Write the general syntax of declaring and initialising an array.
4. Declare one integer and one float array of 10 elements.
5. How arrays are classified?
6. How an elements of an array are stored in memory?
7. What is two dimensional array?
8. Declare one two-dimensional and one three-dimensional array.
9. Write the general syntax of declaring two dimensional array.
10. Give an example for derived data type.
11. What is an array? How it is declared and initialized?
12. Explain how to access single dimensional array elements?
13. Explain different ways of initializing one dimensional arrays?
14. Explain the memory representation of one-dimensional-array.
15. Write a program to store marks and register number of a 10 students using arrays.
16. Explain two dimensional arrays.
17. Explain declaration and initialization of two dimensional arrays.
18. Explain memory representation of 2-D arrays.
19. Write a program to add and subtract two matrices.
20. Write a program to multiply two matrices.
21. Explain the need of arrays?
22. What are the advantages and disadvantages of arrays?
23. Write the applications of arrays.
24. What is static and dynamic arrays?