

Chapter 9: Arrays

What are we studying in this chapter?

- ◆ Using arrays
- ◆ Using arrays with functions (Inter-function communication)
- ◆ Linear search, Binary search, Bubble sort, selection sort
- ◆ Two dimensional arrays, Multi-dimensional arrays
- ◆ Programming examples.

9.1 Concept of arrays (Why arrays?)

So far in previous chapters, we have learnt:

- ◆ how to read the data
- ◆ how to process the data
- ◆ how to display the data.

Now, let us consider the program to read marks of 3 students and print them.

Example 9.1: Program to read marks of 3 students

```
#include <stdio.h>

void main()
{
    int m1, m2, m3;

    printf("Enter marks of 3 students\n");
    scanf("%d %d %d",&m1, &m2, &m3);

    printf("%d %d %d", m1, m2, m3);
}
```

The above program works fine. However, if we want to read marks of say 20 or more students, then we have to declare 20 or more variables, process them and print them. This is a very tedious task due to following reasons:

- ◆ Difficult to declare more number of variables
- ◆ Difficult to write program to read data into more number of variables and difficult to print the data stored in more number of variables
- ◆ Very difficult to process large amount of data
- ◆ Length of program increases as the number of variables increases
- ◆ Consumes too much time to type, modify and correct
- ◆ It is not a good programming style

9.2 Arrays

Now, “What is the solution for this problem? How to process large amount of data? Is there any solution for this?” Yes, there should be some solution (Every problem will have a solution). *All the disadvantages can be overcome using a very powerful data structure called arrays* along with looping constructs that we have studied in previous chapters.

Now, let us see “What is an array? How array can be used and manipulated?”

Definition: An array is a special and very powerful data structure in C language. An array is a collection of similar data items. All elements of the array share a common name. Each element in the array can be accessed by the subscript (or index). Array is used to store, process and print large amount of data using a single variable.

Ex 1: Set of integers, set of characters, set of students, set of pens etc. are examples of various arrays.

Ex 2: Marks of all students in a class is *an array of marks*

Ex 3: Student names in a class is *an array of student names*

The pictorial representation of an array of 5 integers, an array of 5 floating point numbers and an array of 5 characters is shown below:

A[0]	10	B[0]	5.5	C[0]	'A'
A[1]	20	B[1]	6.5	C[1]	'B'
A[2]	30	B[2]	7.5	C[2]	'C'
A[3]	40	B[3]	8.5	C[3]	'D'
A[4]	50	B[4]	9.5	C[4]	'E'
Array of 5 integers		Array of 5 floats		Array of 5 characters	

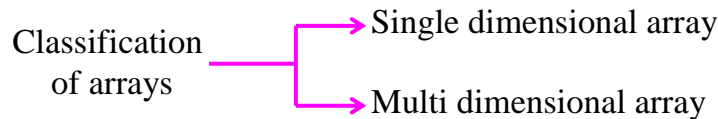
Now, the question is “Is it possible to have a set consisting of different kinds of items?” Definitely no. Even in arrays, it is not possible to group items of different data types. For example,

80	90.18	“Rama”	32	100
a[0]	a[1]	a[2]	a[3]	a[4]

is invalid way of storing the elements in an array. This is because, it is a collection of *integers*, *floats* and *string* which are all dissimilar data types.

9.2 Classification of arrays

Now, let us see “*How arrays are classified?*” The arrays are classified as shown below:



9.2.1 Single-dimensional arrays

Now, let us see “*What is a single-dimensional array?*” A *single-dimensional* array (also called one-dimensional array) is a *linear list* consisting of related data items of same type. In memory, all the data items are stored in contiguous memory locations one after the other. *For example*, a single dimensional array consisting of 5 integer elements is shown below:

10	15	20	25	30
A[0]	A[1]	A[2]	A[3]	A[4]

Now, the question is “*How to access these elements?*” Since an array is identified by a common name any element in the array can be accessed by specifying the subscript(or an index). An index or subscript gives the position of an element in the array. For example, in the above array:

- ◆ 0th item 10 can be accessed by specifying A[0]
- ◆ 1st item 20 can be accessed by specifying A[1]
- ◆ 2nd item 30 can be accessed by specifying A[2]
- ◆ 3rd item 40 can be accessed by specifying A[3]
- ◆ 4th item 50 can be accessed by specifying A[4]

Now, can you tell “*What are the basic properties of arrays?*” An array has the following basic properties:

- ◆ The array elements should be of the same data type.
- ◆ The data items are stored contiguously in memory.
- ◆ The subscript of first item is always zero.
- ◆ Each data item is accessed using the name of the array, but, with different subscripts.
- ◆ The index of the array is always an integer.

```

Ex 1: a[1.5]      // Error Index has to be integer.
Ex 2: a[1]       // OK Index is an integer.
Ex 3: a['5']     // OK Index is an integer.
                // '5' has an ASCII value which is integer.
  
```

9.4 Arrays

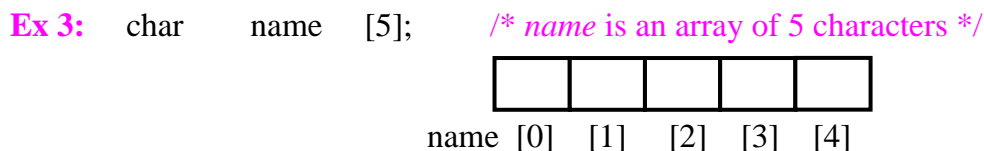
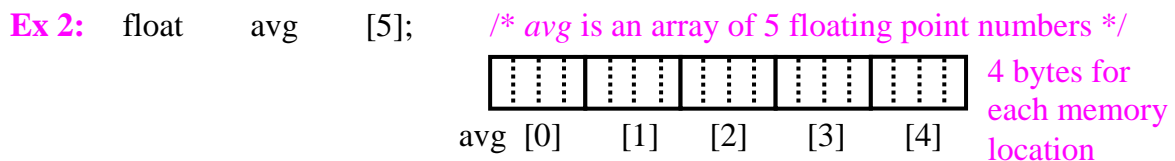
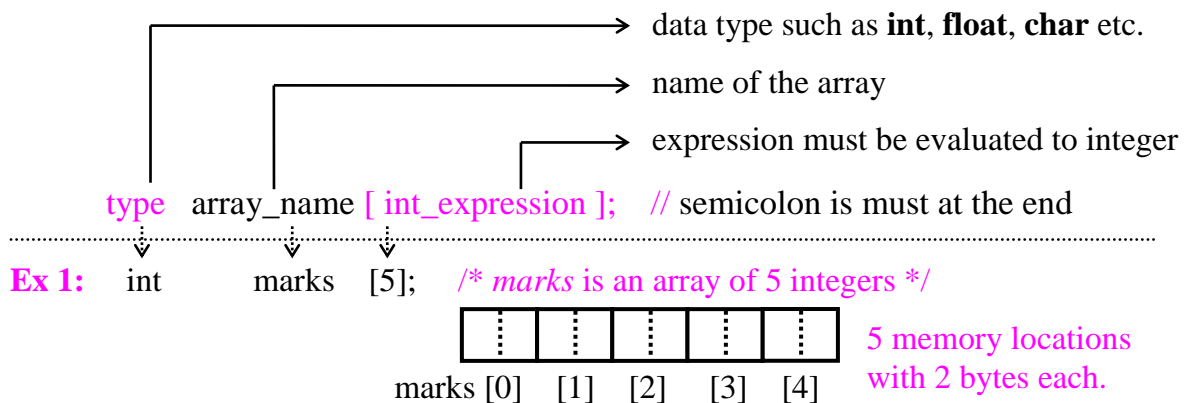
9.2.2 Declaration of single dimensional arrays

Now, let us see “How to declare and define a single dimensional array?” DEC-2015/JAN-2016

As we declare and define variables before they are used in a program, an array also must be declared and defined before it is used. The declaration and definition informs the compiler about the:

- ◆ Type of each element of the array
- ◆ Name of the array
- ◆ Number of elements (i.e., size of the array)

The compiler uses this size to reserve the appropriate number of memory locations so that data can be stored, accessed and manipulated when the program is executed. A single dimensional array can be declared and defined using the following syntax:



Now, let us see “What are the rules to be followed while declaring an array?” The rules to be followed are shown below:

- ◆ The expression must be enclosed within brackets
 - Ex 1:** `int marks[5];` *// OK : 5 is enclosed within brackets*
 - Ex 2:** `int marks(5);` *// Error: 5 is enclosed within parentheses*

- Ex 3:** `int marks{5};` // Error: 5 is enclosed within braces
- ◆ The expression can be an integer constant

Ex 1: `int marks[5];` // OK

Ex 2: `int marks[5.5];` // Error: expression must be an integer
 - ◆ The expression can be an integer expression without variables.

Ex 1: `int marks[3+2];` // OK

Ex 2: `int marks[3+a];` // Error: expression must have constants
// but, *a* is a variable
 - ◆ The integer expression enclosed within brackets must end with semicolon.

Ex 1: `int marks[5];` // OK

Ex 2: `int marks[5]` // Error: Must end with semicolon
 - ◆ The integer expression must be evaluated to an integer greater than or equal 1.

Ex 1: `int a[5];` // OK : expression is evaluated to a value ≥ 1

Ex 2: `int a[2+3];` // OK : expression is evaluated to a value ≥ 1

Ex 3: `int a[0];` // Error: Must be evaluated to a value ≥ 1

Ex 4: `int a[2-2];` // Error: Must be evaluated to a value ≥ 1
 - ◆ The expression can contain named constants. This is another way of declaring and defining arrays. For example, consider the statements:

`const int ARRAY_SIZE = 5;` (1) /* Declaring named constant */

`int a[ARRAY_SIZE];` (2)

 - 1) In the first statement, observe that the identifier `ARRAY_SIZE` is declared as constant with the value 5. The identifier `ARRAY_SIZE` is called named constant. This is because, we are identifying the constant 5 by the name `ARRAY_SIZE` and hence the name named constant.
 - 2) In second statement, we use the value of the named constant `ARRAY_SIZE` to declare an array and specify its size.
 - ◆ The expression can contain symbolic names. For example,

`#define MAX 3` // MAX is a defined constant

`int marks [2 + MAX];` // OK

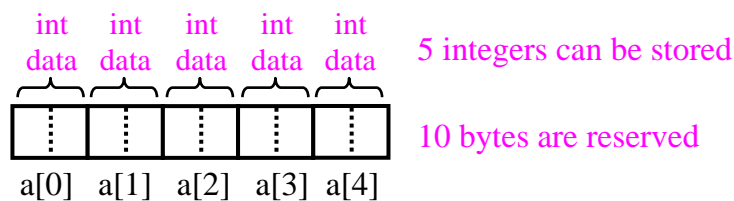
Note: The amount of memory to be allocated is decided during compilation but not during execution. So, size has to be specified and it must be evaluated to an integer value greater than or equal to 1. When we declare and define an array, its size must be known.

9.6 📖 Arrays

Example 9.2: Explain the instruction: `int a[5];`

It is an instruction given to the compiler to allocate memory for 5 integer values so that all the integer values can be accessed using array name *a* followed by the subscript.

- ◆ Since the size specified is 5 and by assuming `sizeof(int)` as 2 bytes, totally 10 bytes are reserved.
- ◆ Every 2 bytes are used to store an integer value. The pictorial representation is shown below:

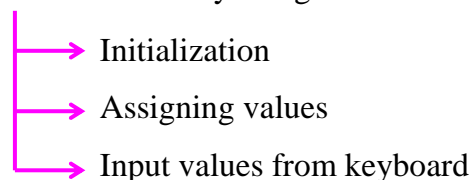


Observe the following points with respect to array *a*:

- ◆ 5 integer values can be stored
- ◆ Array always starts from index 0.
- ◆ The 5 elements can be accessed by using: `a[0]`, `a[1]`, `a[2]`, `a[3]` and `a[4]`
- ◆ For the above array, the following are invalid: `a[-1]`, `a[5]`, `a[6]` etc.
- ◆ Memory is allocated but not initialized

9.3 Storing values in Arrays

In previous section, we have seen that declaration and definition reserves the space for the elements in the array. But, array elements are not initialized and hence no values are stored. Now, let us see “How to store values into array?” It is very simple. The values can be stored in array using three methods:



9.3.1 Initialization of single dimensional arrays

Now, we shall answer the question “What is initialization? How arrays can be initialized?”

Definition: Assigning or providing the required values to a variable before processing is called *initialization*. As we initialize a variable to some initial value (for example,

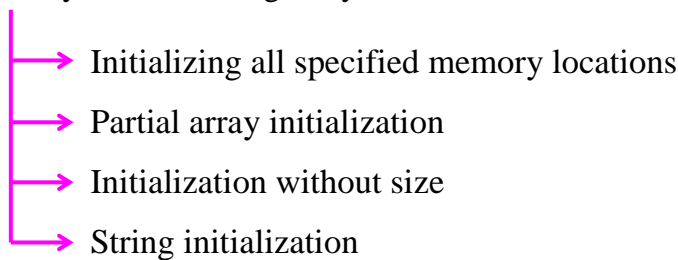
sum = 0 before the for-loop), we can initialize the individual elements of an array. Array elements can be initialized at the time of declaration. The syntax is shown below:

```
type array_name[expression] = {v1, v2, .....,vn};
```

where

- ◆ *type* can be a data type such as **int**, **float**, **char** etc.
- ◆ *array_name* is the name of the array.
- ◆ *expression* within brackets should be evaluated to a positive integer only.
- ◆ v1, v2, ... vn are the values and should be enclosed within ‘{’ and ‘}’ separated by commas. The value v1 is stored in 0th location; v2 is stored in 1st location and so on.

The various ways of initializing arrays are shown below:



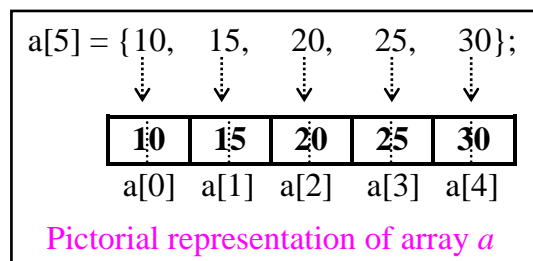
9.3.2 Initializing all specified memory locations

Now, let us see “*How to initialize all the specified memory locations?*” 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 integer, character, float etc. The values are copied into array in the order specified in the declaration.

Example 9.3: Consider integer initialization shown below:

```
int a[5] = { 10, 15, 20, 25, 30};
```

During compilation, memory for 5 integers is reserved by the compiler for the variable *a*. If the size of integer is 2 bytes, 5*2=10 bytes will be reserved for the variable *a*. All 5 integers are initialized as shown in figure:



Other examples,

```
int a[5] = { 10, 15, 20, 25, 30, 40, 50};
```

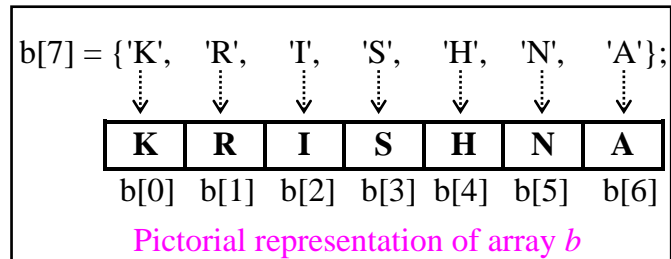
//Error: Number of initial values
//are more than the size of array

9.8 Arrays

Example 9.4: Consider character initialization shown below:

```
char b[7] = {'K', 'R', 'T', 'S', 'H', 'N', 'A'};
```

During compilation, 7 contiguous memory locations are reserved by the compiler for the variable *b*. Since the size of each character is 1 byte, 7 bytes will be allocated for the variable *b*. All 7 locations are initialized as shown in figure:



Other examples,

```
char b[5] = {'M', 'O', 'N', 'I', 'K', 'A'}; //error: Number of initial values  
//are more than the size of array
```

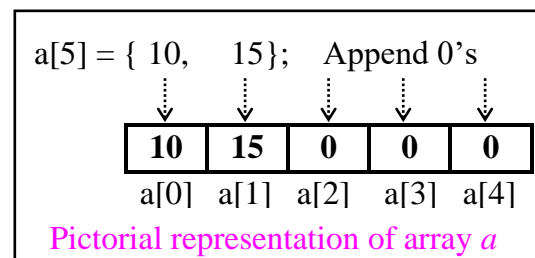
Note: It is not possible to initialize array elements in the middle of array without initializing the predecessor locations.

9.3.3 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 are initialized in the order from 0th location. The remaining locations will be initialized to zero automatically. For example, consider the partial initialization shown below:

```
int a[5] = { 10, 15};
```

Even though compiler reserves space for 5 integers using this declaration, the compiler initializes first two locations with 10 and 15. The next set of memory locations are automatically initialized to 0's by the compiler as shown in figure:

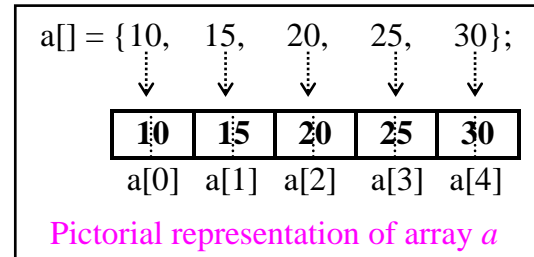


9.3.3 Initialization without size

Consider the declaration along with initialization shown below:

```
int a[] = {10, 15, 20, 25, 30};  
↓  
size not specified
```


In this declaration, even though we have not specified number of elements to be used in array *b*, the array size will be set to the total number of initial values specified. The compiler will calculate the size of the array based on the number of initial values. Since number of elements is 5, totally $5 * 2 = 10$ bytes are reserved where 2 is the sizeof integer. The array *a* is initialized as shown in figure:

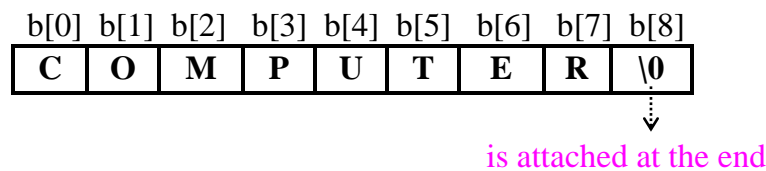


9.3.4 Array initialization with a string

The sequence of characters enclosed within two double quotes is a string. The string always ends with NULL character denoted by `'\0'`. Consider the declaration with string initialization:

```
char b[] = "COMPUTER";
```

The array *b* is initialized as shown below.



Observe that size of the array is 9 bytes (i.e., string length + 1 byte for null character).

Ex 1: `char b[9] = "COMPUTER";`

8 + 1 byte for `'\0'` = 9 bytes

Array size is greater than or equal string length + 1. So, it is correct

Ex 2: `char b[8] = "COMPUTER";`

8 + 1 byte for `'\0'` = 9 bytes

Array size is not greater or equal string length + 1. So, declaration is wrong

Note: For string initialization, usually programmers prefer the following declaration:

```
char b[] = "COMPUTER";
```

9.10 Arrays

Now, the question is “Is it not possible to specify the size of the array when the program is executed?”

No, It is not possible. The size of the array must be known during compilation. For example, consider the following program segment:

```
int array_size;           /* Line: 1 */
int a[array_size];        /* Line: 2 */
                           |
                           |
printf("Enter the size of the array\n"); /* Line: 3 */
scanf("%d", &array_size); /* Line: 4 */
                           |
                           v
```

Note: It is a variable. The size must be a constant. So, it is not possible to enter the size of the array when the program is being executed. Size of the array must be known during compilation. So, when line 2 is compiled, the compiler gives an error.

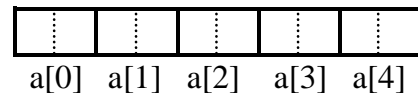
9.3.5 Assigning values to array

Now, let us see “How to assign the values to array without initialization?” Answer is simple – use assignment operator. We can assign values to individual elements of the array using the assignment operator. For example, consider the declaration along with pictorial representation:

Declaration

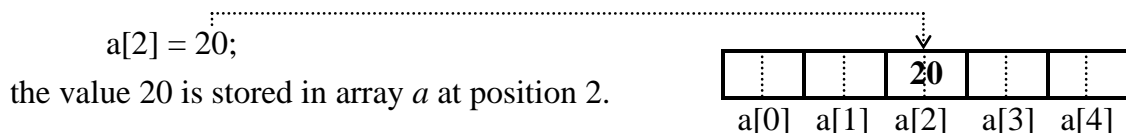
```
int a[5];
```

Memory representation

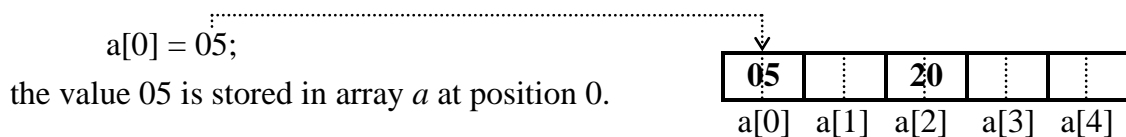


The elements 20, 05, 40, 50 and 60 can be inserted into array at positions 2, 0, 4, 1 and 3 as shown below:

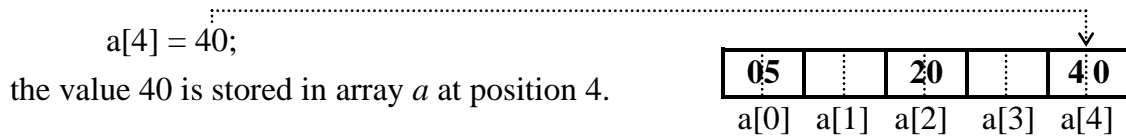
After executing the instruction:



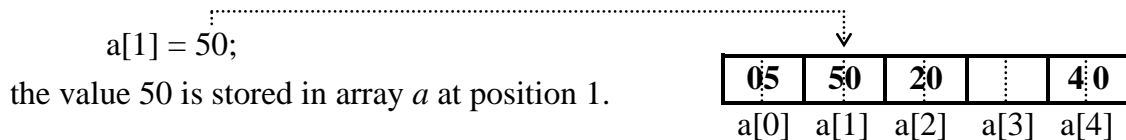
After executing the instruction:



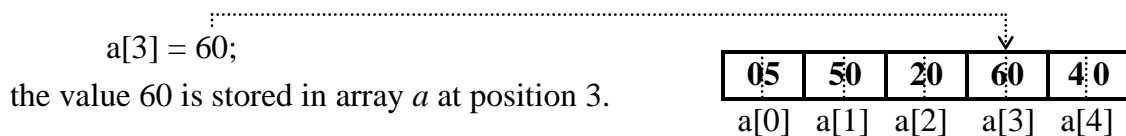
After executing the instruction:



After executing the instruction:



After executing the instruction:



Now, we know how to assign individual elements using the assignment operator. The question is “Is it possible to assign one array to other array?” No. It is not possible. Even if the type of elements, size and number of elements matches, it is not possible to assign one array to other array. For example, consider the following two arrays:

```
int a[5] = { 10, 20, 30, 40, 50 };
int b[5];

b = a;           /* Error: Cannot assign one array to other array */
```

The question is “How to copy all elements of array *a* into array *b*”? It is very easy. Copy individual elements of *a* into *b* as shown below:

```

b [0] = a [0] ;      /* b[0] is assigned the value of a[0] */
b [1] = a [1] ;      /* b[1] is assigned the value of a[1] */
b [2] = a [2] ;      /* b[2] is assigned the value of a[2] */
b [3] = a [3] ;      /* b[3] is assigned the value of a[3] */
b [4] = a [4] ;      /* b[4] is assigned the value of a[4] */
    ↓           ↓
In general, b [i] = a [i];
             ↙       ↘
             i = 0 to 4 (we are forced use for loop)
```

<pre>for (i = 0; i <= 4; i++) { b[i] = a[i]; }</pre>	or	<pre>for (i = 0; i < 5; i++) { b[i] = a[i]; }</pre>
--	----	---

```
int a[5];
```

Using `a[0]` through `a[4]` we can access 5 integers.

Note: In general, Using $a[0]$ through $a[n-1]$ we can access n data items.

```
scanf("%d", &a[0]);
```

In general, **scanf("%d",&a[i])** where $i = 0, 1, 2, 3, \dots, n-1$. So, in C language, if we want to read n data items from the keyboard, the following statement can be used:

Using operator <=
Using operator <

Using operator <

Similarly to display n data items stored in the array, replace *scanf()* by *printf()* statement as shown below:

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

Now, let us write a program to read n elements from the keyboard and to display n elements on to the screen.

Example 9.5: Program to read n items from the keyboard and to display n elements on the monitor

#include <stdio.h>	TRACING
void main() {	Execution starts from here
int n, a[10], i;	
printf ("Enter the no. of items");	Input
scanf ("%d",&n);	Enter the no. of items
.....	5
printf ("Enter n elements");	Enter n elements
for (i = 0; i < n; i++)	i = 0 1 2 3 4
{	a[0] a[1] a[2] a[3] a[4]
scanf ("%d",&a[i]);	10 20 30 40 50
}	
.....	
printf ("The N elements are");	The N elements are
for (i = 0; i < n; i++)	i = 0 1 2 3 4
{	a[0] a[1] a[2] a[3] a[4]
printf ("%d ", a[i]);	10 20 30 40 50
}	
}	

9.4 Arrays and Functions

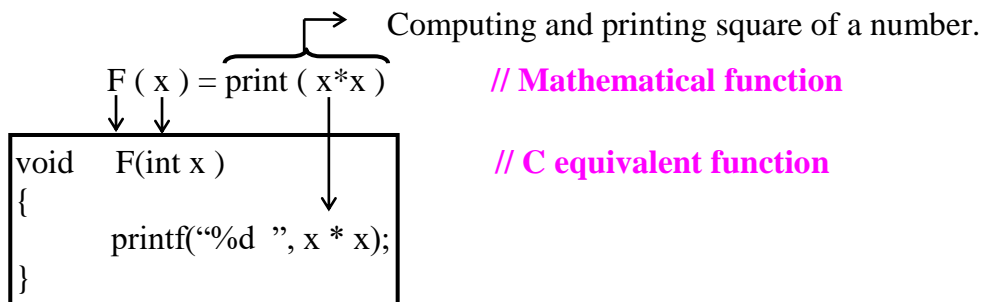
Now, let us see "How to pass arrays to functions?" The arrays can be passed to functions using two methods:

- Passing individual elements of the array
- Passing the whole array

9.14 Arrays

9.4.1 Passing individual elements of array

Now, let us see “How array elements can be passed to functions?” All array elements can be passed as individual elements to a function like any other variable. For example, let us write a program to pass array elements to a function which print square of elements of the array. We can compute and print square of a given number using the following mathematical function.



For better readability and understanding purpose, we can replace the function name `F` by the name `print_square()` and the function can be written as shown below:

Example 9.6: Program to print square of a given number `x`

not returning any value.

```
void print_square (int x)
{
    printf("%d ", x * x);
}
```

Note: The above function accept x as input parameter, computes and print square of x and does not return any value. So, the return type of the function must be **void**

Now, let us see, how the above function can be used to print the square of array elements. For example, consider the following array:


5	2	3	6	4
<code>a[0]</code>	<code>a[1]</code>	<code>a[2]</code>	<code>a[3]</code>	<code>a[4]</code>

The square of each element in the array can be printed using the above `print_square()` function as shown below:

```

print_square (a [0] );
print_square (a [1] );
print_square (a [2] );
print_square (a [3] );
print_square (a [4] );

```



In general, `print_square (a [i]);` where $i = 0$ to $n-1$

So, the statement: `print_square (a[i])` has to be repeatedly executed for all the values of i ranging from 0 to $n-1$ and can be written using for loop as shown below:

```

for ( i = 0; i < n; i++)
{
    print_square ( a[i] );
}

```

The complete C program is shown below:

Example 9.7: C Program to print square of given array elements

```
#include <stdio.h>
```

```
/* Include: Example 9.6: Function to accept an integer value & print its square */
```

```
void main()
```

```
{
```

```
    int n, a[10], i;
```

```
    printf("Enter the no. of items");
```

```
    scanf("%d",&n);
```

```
    printf("Enter n elements");
```

```
    for (i = 0; i < n; i++)
```

```
    {
```

```
        scanf("%d",&a[i]);
```

```
    }
```

```
    printf("Squares of n items:");
```

```
    for (i = 0; i < n; i++)
```

```
    {
```

```
        print_square( a[i] );
```

```
    }
```

```
}
```

Execution starts from here

Enter the no. of items

5

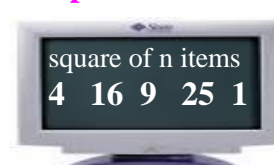
Enter n elements

i = 0 1 2 3 4

a[0] [1] [2] [3] [4]

2 4 3 5 1

Output



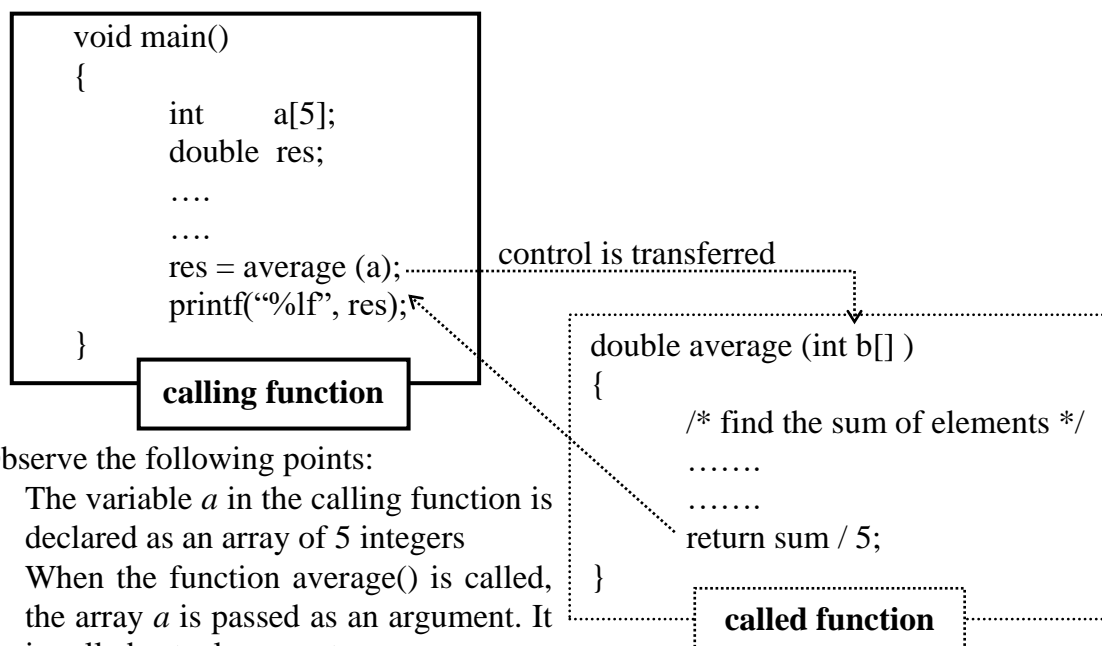
9.16 Arrays

9.4.2 Passing the whole array

Now, let us see “How whole array can be passed to a function?” Suppose, we want to pass whole array to a function. In such situation, we must follow the two rules:

- ◆ The function must be called by passing only the name of the array.
- ◆ In the function definition, the parameter must be declared as an array of the same type as that of actual parameter. There is no need to specify the size of the array.

For example, consider the following program segment:



Observe the following points:

- ◆ The variable *a* in the calling function is declared as an array of 5 integers
- ◆ When the function `average()` is called, the array *a* is passed as an argument. It is called actual parameter
- ◆ Control is transferred to the function `average()`
- ◆ In the called function, the variable *b* is declared as an array. Observe that type of array *a* in the calling function and type of array *b* in the called function remains same.
- ◆ The function computes the average of 5 numbers and the result is returned to the calling function.
- ◆ The value returned from called function is copied into variable *res* and the result is displayed on the screen.

Now, the question is “Will change in an array which is passed as a parameter, will affect the actual parameter?” The answer is yes. Only in case of arrays, the parameters are not passed by value. The parameter passing is similar to pass by reference. Suppose, an array *a* is passed to a function, and array *b* is the corresponding parameter in called function. Any change in array *b* in the called function, will have corresponding changes in array *a* in the calling function. So, whether we like it or not, parameter passing using arrays is similar to pass by reference. For example, consider the following program:

Example 9.8: C Program to print square of given array elements

```
#include <stdio.h>
```

```
void reverse (int b[])
```

```
{
    b[0] = 50, b[1] = 40, b[2] = 30, b[3] = 20, b[4] = 10;
}
```

```
void main()
```

```
{
    int a[5] = {10, 20, 30, 40, 50};
    int i;
```

```
reverse(a);
```

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

```
}
```

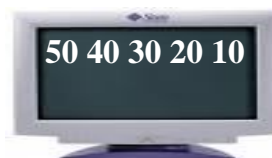
①

10	20	30	40	50
a[0]	a[1]	a[2]	a[3]	a[4]

②

50	40	30	20	10
a[0]	a[1]	a[2]	a[3]	a[4]

③



Tracing the above program, observe the following points:

- ① The compiler reserve space for 5 integer values and will be initialized with 10, 20, 30, 40 and 50.
- ② Control is transferred to function reverse(). Since the parameter *b* is an array, any change in array *b* will have the corresponding changes in array *a*. So, array *a* elements now contain 50, 40, 30, 20, 10.
- ③ The contents of array *a* are displayed on the screen.

9.18 Arrays

9.5 Programming examples

Important: Given any problem, most important thing is how to write a program. Here, I have given some simple techniques in a logical and systematic manner. Let us follow these techniques in designing and developing a program:

Step 1: **Identify parameters to function:** If array a is given which consists of n elements, then parameters are:

int a[], int n

Step 2: **Return type:** If we find average value and return this value, its type is **double**. So, return type is:

double

Step 3: **Design the function body.** It is explained by taking various examples. The variables other than parameters should be declared in the beginning of the function body.

Note: All variables other than the parameters must be declared inside the body of the function.

9.5.1 Addition of N elements in an array and to find the average

Design: Now, let us design a function to find the sum of array elements and find the average.

Step 1: Identify parameters to function: We have to find the average of elements stored in an array a consisting of n floating point elements. So, given input is array a and number of elements n

parameters are : **double a[], int n**

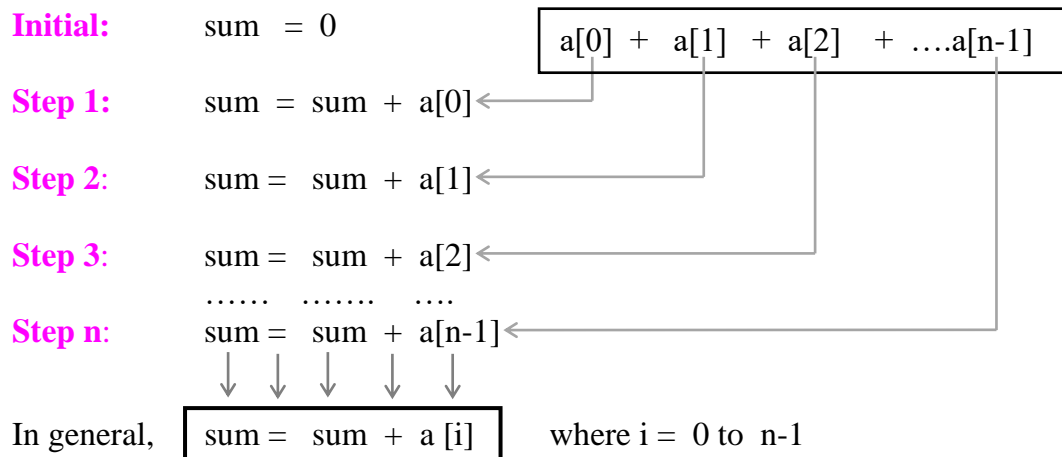
Step 2: Return type : We find average value which is of type **double**. So,

return type : **double**

Step 3: Designing function body: The array a consisting of n elements can be added as shown below:

$$\text{sum} = a[0] + a[1] + a[2] + \dots a[n-1]$$

Before adding any of the terms in the above series, sum will be zero. So, we initialize sum to 0 and start adding successive terms one by one as shown below:



The above statement has to be repeatedly executed for each value of i ranging from 0 to $n-1$ and can be written using for loop as shown below:

```
sum = 0;
for (i = 0; i < n; i++)
{
    sum = sum + a[i];
}
```

After computing sum of all elements, we can return the average by dividing sum by n using the following statement:

```
return sum/n;           /* Return the average */
```

Note: All variables other than parameters should be declared as local variables. The complete function is shown below:

Example 9.9: C function returning average value of array elements

```
double find_average (double a[], int n)
{
    int    i;
    double sum;

    sum = 0;
    for (i = 0; i < n; i++)
    {
        sum = sum + a[i];
    }

    return sum/n;           /* Return the average */
}
```

9.20 Arrays

The above function can be invoked as shown below:

Example 9.10: C Program to read n elements and to find the average

```
#include <stdio.h>
```

```
/* Include: Example 9.9: Function find average of  $n$  items stored in array  $a$  */
```

void main() { int n, i; double a[10], average; printf ("Enter the number of elements\n"); scanf ("%d",&n); /* Read N elements */ printf ("Enter N elements"); for (i = 0; i < n; i++) { scanf ("%lf",&a[i]); } average = find_average(a, n); printf ("Average = %lf\n", average); }	TRACING Execution starts from here Input Enter the no. of elements 5 Enter N elements a[0] a[1] a[2] a[3] a[4] 10 20 30 40 50 average = (10 +20 +30 + 40 +50)/5 Average = 30.0
--	---

9.5.2 To find sum of +ve, -ve numbers and their average

Design: Now, let us design a function to find the sum of array elements and find the average. We also print sum of positive numbers and sum of negative numbers:

Step 1: Identify parameters to function: We have to find the average of elements stored in an array a consisting of n floating point elements. So, given input is array a and number of elements n

parameters are : **double a[], int n**

Step 2: Return type: We find average value which is of type **double**. So,

return type : **double**

Step 3: Designing function body: Given an array a consisting of n elements can be added as shown in previous section. The statement:

sum = sum + a[i]

in the previous example should be changed to

psum = psum + a[i] if (a[i] >= 0)

and

nsum = nsum + a[i] otherwise

The above two statements can be written as shown below:

if (a[i] >= 0) psum = psum + a[i]; else nsum = nsum + a[i];	}	should be repeated for i = 0 to n-1
--	---	-------------------------------------

with initial value of *psum* and *nsum* as zero. Now, the partial code can be written as shown below:

```

psum = nsum = 0;
for ( i = 0; i < n; i++ )
{
    if (a[i] >= 0)
        psum = psum + a[i];
    else
        nsum = nsum + a[i];
}
    
```

Sum of positive numbers and negative numbers can be displayed using the following statements:

```

printf("Sum of positive nos = %lf\n", psum);
printf("Sum of negative nos = %lf\n", nsum);
    
```

The average value can be returned using the statement:

```

return (psum + nsum) / n;
    
```

Note: All variables other than parameters should be declared as local variables:

- ◆ Variables of type **double** are *psum*, *nsum*
- ◆ Variables of type **int** are *i*

Now, the complete C function can be written as shown below:

9.22 Arrays

Example 9.11: C function to print sum of +ve, -ve numbers and return average value

double find_sum_average (double a[], int n)	a[0] a[1] a[2] a[3] a[4], n
{	1 -2 3 -4 5 5
double psum, nsum;	
int i;	
.....	
/* Initialize positive and negative sum */	<u>Computations</u>
psum = nsum = 0;	psum = nsum = 0
.....	
/* Add positive and negative elements */	
for (i = 0; i < n; i++)	i = 0 1 2 3 4
{	a[0] a[1] a[2] a[3] a[4]
if (a[i] >= 0)	1 -2 3 -4 5
psum = psum + a[i];	psum = 1 + 3 + 5 = 9
else	
nsum = nsum + a[i];	nsum = -2 + -4 = -6
}	
.....	
printf ("Sum of positive nos = %f\n", psum);	<u>Output</u>
printf ("Sum of negative nos = %f\n", nsum);	Sum of positive nos = 9
	Sum of negative nos = -6
return (psum + nsum) / n;	return (9 + (-6)) / 5 = 0.6
}	

The above function can be invoked as shown below:

Example 9.12: C Program that calls function to print sum of positive numbers and negative numbers and return average of those numbers

```
#include <stdio.h>
```

```
/* Include: Example 9.11: Function to find sum of +ve, -ve numbers and average */
```

```
void main()
```

```
{
```

```
    int n, i;
```

```
    double a[10], average;
```

```
    printf ("Enter the number of items\n");
```

```
    scanf ("%d",&n);
```

```
    .....
```

TRACING

```
Enter the number items
```

```
5
```

<pre> /* Read n elements */ printf("Enter N elements"); for (i = 0; i < n; i++) { scanf("%lf",&a[i]); } average = find_sum_average (a , n) ; printf ("Average of all nos = %lf", average); } </pre>	<pre> Enter N elements i = 0 1 2 3 4 a[0] a[1] a[2] a[3] a[4] 1 -2 3 -4 5 </pre> <hr/> <pre> average = 0.6 </pre> <hr/> <p>Output</p> <pre> Average of all nos = 0.6 </pre>
--	--

9.5.3 Largest of N numbers

Now, let us design a function to find largest of n elements stored in array a .

Step 1: Identify parameters to function: We have to find the position of largest element in an array a consisting of n integer elements. So,

parameters are : **int a[], int n**

Step 2: Return type: We are returning position of largest element which is of type **int**. So,

return type : **int**

Step 3: Designing function body: Let $a[0], a[1], \dots$ etc. be an array of n elements. Let us assume the first element itself is largest and hence its position is 0. So,

```
pos = 0;    /* Assume first item is big whose position is 0*/
```

In the next step, subsequent elements $a[1], a[2], \dots, a[n-1]$ should be compared with $a[pos]$. In general, we compare $a[i]$ with $a[pos]$ as shown below:

```
if ( a[i] > a[pos] )
```

If above condition is true, $a[i]$ is larger and its position is i . This will be the position of largest element and hence we store it in pos . Now, the code can be written as:

```
if ( a[i] > a[pos] ) pos = i; /* Find the position of largest item */
```

9.24 Arrays

The above statement has to be repeated for $i = 1, 2, 3, \dots, n-1$. Now, the code takes the following form with initial value of *pos* as zero:

```
pos = 0;
for ( i = 1; i < n; i++)
{
    if ( a[i] > big ) pos = i;
}
```

Now, *pos* contains the position of largest item and hence its value can be returned using the statement:

```
return pos;    /* Return the position of largest */
```

Note: All variables other than parameters should be declared as local variables i.e., the variables *pos* and *i* should be declared inside the function with type **int**

The complete function can be written as shown below:

Example 9.13: C function that returns the position of the largest item

```
int largest ( int a[], int n)
{
    int    i, pos;
    .....
    /* Initial largest position */
    pos = 0;
    .....
    /* Computing the position of largest*/
    for ( i = 1; i < n; i++)
    {
        if ( a[i] > a[pos] ) pos = i;
    }
    .....
    return pos;    /* return the position of largest item */
}
```

The above function can be used to find the largest element and its position as shown in the following program:

Example 9.14: C Program to print largest and its position

```
#include <stdio.h>

/* Include: Example 9.13: Function to return the position of largest item */

void main()
{
    int    a[10], n, i, pos;
    .....
    printf("Enter the number of elements\n");
    scanf("%d",&n);
    .....
    printf("Enter %d elements\n", n);
    for ( i = 0; i < n; i++)
    {
        scanf("%d",&a[i]);
    }
    .....
    pos = largest ( a, n );
    .....
    printf("Largest = %d\n", a[pos]);
    printf("Position = %d\n", pos+1);
}

```

Enter the number of elements
5
Enter 5 elements
i = 0 1 2 3 4
a[0] [1] [2] [3] [4]
35 50 55 10 45
pos = 2
Largest = 55
Position = 3

9.5.4 Generation of Fibonacci numbers using arrays

First, let us see “What are Fibonacci numbers?”

Definition: The **Fibonacci numbers** are a series of numbers such that each number is the sum of the previous two numbers except the first and second number. The Fibonacci numbers are shown below:

0, 1, 1, 2, 3, 5, 8, 13,

Design: Now, let us design a function to generate Fibonacci numbers. Let us store all the generated Fibonacci numbers in the array *a*.

Step 1: Identify parameters to function: We are required to generate *n* Fibonacci numbers and store it in array *a*. So, parameters are : **int a[], int n**

Step 2: Return type: We are not returning any value. So, return type : **void**

Step 3: Designing function body: The first two Fibonacci numbers are 0 and 1 and so, they can be stored in locations *a[0]* and *a[1]*. The subsequent Fibonacci numbers can be generated by adding the previous two Fibonacci numbers as shown below:

9.26 Arrays

1 st Fibonacci number	:	a[0]	=	0	initial values
2 nd Fibonacci number	:	a[1]	=	1	
3 rd Fibonacci number	:	a[2]	=	a[1] + a[0] = 1	
4 th Fibonacci number	:	a[3]	=	a[2] + a[1] = 2	
5 th Fibonacci number	:	a[4]	=	a[3] + a[2] = 3	
.....	:	=	
.....	:	=	
n th Fibonacci number	:	a[n-1]	=	a[n-2] + a[n-3]	

In general, ith fibonacci no: $a[i] = a[i-1] + a[i-2]$
for $i = 2, 3, 4, 5, \dots, n-1$

Thus, the partial code can be written as:

```
a[0] = 0, a[1] = 1;
for (i = 2; i < n; i++)
{
    a[i] = a[i-1] + a[i-2];
}
```

Note: All variables other than parameters such as index variable i should be declared as local variable. The function to generate n Fibonacci series is shown below:

Example 9.15: C function to generate n Fibonacci numbers

<pre>void Fibonacci (int a[], int n) { int i; a[0] = 0; a[1] = 1; for (i = 2; i < n; i++) { a[i] = a[i-1] + a[i-2]; } }</pre>	<div style="border-left: 1px dashed black; padding-left: 10px;"> <div style="border-bottom: 1px dotted black; margin-bottom: 5px;">a[0] = 0</div> <div style="border-bottom: 1px dotted black; margin-bottom: 5px;">a[1] = 1</div> <div style="margin-bottom: 5px;">a[0] [1] [2] [3] [4]</div> <div>0 1 1 2 3</div> </div>
--	--

The above function can be called to generate n Fibonacci series as shown below:

Example 9.16: C Program to print Fibonacci numbers

```
#include <stdio.h>
```

```
/* Include: Example 9.15: Function to generate Fibonacci numbers */
```

```
void main()
```

```
{
```

```
    int    n, i, a[100];
```

```
    printf("Enter the value for n\n");
```

```
    scanf("%d",&n);
```

```
    /* Compute Fibonacci numbers */
```

```
    Fibonacci (a, n);
```

```
    printf("The Fibonacci numbers are\n");
```

```
    for ( i = 0; i < n; i++)
```

```
    {
```

```
        printf("%d\n",a[i]);
```

```
    }
```

```
}
```

TRACING

```
Enter the value for n
```

```
5
```

```
a[0] [1] [2] [3] [4]
```

```
0   1   1   2   3
```

Output

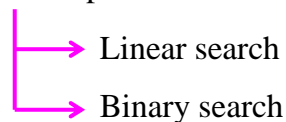
```
The Fibonacci numbers are
```

```
0 1 1 2 3
```

9.6 Searching

First, let us see “*What is searching? What are the various searching techniques?*”

Definition: More often we will be working with large amount of data. It may be necessary to determine whether a particular item is present in the large amount of data. *This process of finding a particular item in the large amount of data is called searching.* The two important and simple searching techniques are shown below:



9.6.1 Linear search (Sequential search)

Now, let us see “*What is linear search?*”

Definition: A *linear search* also called *sequential search* is a simple searching technique. In this technique we search for a given *key* item in the list in linear order (sequential order) i.e., one after the other from first element to last element or vice versa. The search may be successful or unsuccessful. If key item is present, we say search is successful, otherwise, search is unsuccessful. For example, consider the following array:

9.28 Arrays

10	15	20	25	30
a[0]	a[1]	a[2]	a[3]	a[4]

- ♦ **Successful search:** If search key is 25, it is present in the above list and we return its position 3 indicating “Successful search”.
- ♦ **Unsuccessful search:** If search key is 50, it is not present in the above list and we return -1 indicating “Unsuccessful search”

Design: Now, let us see how to search for an item.

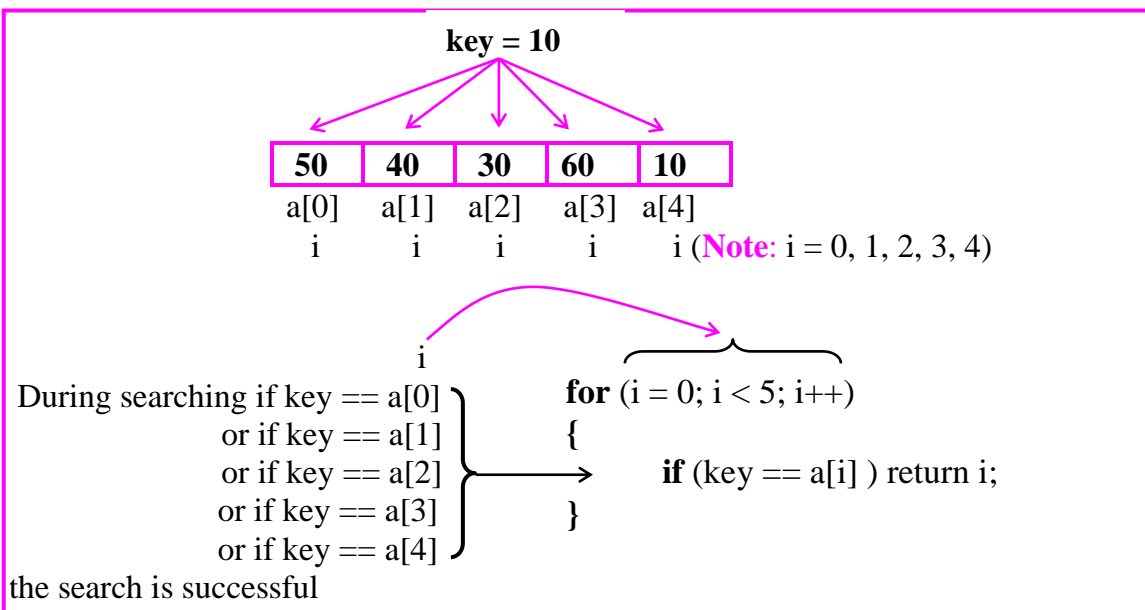
Step 1: Identify parameters to function: We have to search for *key* item in an array *a* consisting of *n* elements. So, input must be *key*, array *a* and *n*. So,

parameters are : **int** key, **int** a[], **int** n

Step 2: Return type: We are returning position of *key* item if found, otherwise, we return -1. Note that the position is integer and -1 is also integer. So,

return type : **int**

Step 3: Designing function body: Let a[0], a[1],.....etc. be an array of *n* elements. Let us take an example. Assume 10 is the *key* item to be searched in the list shown in figure. The list consist of items 50, 40, 30, 60 and 10 in order. In the worst scenario, *key* item may have to be compared with all the elements in the array. Observe from following figure that, *key* item 10 has to be compared with a[0], a[1], a[2], a[3] and a[4] as shown below:



But, once the value of i is greater than or equal to 5, it is an indication that *item* is not present and we return -1. The code for this can be written as:

```
return -1;    /*Unsuccessful search */
```

Note: The terminal condition in the *for* loop i.e., $i < 5$ can be replaced by $i < n$ for a general case. Now, the code can be written as:

```
for (i = 0; i < n; i++)
{
    if (key == a[i] ) return i;    /* Successful search */
}

return - 1;    /* Unsuccessful search */
```

Note: All variables other than parameters should be declared as local variables i.e., the variable i which is of type **int** must be declared inside the function definition.

The C function to search for a *key* item is shown below:

Example 9.17: C function to implement linear search.

```
int linear ( int key, int a[], int n)
{
    int i;
    .....
    for (i = 0; i < n; i++)
    {
        if (key == a[i] ) return i;    /* Successful search */
    }
    .....
    return - 1;    /* Unsuccessful search */
}
```

The C function that calls above function is shown below:

Example 9.18: C Program to implement linear search

```
#include <stdio.h>
```

```
/* Include: Example 9.17: Function to implement linear search */
```

9.30 Arrays

	<u>TRACING1</u>	<u>TRACING2</u>
void main() {		
int n, key, a[20], i, pos;		
.....		
printf ("Enter the value of n\n");	Enter value of n	Enter value of n
scanf ("%d",&n);	5	5
.....		
printf ("Enter n values\n");	Enter n values	Enter n values
for (i = 0; i < n; i++) scanf ("%d",&a[i]);	10 20 50 40 30	10 20 50 40 20
.....		
printf ("Enter the item to be searched\n");	Enter item to search	Enter item to srch
scanf ("%d",&key);	30	60
.....		
<i>/* Search for an element */</i>	<u>Output</u>	<u>Output</u>
pos = linear (key, a , n);	pos = 4	pos = -1
.....		
if (pos == -1)		
printf ("Item not found\n");		Item not found
else		
printf ("Item found\n");	Item found	
}		

Advantages of linear search

- ◆ Very simple approach
- ◆ Works well for small arrays
- ◆ Used to search when the elements are not sorted

Disadvantages of linear search

- ◆ Less efficient if the array size is large
- ◆ If the elements are already sorted, linear search is not efficient.

9.6.2 Binary search

To overcome the disadvantages of *linear search*, we use *binary search*. Now, let us see “*What is binary search? What is the concept used in binary search?*”

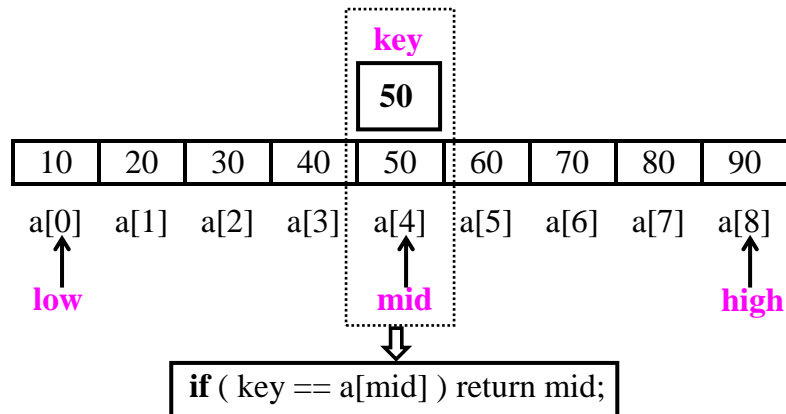
Definition: A *binary search* is a simple and very efficient searching technique which can be applied if the items to be compared are either in ascending order or descending order.

The general idea used in binary search is similar to the way we search for the telephone number of a person in the telephone directory. Obviously, we do not use *linear search*. Instead, we open the book from the middle and the *name* is compared with the element at the middle of the book. If the name is found, the corresponding

```
mid = ( low + high ) / 2
```

9.32 Arrays

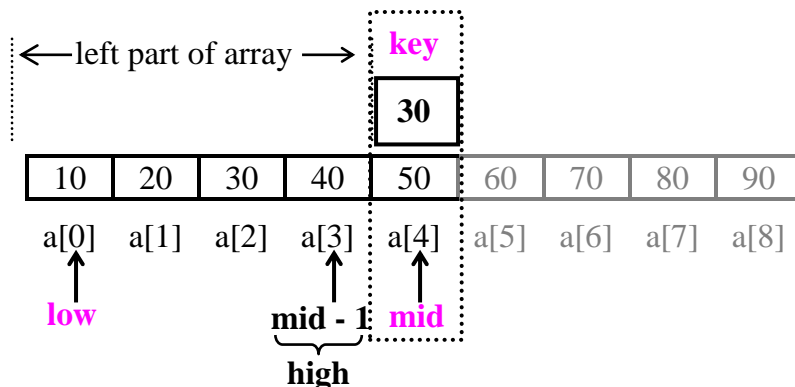
The *key* to be searched is compared with middle element. The pictorial representation and equivalent code can be written as shown below:



After executing the statement, if *key* is same as $a[mid]$, we return the position of *key*. If *key* is not same as $a[mid]$, it may be present either in the left part of the array or right part of the array and leads to following 2 cases:

Case 1: key towards left of mid : If *key* is less than the middle element, the left part of array has to be compared from low to $mid-1$, as shown in figure below:

i.e., low to $high$

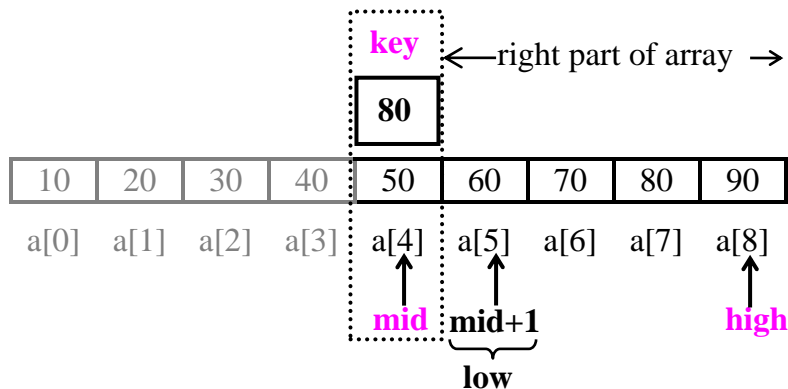


Note that **position of low has not been changed**. But, position of **$high$ is changed to $mid-1$** . The equivalent code can be written as:

```
if ( key < a[mid] ) high = mid - 1;
```

Case 2: key towards right of mid : If *key* is greater than the middle element, the right part of array has to be compared from $mid + 1$ to $high$ as shown in figure below:

i.e., low to $high$



Note that *position of high has not been changed*. But, position of *low* is changed to *mid+1*. The equivalent code can be written as:

```
if ( key > a[mid] ) low = mid + 1;
```

Note that entire step 4 has to be repeatedly executed as long as *low* is less than or equal to *high*. The code for this can be written as shown below:

```
while ( low <= high )
{
    mid = ( low + high ) / 2;           /* Find the mid point */
    .....
    if ( key == a[mid] ) return mid;    /* Item found */
    .....
    if ( key < a[mid] )
        high = mid - 1;               /* To search left part */
    .....
    if ( key > a[mid] )
        low = mid + 1;                /* To search right part */
}
```

Finally, when the value of *low* exceeds the value of *high* indicates that *key* is not present in the array and return -1 using the following statement:

```
return -1;           /* Search unsuccessful */
```

Note: All variables other than parameters should be declared as local variables i.e., the variables *low*, *high* and *mid* should be declared as **int** data type in function.

The complete C function can be written as shown below:

9.34 Arrays

Example 9.19: C function returning position of item found. Otherwise return -1

```
int binary_search ( int key, int a[], int n)
{
    int    low, high, mid;

    low = 0;                               /* Initialization */
    high = n-1;

    .....
    while ( low <= high )
    {
        mid = ( low + high ) / 2;          /* Find the mid point */
        .....
        if ( key == a[mid] ) return mid;    /* Item found */
        .....
        if ( key < a[mid] ) high = mid - 1; /* To search left part */
        .....
        if ( key > a[mid] ) low = mid + 1;  /* To search right part */
    }
    .....
    return -1;    /* Unsuccessful search */
}
```

The C program that uses the above function is shown below:

Example 9.20: C program search for an item using binary search

```
#include <stdio.h>

/* Include: Example 9.19: Function binary search */
void main()
{
    int  n, a[10];          /* Variables associated with array */
    int  key;               /* Element to be searched */
    int  position;          /* Contains position of key if found:
                           -1 if not found */

    printf ("Enter the number of elements\n");
    scanf ("%d", &n);

    Enter no. of items
    5
```

<pre> printf ("Input %d elements\n", n); for (i = 0; i < n; i++) scanf ("%d", &a[i]); printf ("Enter the element to be searched\n"); scanf ("%d", &key); </pre>	<p>Elements in the list</p> <p>10 20 30 40 50</p> <p>a [0] [1] [2] [3] [4]</p> <p>Input1 Input 2</p> <p>55 40</p>
<pre> pos = binary_search (key, a, n); </pre>	<p>pos = -1 pos = 3</p>
<pre> if (pos == -1) printf ("Item not found \n"); else printf ("Item found at pos: %d\n", position); } </pre>	<p>Output1</p> <p>Item not found</p> <p>Output2</p> <p>Item found at pos: 3</p>

Advantages of binary search

- ◆ Simple technique
- ◆ Very efficient searching technique

Disadvantages of binary search

- ◆ The list of elements to be searched should be sorted.
- ◆ It is necessary to obtain the middle element which is possible only if the elements are stored in the array. If the elements are stored in linked list, this method cannot be used.

9.7 Sorting techniques

First, we shall see “*What is sorting?*”

Definition: More often programmers will be working with large amount of data and it may be necessary to arrange them in ascending or descending order. This process of arranging the given elements so that they are in ascending order or descending order is called *sorting*. For example, consider the unsorted elements:

10, 50, 25, 20, 15

After sorting them in ascending order, we get the following list:

10, 15, 20, 25, 50

After sorting them in descending order, we get the following list:

50, 25, 20, 15, 10

9.7.1 Bubble Sort

9.36 Arrays

Before writing the algorithm or the program let us know the answer for “*What is the concept used in bubble sort (Why it is also called sinking sort)?*”

Design: Once we know what is sorting, the next question is “*How to sort the elements using bubble sort?*”

Step 1: Identify parameters to function: Given an array a consisting of n elements we have to sort them in ascending or descending order. So,

parameters are : **int a[], int n**

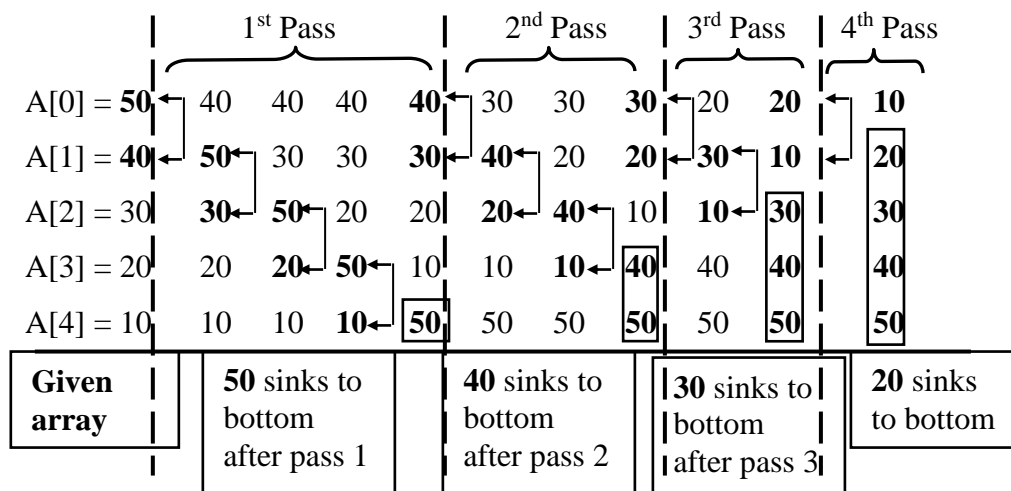
Step 2: Return type: Our intention is only to sort the numbers. Hence, we are not returning any value. So,

return type : **void**

Step 3: Designing body of the function: This is the simplest and easiest sorting technique. In this technique, the two successive items $A[i]$ and $A[i+1]$ are exchanged whenever $A[i] \geq A[i+1]$. For example, consider the elements shown below:

50, 40, 30, 20, 10

The elements can be sorted as shown below:



- ◆ In the first pass 50 is compared with 40 and they are exchanged since 50 is greater than 40.
- ◆ Next 50 is compared with 30 and they are exchanged since 50 is greater than 30. If we proceed in the same manner, at the end of the first pass the largest item occupies the last position.

- ◆ On each successive pass, the items with the next largest value will be moved to the bottom and thus elements are arranged in ascending order.

Note: Observe that after each pass, the larger values sink to the bottom of the array and hence it is called *sinking sort*. The following figure below shows the output of each pass.

Original Array	Out put of each pass			
	1 st	2 nd	3 rd	4 th
A[0] = 50	40	30	20	10
A[1] = 40	30	20	10	20
A[2] = 30	20	10	30	30
A[3] = 20	10	40	40	40
A[4] = 10	50	50	50	50

Note: Observe that at the end of each pass smaller values gradually “bubble” their way upward to the top (like air bubbles moving to surface of water) and hence called *bubble sort*.

Now we concentrate on the designing aspects of this sorting technique. The comparisons that are performed in each pass are shown below:

Passes →	Pass-1	Pass-2	Pass 3	Pass 4
	0 – 1 1 – 2 2 – 3 3 – 4	0 – 1 1 – 2 2 – 3	0 – 1 1 – 2	0 – 1
Comparing	↑ i=0 to 3	↑ i=0 to 2	↑ i=0 to 1	↑ i=0 to 0
	5-2	5-3	5-4	5-5
	5-(1+1)	5-(2+1)	5-(3+1)	5-(4+1)
	n- (j+1)	n -(j+1)	n-(j+1)	n-(j+1)

- ◆ In general we can say $i = 0$ to $n-(j+1)$ or $i = 0$ to $n - j - 1$.
- ◆ Here, $j = 1$ to 4 represent pass number. In general $j = 1$ to $n-1$.
- ◆ So, the partial code can be written as:

```

for j = 1 to n-1
    for i = 0 to n-j-1
        if ( A[i] > A[i +1] )
            exchange ( A[i] , A[i+1] )
    
```

9.38 Arrays

```
        end if
    end if
end if
```

Now, the complete bubble sort algorithm can be written as shown below:

Example 9.21: Algorithm for bubble sort

Algorithm BubbleSort(*a*[], *n*)

```
    for j ← 1 to n – 1 do           // Perform n-1 passes
        for i ← 0 to n-j-1 do       // To compare items in each pass
            if ( a[i] > a[i+1] ) // If out of order exchange
                temp ← a[i]
                a[i] ← a[i+1]
                a[i+1] ← temp
            end if
        end for
    end for
```

Note: All variables other than parameters should be declared as local variables i.e., the variables *i*, *j* and *temp* should be declared as **int** data type in function.

The C equivalent for the above algorithm can be written as shown below:

Example 9.22: C function to sort the array elements in ascending order

```
void bubble_sort ( int  a[], int  n )
{
    int    j;           /* Represent pass number */
    int    i;           /* To access elements in each pass */
    int    temp;        /* Temporary variable used to exchange*/

    for ( j = 1; j < n; j++)      /* Number of passes required: n-1 */
    {
        for ( i = 0; i < n - j; i++) /* To access elements in each pass */
        {
            if ( a[i] >= a[i+1] ) /* Exchange if out of order */
            {
                temp = a[i];
                a[i] = a[i+1];
            }
        }
    }
}
```

```

        a[i+1] = temp;
    }
}
}

```

The C program that uses the above function to sort the elements is shown below:

Example 9.23: C program to read n numbers, sort them using bubble sort

```
#include <stdio.h>
```

```
/* Include: Example 9.22: bubble sort */
```

```
void main()
{
```

```

    int    a[20];           /* Array of integers to be sorted */
    int    n;               /* Number of elements in array a and b */
    int    i;               /* Index used to access elements in a and b */

```

Input

printf ("Enter the number of items"); scanf ("%d",&n);	Enter the number of items 5
printf ("Enter the items to sort \n"); for (i = 0; i < n; i++) { scanf ("%d",&a[i]); }	Enter the items to search i = 0 1 2 3 4 a[0] [1] [2] [3] [4] 40 50 30 10 20
bubble_sort (a, n);	10 20 30 40 50 a[0] [1] [2] [3] [4]
printf ("The sorted elements are"); for (i = 0; i < n; i++) { printf ("%d\n",a[i]); }	The sorted elements are 10 20 30 40 50

Advantages of bubble sort

- ♦ Very simple and easy to program

9.40 Arrays

- ◆ Straight forward approach

Disadvantages of bubble sort

- ◆ It runs slowly and hence it is not efficient. More efficient sorting techniques are present
- ◆ Even if the elements are sorted, $n-1$ passes are required to sort.

2.2.1 Selection sort (Straight selection sort or pushdown sort)

Before writing the algorithm, let us know the answer for “*What is the concept used in selection sort? Design and analyze the same*”

Procedure: As the name indicates, we first find the smallest item in the list and we exchange it with the first item. Obtain the second smallest in the list and exchange it with the second element and so on. Finally, all the items will be arranged in ascending order. Since, the next least item is selected and exchanged appropriately so that elements are finally sorted, this technique is called *Selection sort*. Let us see, how the elements **45 20 40 5 15** can be sorted using selection sort.

Given items	After pass 1	After pass 2	After pass 3	After pass 4
A[0] = 45 ←	5	5	5	5
A[1] = 20	20 ←	15	15	15
A[2] = 40	40	40 ←	20	20
A[3] = 5 ←	45	45	45 ←	40
A[4] = 15	15 ←	20 ←	40 ←	45
1 st smallest is 5. Exchange it with 1 st item	2 nd smallest is 15. Exchange it with 2 nd item	3 rd smallest is 20. Exchange it with 3 rd item	4 th smallest is 40. Exchange it with 4 th item	All elements are sorted

Design: The smallest element from i^{th} position onwards can be obtained using the following code.

```
pos ← i           where i = 0, 1, 2, 3
for j ← i+1 to n-1
    if ( a[j] < a[pos] ) pos ← j
end for
```

After finding the position of the smallest number, it should be exchanged with i^{th} position. The equivalent statements are shown below:


```
temp ← a[pos]
a[pos] ← a[i]
a[i] ← temp
```

Note: The value of i range from 0 to 3
i.e., $i = 0$ to 3
 $i = 0$ to $5-2$

$i = 0$ to $n - 2$

The above statements have to be repeatedly executed for $i = 0$ to $n - 2$. Now, the selection sort algorithm is shown below:

Example 2.1: Algorithm for selection sort

Algorithm SelectionSort($a[], n$)

//Purpose: Sort the given elements using selection sort

// Inputs :

// n – the number of items present in the array
// a – the items to be sorted are present in the array

//Output:

// a – contains the sorted list

for $i \leftarrow 0$ **to** $n-2$ **do**

pos $\leftarrow i$;

//Assume ith element as smallest

for $j \leftarrow i + 1$ **to** $n-1$ **do**

// Find the position of the smallest item

if ($a[j] < a[pos]$) pos $\leftarrow j$;

end for

temp $\leftarrow a[pos]$

//Exchange ith item with smallest element

$a[pos] \leftarrow a[i]$

$a[i] \leftarrow temp$

end for

End of algorithm selection sort

9.8 Evaluate polynomial

A polynomial can be evaluated using following two simple techniques:

- ◆ By adding all the terms in polynomial

9.42 Arrays

- ♦ Horner's method

9.8.1 By adding all the terms of a polynomial

Design: Now, let us see “How to evaluate a polynomial in a straight forward manner?” The general polynomial equation is given by

$$a_0 x^0 + a_1 x^1 + a_2 x^2 + \dots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1} + a_n x^n$$

Step 1: Identify parameters to function: Given an array a consisting of $n+1$ elements ranging from $a[0]$ to $a[n]$ of type **double**, and value of x which may be **double**, we have to evaluate the polynomial. So,

parameters are : **double a[], int n, double x**

Step 2: Return type: After evaluating the polynomial, let us return the result of polynomial which is of type **double**. So,

return type : **double**

Step 3: Designing body of the function: Here, we solve the polynomial by adding all the terms of polynomial using straight forward technique. From the above polynomial equation it is observed that the general term is given by

$$\sum a_i x^i \text{ for } i = 0, 1, 2, 3, \dots, n$$

The C equivalent code to find the sum of all the terms is given by

```
sum = 0;
for (i = 0; i <= n; i++)
{
    sum = sum + a[i] * pow(x, i);
}
```

Note: All variables other than parameters should be declared as local variables i.e., the variables sum which is of type **double** and i which is of type **int** should be declared inside the function.

The C function to evaluate the above polynomial is shown below:

Example 9.24: C function to evaluate the polynomial

```
double evaluate ( double a[], int n, double x )
{
```

```

    double sum;
    int i;

    sum = 0;
    for (i = 0; i <= n; i++)
    {
        sum = sum + a[i] * pow(x, i);
    }

    return sum;
}

```

The C program to evaluate the polynomial and print the result as shown below:

Example 9.25: C Program to read the co-efficients and evaluate the polynomial

```

#include <stdio.h>
#include <math.h>

/* Include: Example 9.24: function to evaluate the polynomial */

```

```

void main()
{

```

```

    int n, i;
    double a[30], x, res;

```

TRACING

```

    printf("Enter the value of n\n");
    scanf("%d",&n);

```

Enter the value of n
5

```

    printf("Enter n+1 values\n");
    for (i = 0; i <= n; i++)
    {
        scanf("%lf",&a[i]);
    }

```

Enter n+1 values
i = 0 1 2 3 4 5
a[0] [1] [2] [3] [4] [5]
6 5 4 3 2 1

```

    printf("Enter the value for x\n");
    scanf("%lf",&x);

```

Enter the value for x
1

```

    res = evaluate (a, n, x);

```

res = 21

```

    printf("Result = %lf\n",res);
}

```

Result = 21

Note: To justify that it is a simple technique, but not efficient method, consider the polynomial

$$10 + 5x + 4x^2 + 3x^3 + 8x^4 + 9x^5$$

9.44 Arrays

Evaluation of this polynomial is carried out by substituting the value for x in the above polynomial and adding each term. The above polynomial can be written as

$$10 + \underbrace{5*x}_1 + \underbrace{4*x*x}_2 + \underbrace{3*x*x*x}_3 + \underbrace{8*x*x*x*x}_4 + \underbrace{9*x*x*x*x*x}_5$$

No. of multiplications = 1 + 2 + 3 + 4 + 5 = 15
No. of additions = 5

Even though it is a simple technique, observe that 15 multiplications and 5 additions are used during evaluation of the above polynomial. In general, if there are n terms, the total number of multiplications is given by $\frac{n(n+1)}{2}$ and total number of additions will be n . So, this involves too many multiplications and additions

Advantages

- ◆ It is a straight forward and simple technique
- ◆ Readable and understandable easily

Disadvantages

- ◆ Numerous arithmetic operations such as addition and multiplication have to be performed. So, it is not efficient

This disadvantage can be eliminated using Horner's method. The number of computations to be carried out can be reduced and still the evaluation of polynomial can be carried out in much easier and efficient method.

9.8.2 Horner's method of solving polynomial equation

It is one of the most preferred methods to evaluate a polynomial because of its simple approach and lesser arithmetic operations.

Design: Now, let us see “How to evaluate a polynomial using Horner's method?” The general polynomial equation is given by

$$a_0 x^0 + a_1 x^1 + a_2 x^2 + \dots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1} + a_n x^n$$

Step 1: Identify parameters to function: same as previous problem

Step 2: Return type: same as previous problem

Step 3: Designing body of the function: Consider the following polynomial

$$10 + 5x + 4x^2 + 3x^3 + 8x^4 + 9x^5$$

The above polynomial can also be written as shown below:

$$10 + x (5 + x (4 + x (3 + x (8 + 9x))))$$

which requires only 5 multiplications and 5 additions. In general, given the polynomial of the form

$$a_0 x^0 + a_1 x^1 + a_2 x^2 + \dots + a_{n-2} x^{n-2} + a_{n-1} x^{n-1} + a_n x^n$$

we can rearrange as shown below:

$$a_0 + x(a_1 + x(a_2 + x(a_3 + \dots + x(a_{n-4} + x(a_{n-3} + x(a_{n-2} + x(a_{n-1} + x a_n)))) \dots)))$$

The above representation requires only n multiplications and n additions there by number of computations can be reduced and efficiency can be improved. This method of solving the polynomial is called *Horner's method* of evaluating the polynomial. The equation can be written in reverse as shown below:

$$\begin{array}{ccccccc}
 (((a_n x + a_{n-1})x + a_{n-2})x + a_{n-3})x & \dots & + a_3)x + a_2)x + a_1)x + a_0 \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 (sum + a_{n-1})x & & & & & & \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 (sum + a_{n-2})x & & & & & & \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 sum + a_{n-3})x & & & & & & \\
 \dots & & & & & & \\
 (sum + a_3)x & & & & & & \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 (sum + a_2)x & & & & & & \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 (sum + a_1)x & & & & & & \\
 \downarrow & & \downarrow & & \downarrow & & \downarrow \\
 sum + a_0 & & & & & &
 \end{array}$$

In general, we can write $sum = (sum + a_i)x$ for $i = n-1, n-2, \dots, 3, 2, 1$

Observe from the above figure that the following three statements can be written:

- 1. Initial** $sum = a_n x$
- 2. General** $sum = (sum + a_i) x$ for $i = n-1, n-2, \dots, 3, 2, 1$
- 3. Final** $sum = sum + a_0$

The partial algorithm for the above three steps are shown below:

```
sum = a[n]*x
for i = n-1 down to 1
```

9.46 Arrays

```
        sum = (sum + a[i]) * x
    end for

sum = sum + a[0];
```

Note: All variables other than parameters should be declared as local variables i.e., the variables *sum* which is of type **double** and *i* which is of type **int** should be declared inside the function.

The C function to evaluate the above polynomial is shown below:

Example 9.26: C function to evaluate the polynomial

```
double evaluate ( double a[], int n, double x )
{
    double sum;
    int i;
    .....
    sum = a[n] * x;
    .....
    for (i = n-1; i >= 1; i--)
    {
        sum = ( sum + a[i] ) * x;
    }
    .....
    sum = sum + a[0];
    .....
    return sum;
}
```

The above function can be used to evaluate the polynomial and print the result as shown below:

Example 9.27: C Program to read the co-efficients and evaluate the polynomial

```
#include <stdio.h>
#include <math.h>
```

```
/* Include: Example 9.26: function to evaluate the polynomial */
```

```
void main()
{
    int n, i;
    double a[30], x, res;
```

TRACING

<code>printf("Enter the value of n\n");</code>	Enter the value of n
<code>scanf("%d",&n);</code>	5
<code>printf("Enter n+1 values\n");</code>	Enter n+1 values
<code>for (i = 0; i <= n; i++)</code>	i = 0 1 2 3 4 5
<code>{</code>	a[0] [1] [2] [3] [4] [5]
<code> scanf("%lf",&a[i]);</code>	6 5 4 3 2 1
<code>}</code>	
<code>printf("Enter the value for x\n");</code>	Enter the value for x
<code>scanf("%lf",&x);</code>	1
<code>/* Evaluate the polynomial */</code>	
<code>res = evaluate (a, n, x);</code>	res = 21
<code>printf("Result = %lf\n",res);</code>	Result = 21.000000
<code>}</code>	

9.9 Two-dimensional arrays (Multi-dimensional arrays)

In this section, let us concentrate on two-dimensional arrays. The term *dimension* represents number of indices (plural of index) used to access a particular item in an array. Now, the question is “*What are multi-dimensional arrays?*”

Definition: Arrays with one set of square brackets [] are single dimensional arrays. Arrays with two sets of square brackets [][] are called 2-dimensional arrays and so on. Arrays with two or more dimensions are called *multi-dimensional arrays*. It is the responsibility of the programmer to specify the number of elements to be processed within square brackets. For example,

```
int a[10];           /* a is declared as single dimensional array */
int b[10][10];       /* b is declared as two dimensional array */
int c[3][4][5];      /* c is declared as three dimensional array */
```

Arrays with more than 2-dimensions are not the scope of this book. Now, we discuss 2-dimensional arrays. Before proceeding further, let us answer the question “*What are 2-dimensional arrays? When are 2-dimensional arrays used?*”

Definition: Arrays with two sets of square brackets [][] are called 2-dimensional arrays. A two-dimensional array is used when data items are arranged in row-wise and column wise in a tabular fashion. Here, to identify a particular item we have to

9.48 Arrays

specify two indices (also called subscripts): the first index identifies the row number and second index identify the column number of the item.

9.10 Declaration of two dimensional arrays

Now, let us see “*How to declare two dimensional arrays?*” [DEC-2015/JAN-2016 3 MARKS] We know that all the variables are declared before they are used in the program. Similarly, a 2-dimensional array must be declared before it is used along with size of the array. The size used during declaration of the array is useful to reserve the specified memory locations. A 2-d array in C can be declared using the following syntax:

```
data_type array_name[exp1][exp2];
```

where

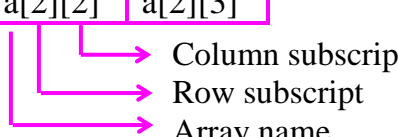
- ◆ *data_type* can be **int**, **float**, **char** etc.
- ◆ *array_name* is the name of the array.
- ◆ *exp1* and *exp2* are constant expressions. Here, *exp1* is the row index and *exp2* is the column index.

For example, consider the following declaration

```
int a[3][4];
```

The array *a* has two sets of square brackets `[]` and hence it is a 2-dimensional array with 3 rows and 4 columns. This declaration informs the compiler to reserve 12 locations ($3 * 4 = 12$) contiguously one after the other. The pictorial representation of this array *a* is shown below:

	Col- 0	Col -1	Col -2	Col -3
Row 0	a[0][0]	a[0][1]	a[0][2]	a[0][3]
Row 1	a[1][0]	a[1][1]	a[1][2]	a[1][3]
Row 2	a[2][0]	a[2][1]	a[2][2]	a[2][3]



9.11 Initialization of 2-dimensional arrays

Now, let us see “*How each memory location can be initialized?*” DEC-2015/JAN-2016] 3-MARKS

Assigning required value to a variable before processing is *initialization*. As we initialize a variable to the required value, we can initialize the individual elements of

a 2-dimensional array during initialization. The initialization can be using the following syntax.

```
data_type array_name[exp1] [exp2] = {  {a1, a2, .....an },
                                         {b1, b2, .....bn},
                                         .....
                                         .....
                                         {z1, z2,.....zn}
                                         };
```

where

- ◆ *data_type* can be **int**, **float** etc.,
- ◆ *exp1* and *exp2* are enclosed within square brackets. Both *exp1* and *exp2* can be integer constants or constant integer expressions .
- ◆ a₁ to a_n are the values assigned to 1st row, b₁ to b_n are the values assigned to 2nd row and so on.

9.11.1 Initializing all specified memory locations

Consider the initialization statement shown below:

```
int a[4][3] = {      {11, 22, 33},
                     {44, 55, 66},
                     {77, 88, 99},
                     {10, 11, 12}
                };
```

The declaration indicates that array *a* has 4 rows and 3 columns. The pictorial representation of this 2-dimensional array is shown below:

		columns →		
		0	1	2
rows ↓	0	11	22	33
	1	44	55	66
	2	77	88	99
	3	10	11	12

Note that a[0][0] = 11, a[2][1] = 88, a[3][2] = 12 and so on.

9.11.2 Partial array initialization

If the number of values to be initialized are less than the size of the array, then the elements are initialized from left to right one after the other. The remaining locations will be initialized to zero automatically.

Example 9.28: Consider the partial initialization shown below:

```
int a[4][3] = {
```

9.50 Arrays

```
        {11, 22},
        {33, 44},
        {55, 66},
        {77, 88}
    };
```

This declaration indicates that the array *a* has 4 rows and 3 columns in which only first two columns of each row are initialized. The 3rd column of each row will be initialized with zeros automatically as shown below:

		columns		
		0	1	2
rows	0	11	22	0
	1	33	44	0
	2	55	66	0
	3	77	88	0

i.e., *a*[0][2], *a*[1][2], *a*[2][2], *a*[3][2] all are initialized with zeros.

Example 9.29: consider the partial initialization shown below:

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

In the array declared each row should have three columns. But, we are initializing 1st row with 4 elements. This results in **syntax error** and the compiler flashes an error.

Example 9.30: What is the memory map of following 2-dimensional array?

```
int a[4][3] = {
                {11, 22, 33},
                {43, 55, 66},
                {77, 88, 99},
                {10, 11, 12}
            };
```

Even though we represent a matrix pictorially as a 2-dimensional array, in memory they are stored contiguously one after the other. Assume that address of first byte of *a*

(usually called base address) is 2000 and size of integer is 2 bytes. The memory map for the 2-dimensional array *a* is shown below:

<code>a[0][0]</code>	<code>a[0][1]</code>	<code>a[0][2]</code>	<code>a[1][0]</code>	<code>a[1][1]</code>	<code>a[1][2]</code>	<code>a[2][0]</code>	<code>a[2][1]</code>	<code>a[2][2]</code>	<code>a[3][0]</code>	<code>a[3][1]</code>	<code>a[3][2]</code>
11	22	33	44	55	66	77	88	99	10	11	12
2000	2002	2004	2006	2008	2010	2012	2014	2016	2018	2020	2022

9.11.3 Reading and writing two-dimensional arrays

Design: Now, let us see “How to read a 2-d matrix *z* of size *m* x *n* and how to print a 2-d matrix *z* of size *m* x *n*?”

Step 1: Identify parameters to function: Given the size of the matrix *m* x *n*, we have to read elements into matrix *z* or print elements of matrix *z*. So,

parameters are : `int z[][10], int m, int n`

Step 2: Return type: After reading a matrix or printing a matrix we are not returning any value and hence,

return type : `void`

Now, let us see “What are the different ways of accessing elements in a 2-dimensional array?” The elements of a matrix can be accessed in two different ways:

- ◆ Row major order
- ◆ Column major order

Row major order: In this method all elements are accessed one by one row wise i.e., all the elements in each row are accessed starting from 0th row to last row as shown below:

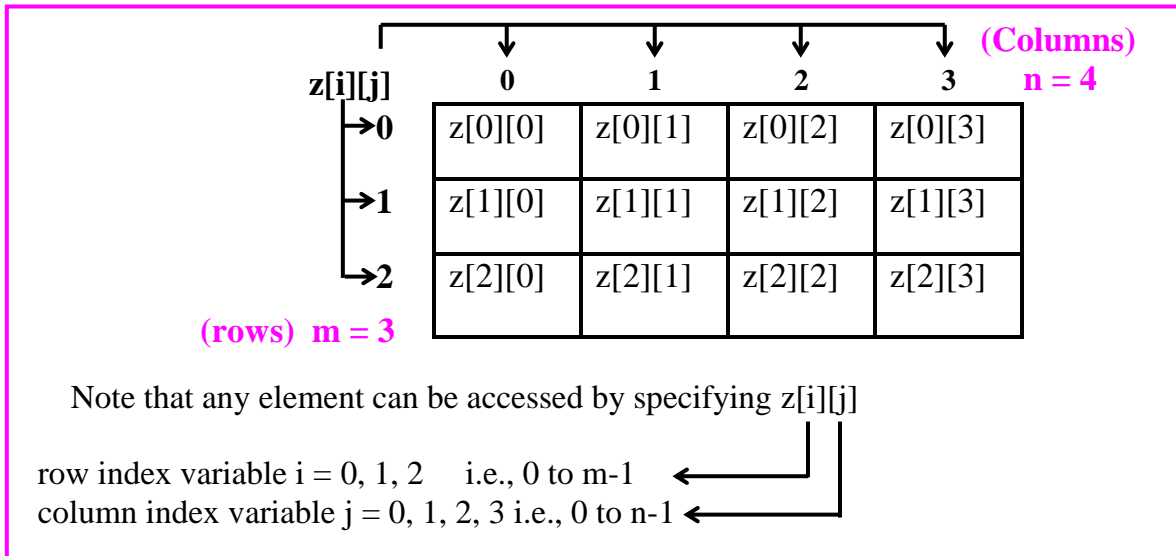
- ◆ Firstly, all elements in 0th row are accessed
- ◆ Secondly, all elements in 1st row are accessed
- ◆ Thirdly, all elements in 2nd row are accessed and so on.

Column major order: In this method all elements are accessed one by one column wise i.e., all the elements in each column are accessed starting from 0th column to last column as shown below:

- ◆ Firstly, all elements in 0th column are accessed
- ◆ Secondly, all elements in 1st column are accessed
- ◆ Thirdly, all elements in 2nd column are accessed and so on.

9.52 Arrays

Step 3: Designing body of the function (row major order): Any item can be accessed by specifying the *row* index and *column* index. Normally, we access the items row by row (**Note: Identified as row major order**) as shown below:



Note: To access each item *row wise* in 2-dimensional array, the row index i should be in the outer loop and column index j should be in the inner loop. So, the C statements to access any element $z[i][j]$ can be written as:

```

for (i = 0; i <= m-1; i++)          for (i = 0; i < m; i++)      /*m –Rows   */
{
    for (j = 0; j <= n-1; j++)        for (j = 0; j < n; j++) /*n-Columns*/
    {
        z[i][j];                      {
        }                               z[i][j];          /*access   */
    }
}
    
```

- ♦ By replacing $z[i][j]$ by `scanf("%d", &z[i][j])` we can read $m \times n$ elements.
- ♦ By replacing $z[i][j]$ by `printf("%d", z[i][j])` we can display $m \times n$ elements.

Note: Apart from the parameters m , n and z other variables used are: i and j . They give the position of row and column value of matrix z which is of type integer. So, i and j should be declared as local variables inside the function with data type **int**.

Example 9.31: C function to read a matrix of size $m \times n$ using row major order

row size is optional

column size has to be specified. otherwise, it is syntax error

```

void read_matrix ( int z[ ][10], int m, int n)
{
    int i;                /* Used as index to row items */
    int j;                /* Used as index to column items */

    for (i = 0; i < m; i++) /* m – rows */
    {
        for (j = 0; j < n; j++) /* n – columns */
        {
            scanf("%d",&z[i][j]); /* Read m x n items */
        }
    }
}

```

Note: When we pass 2 dimensional array as a parameter, row size is optional, but, column size has to be specified. **Omitting the column size leads to syntax error.**

Example 9.32: C function to print a matrix of size m x n

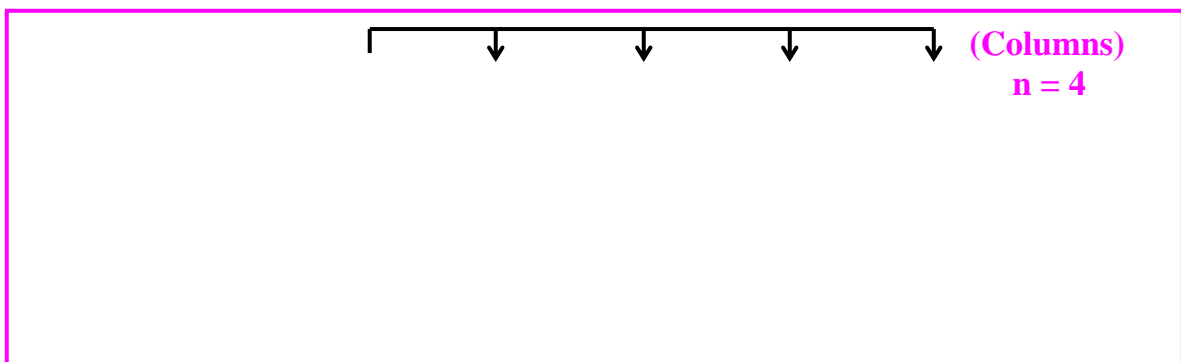
```

void print_matrix ( int z[ ][10], int m, int n)
{
    int i;                /* Used as index to row items */
    int j;                /* Used as index to column items */

    for (i = 0; i < m; i++) /* m – rows */
    {
        for (j = 0; j < n; j++) /* n – columns */
        {
            printf("%d  ", z[i][j]); /* print m x n items */
        }
        printf("\n");          /* move to new line */
    }
}

```

Step 3: Designing body of the function (column major order): Any item can be accessed by specifying the *column* index and *row* index. The elements can be accessed column wise by simply exchanging inner loop and outer loop given in previous section. This is pictorially represented as shown below:



9.54 Arrays

$z[i][j]$	0	1	2	3
→0	$z[0][0]$	$z[0][1]$	$z[0][2]$	$z[0][3]$
→1	$z[1][0]$	$z[1][1]$	$z[1][2]$	$z[1][3]$
→2	$z[2][0]$	$z[2][1]$	$z[2][2]$	$z[2][3]$

(rows) $m = 3$

Note that any element can be accessed by specifying $z[i][j]$

column index variable $j = 0, 1, 2, 3$ i.e., 0 to $n-1$

row index variable $i = 0, 1, 2$ i.e., 0 to $m-1$

Note: To access each item *column wise* in 2-dimensional array, the row index i should be in the inner loop and column index j should be in the outer loop. So, the C statements to access any element $z[i][j]$ in column major order can be written as:

<pre> for (j = 0; j <= n-1; j++) { for (i = 0; i <= m-1; i++) { z[i][j]; } } </pre>	⇒	<pre> for (j = 0; j < n; j++) /*n -columns */ { for (i = 0; i < m; i++) /*m-Columns*/ { z[i][j]; /*access */ } } </pre>
---	---	---

- ◆ By replacing $z[i][j]$ by `scanf("%d", &z[i][j])` we can read $m \times n$ elements column major order
- ◆ By replacing $z[i][j]$ by `printf("%d", z[i][j])` we can display $m \times n$ elements in column major order

Example 9.33: C function to read a matrix of size $m \times n$ using column major order

```

void read_matrix_column_wise ( int z[ ][10], int m, int n)
{
    int i;
    int j;

```

row size is optional ←

column size has to be specified. otherwise, it is syntax error ←

/* Used as index to row items */

/* Used as index to column items */

```

    for (j = 0; j < n; j++)                /* n – columns */
    {
        for (i = 0; i < m; i++)            /* m – rows */
        {
            scanf("%d",&z[i][j]);          /* Read m x n items column wise */
        }
    }
}

```

Note: When we pass 2 dimensional array as a parameter, row size is optional, but, column size has to be specified. **Omitting the column size leads to syntax error.**

Example 9.34: C function to print a matrix of size m x n using column major order

```

void print_matrix_column_wise ( int z[ ][10], int m, int n)
{
    int i;                                /* Used as index to row items */
    int j;                                /* Used as index to column items */

    for (j = 0; j < n; j++)                /* n – columns */
    {
        for (i = 0; i < m; i++)            /* m – rows */
        {
            printf("%d  ", z[i][j]);        /* print m x n items */
        }
        printf("\n");                      /* move to new line */
    }
}

```

9.12 Parallel arrays

Now, let us see “What are parallel arrays?” The same subscripts can be used in two or more arrays at the same time. These arrays which use the same variables as subscripts in a loop are called **parallel arrays**.

Ex 1: To copy n elements of array a to array b can be written as shown below:

```

for (i = 0; i < n; i++)
{
    b[i] = a[i];
}

```

9.56 Arrays

Observe that same index i is used for both arrays a and b hence arrays a and b are parallel arrays.

Ex 2: To add two matrices A and B and store the result in C is achieved using three 2-dimensional arrays A, B and C as shown in next section.

9.13 Addition of two matrices

Design: Now, let us see “How to add two matrices A and B and store in a matrix C of size $m \times n$?”

Step 1: Identify parameters to function: Given the two matrices A and B of size $m \times n$, it is required to store the result in matrix C. So,

parameters are : `int A[][10], int B[][10], int C[][10], int m, int n`

Step 2: Return type: After adding two matrices, we are not returning any value and hence,

return type : `void`

Step 3: Designing body of the function: We know that by specifying $A[i][j]$ we can access elements of matrix A and by specifying $B[i][j]$ we can access the elements of matrix B. After accessing, they have to be added and result should be stored in matrix C by specifying $C[i][j]$. This can be achieved using the following statement:

$$c[i][j] = a[i][j] + b[i][j];$$

Replacing the `scanf()` in function `read_matrix()` by the above statement, we can add two matrices.

Note: Apart from the parameters A, B, C, m and n, other variables used are: i and j and declared as local variables inside the function.

So, the C function to add two matrices is shown below:

Example 9.35: C function to add two matrices

```

void add_matrix (int a[][10], int b[][10], int c[][10], int m, int n)
{
    int i;                /* Used as index to row items */
    int j;                /* Used as index to column items */

    for (i = 0; i < m; i++)    /* m – rows */
    {
        for (j = 0; j < n; j++)    /* n – columns */
        {
            c[i][j] = a[i][j] + b[i][j];    /* Add matrix A and B */
        }
    }
}

```

The complete program along with various inputs and outputs is shown below:

Example 9.36: C Program to read two matrices A and B and store sum in matrix C

```
#include <stdio.h>
```

```
/* Include: Example 9.31: function to read a matrix */
```

```
/* Include: Example 9.32: function to print a matrix */
```

```
/* Include: Example 9.35: function to add two matrices */
```

```
void main()
```

```
{
```

```
    int m, n, a[10][10], b[10][10], c[10][10];
```

```
    printf("Enter the size of the matrix\n");
```

```
    scanf("%d %d",&m, &n);
```

```
/* Read the elements of matrix A*/
```

```
    printf("Enter the elements of matrix a\n");
```

```
    read_matrix(a, m, n);
```

```
    printf("Enter the elements of matrix b\n");
```

```
    read_matrix(b, m, n);
```

TRACING

```
Enter the size of the matrix
2 3
```

```
Enter the elements of matrix a
1 2 3
4 5 6
```

```
Enter the elements of matrix b
1 2 3
4 5 6
```

9.58 Arrays

<code>add_matrix(a, b, c, m, n);</code>		<code>1+1 2+2 3+3 = 2 4 6</code>
		<code>4+4 5+5 6+6 = 8 10 12</code>
.....		
<code>printf("The resultant matrix c\n");</code>		The resultant matrix c
<code>print_matrix(c, m, n);</code>		<code>2 4 6</code>
		<code>8 10 12</code>
}		

9.14 Find the product of two matrices

Design: In this section, let us see “How to multiply matrices A and B of size $m \times n$ and $p \times q$ and store the result in matrix C?”

Step 1: Identify parameters to function: Given the two matrices A and B of size $m \times n$ and $p \times q$, it is required to store product in matrix C. So,

parameters are : `int A[][10], int B[][10], int C[][10], int m, int n, int p, int q`

Step 2: Return type: After multiplying two matrices, we are not returning any value and hence,

return type : `void`

Step 3: Designing body of the function: *Multiplication of two matrices is possible if and only if the number of columns of first matrix and number of rows of the second matrix are same.* Given $m \times n$ as the size of first matrix and $p \times q$ as the size of second matrix. If n is equal to p , multiplication is possible. If n is not equal to p , multiplication is not possible. The equivalent code can be written as shown below:

```
if ( n != p )
{
    printf ("Multiplication is not possible\n");
    exit(0);
}
```

Once control comes out of the **if** statement, it indicates multiplication is possible. If size of A is $m \times n$, size of B is $n \times q$, then size of C is $m \times q$. The general formula to multiply two matrices using index variables i, j, k is shown below:

Size of resulting matrix C is $m \times q$

$$c[i][j] = \sum_{k=0}^{n-1} (a[i][k] * b[k][j])$$

index variable $i = 0, 1, 2, \dots, m-1$
 index variable $j = 0, 1, 2, \dots, q-1$
 index variable $k = 0, 1, 2, \dots, n-1$

The above mathematical formula can be converted into C equivalent statements as shown below:

```
for (i = 0; i < m; i++)
{
    for (j = 0; j < q; j++)
    {
        sum = 0;
        for (k = 0; k < n; k++)
        {
            sum = sum + a[i][k]*b[k][j];
        }
        c[i][j] = sum;
    }
}
```

Note: Apart from the parameters A, B, C, m, n, p and q, other variables used are: i , j , k and sum . They should be declared as local variables inside the function with data type **int**.

So, the C function to multiply two matrices is shown below:

Example 9.37: C function to multiply two matrices A and B and store the result in C

```
void multiply_matrix (int a[][10], int b[][10], int c[][10], int m, int n, int p, int q)
{
    int i;                /* Used as index to matrix A and C */
    int j;                /* Used as index to matrix B and C */
```

9.60 Arrays

```
int k;                                /* Used as index to matrix A and B */
int sum;                              /* Used to store the partial result */

/* Check whether multiplication of two matrices possible */
if (n != p)
{
    printf("Multiplication is not possible\n");
    exit(0);
}

/* Multiply the matrices A and B */
for (i = 0; i < m; i++)
{
    for (j = 0; j < q; j++)
    {
        sum = 0;
        for (k = 0; k < n; k++)
        {
            sum = sum + a[i][k] * b[k][j];
        }
        c[i][j] = sum;
    }
}
```

The complete program along with various inputs and outputs is shown below:

Example 9.38: C Program to read two matrices A and B and store result in matrix C

```
#include <stdio.h>
#include <process.h>    Note: The function exit() is declared in this header file

/* Include: Example 9.31: function to read a matrix in row major order */
/* Include: Example 9.33: function to read a matrix in column major order */
/* Include: Example 9.32: function to print a matrix */
/* Include: Example 9.37: function to multiply two matrices */
void main()
{
    int m, n, p, q, a[10][10], b[10][10], c[10][10];
```

<pre> printf("Enter the size of the matrix a\n"); scanf("%d %d",&m, &n); /* Read the elements of matrix A*/ printf("Enter the elements of matrix a\n"); read_matrix(a, m, n); printf("Enter the size of the matrix b\n"); scanf("%d %d",&p, &q); /* Read the elements of matrix B*/ printf("Enter items of matrix b column wise"); read_matrix_column_wise(b, p, q); /* Multiply matrix a, b and store result in c */ multiply_matrix(a, b, c, m, n, p, q); /* Display the resultant matrix c */ printf("The product matrix c\n"); print_matrix(c, m, q); </pre>	<p>Input</p> <p>Enter the size of the matrix 2 3</p> <p>Enter the elements of matrix a 1 2 3 4 5 6</p> <p>Enter the size of the matrix 3 2</p> <p>Enter the elements of matrix b 1 2 3 3 4 5 5 6</p> <p>Computations</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>$1*1+2*3+3*5$</td> <td>$1*2+2*4+3*6$</td> </tr> <tr> <td>$4*1+5*3+6*5$</td> <td>$4*2+5*4+6*6$</td> </tr> <tr> <td style="text-align: center;">↓</td> <td></td> </tr> <tr> <td>22</td> <td>28</td> </tr> <tr> <td>49</td> <td>64</td> </tr> </table> <p>The resultant matrix c</p> <table style="margin-left: auto; margin-right: auto;"> <tr> <td>22</td> <td>28</td> </tr> <tr> <td>49</td> <td>64</td> </tr> </table>	$1*1+2*3+3*5$	$1*2+2*4+3*6$	$4*1+5*3+6*5$	$4*2+5*4+6*6$	↓		22	28	49	64	22	28	49	64
$1*1+2*3+3*5$	$1*2+2*4+3*6$														
$4*1+5*3+6*5$	$4*2+5*4+6*6$														
↓															
22	28														
49	64														
22	28														
49	64														

9.15 Sum of elements of a given matrix

Design: The procedure is same as addition of two matrices. Instead of adding $a[i][j]$ with $b[i][j]$, we add $a[i][j]$ with *sum* whose value is initialized to zero in the beginning. The equivalent statements are shown below:

Example 9.40: C function to print sum of elements of a given matrix

```

void sum_matrix (int  a[][10], int  m, int  n)
{
    int i, j, sum;

```

9.62 Arrays

```
sum = 0;
for (i = 0, i < m; i++)           /* m – rows */
{
    for (j = 0; j < n; j++)       /* n – columns */
    {
        sum = sum + a[i][j];     /* add all items of a */
    }
}
printf("Sum of all elements = %d\n", sum);
}
```

Once control comes out of the outer loop, the variable *sum* contains the sum of all the elements of a given matrix.

9.16 Sum of rows of a matrix

Design: Same as previous function, but initialize *sum* to zero inside the outer loop. The sum obtained will be copied into a single dimensional array *row_sum*. The array *row_sum* contains sum of each row in the given matrix. The complete C function is shown below:

Example 9.41: C function to find sum of each row

```
void sum_of_rows (int a[ ][10], int m, int n, int row_sum[])
{
    int i, j, sum;

    for (i = 0; i < m; i++)
    {
        sum = 0;
        for (j = 0; j < n; j++)
        {
            sum += a[i][j];
        }
        row_sum[i] = sum;
    }
}
```

9.17 Sum of columns of a matrix

Design: Same as previous function, but interchange i^{th} for loop and j^{th} for loop. The sum obtained will be copied into a single dimensional array *column_sum*. The array

column_sum contains sum of each column in the given matrix. The complete C function is shown below:

Example 9.42: C function to find sum of each column

```
void sum_of_columns (int a[ ][10], int m, int n, int column_sum[])
{
    int i, j, sum;

    for (j = 0; j < n; j++)
    {
        sum = 0;
        for (i = 0; i < m; i++)
        {
            sum += a[i][j];
        }

        column_sum[j] = sum;
    }
}
```

9.18 Display matrix along with row and column

Design: After displaying *i*th row elements and before moving to new line, display the corresponding *row_sum*. The corresponding code can be written as shown below:

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

    printf ("= %d\n", row_sum[i]);
}
```

Now, display the sum of *column_sum* using the following statements:

9.64 Arrays

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

The C function to display matrix along with row sum and column sum can be written as shown below:

Example 9.43: C function to find print a matrix along with row sum and column sum

```
void display(int a[] [10], int m, int n, int row_sum[], int column_sum[])
{
    int i, j;
    for (i = 0; i < m; i++)
    {
        for (j = 0; j < n; j++)
        {
            printf("%d  ", a[i][j]);
        }
        printf("| = %d\n", row_sum[i]);
    }
    printf("-----\n");
    for (j = 0; j < n; j++)
    {
        printf("%d  ", column_sum[j]);
    }
}
```

The complete program that uses the above function to find sum of rows and columns is shown below:

Example 9.44: C Program to read two matrices A and B, store product in matrix C

```
#include <stdio.h>
#include <process.h>          Note: The function exit() is declared in this header file

/* Include: Example 9.31: function to read a matrix */
/* Include: Example 9.41: function to find sum of rows in a matrix */
/* Include: Example 9.42: function to find sum of columns in a matrix */
/* Include: Example 9.43: function to display matrix, row sum and column sum */
```



```
void main()
{
    int m, n, a[10][10], row_sum[10], column_sum[10];

    printf("Enter the size of the matrix a\n");
    scanf("%d %d",&m, &n);

    printf("Enter the elements of matrix a\n");
    read_matrix(a, m, n);

    sum_of_rows(a, m, n, row_sum);

    sum_of_columns(a, m, n, column_sum);

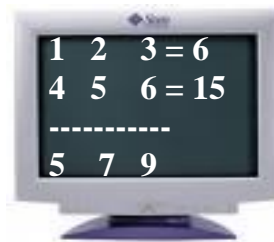
    display (a, m, n, row_sum, column_sum);
}
```

Input

Enter the size of the matrix

2 3

Enter the elements of matrix a

1 2 3**4 5 6**

9.66 Arrays

Exercises

1. What is an array? What is the use of using arrays? How are they declared in C? What are the rules to be followed while using arrays
2. How an array can be initialized? Explain with example.
3. Write a C program to multiply two given matrices A, B and store the result in C
4. Write a note on one dimensional and two dimensional arrays
5. Write a C program to read n integer numbers interactively and to print biggest and smallest of n numbers
6. Given two sets A and B of integers, write a program to read them, determine its UNION and INTERSECTION and print the resultant sets
7. A and B are two arrays each with 10 elements. Write a program to find array C such that
$$\begin{aligned}C[0] &= A[0] + B[9] \\C[1] &= A[1] + B[8] \\C[2] &= A[2] + B[7] \\&\dots\dots\dots \\&\dots\dots\dots \\C[8] &= A[8] + B[1] \\C[9] &= A[9] + B[0]\end{aligned}$$
8. Write a program to find the intersection of 2 arrays A and B with size m and n respectively
9. Write a C program to check for an item in an array of n elements using linear (sequential) search
10. Write a C program to check whether an item is present in the array using binary search
11. Write a C program to sort N numbers using bubble sort