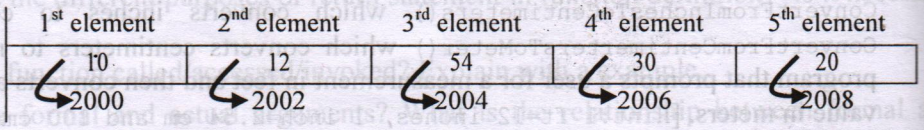
Chapter-8

Arrays and Strings

An array is a group of related data items that share a common name. In other words, an array is a data structure that store a number of data items as a single entity (object). The individual data items are called elements and all of them have same data types. Array is used when multiple data items that have common characteristics are required in a program. In other words, an array is a collection of individual data elements that is ordered (one can count off the elements 0, 1, 2, 3,..), fixed in size and homogeneous (all elements have to be of the same type e.g., `int`, `float`, `char` etc). Suppose we have 20 numbers of type integer and we have to sort them in ascending or descending order. If we have no array, we have to define 20 different variables like `a1`, `a2`, `a3`,.....`a20` of type `int` to store these twenty numbers which will be possible but inefficient. If the number of integers increases the number of variables will also be increased and defining different variables for different numbers will be impossible and inefficient. In such situation where we have multiple data items of same type to be stored, we can use array. In array system, an array represents multiple data items but they share same name. The individual elements are characterized by array name followed by one or more subscripts or indices enclosed in square brackets. The individual data items can be `characters`, `integers`, `floating` point numbers etc. However, they must all be of the same type and the same storage class (i.e. `auto`, `register`, `static` or `extern`).

8.1 One-Dimensional Array

There are several forms of an array in C: `one dimensional` and `multi-dimensional` array. In one-dimensional array, there is a single subscript or index whose value refers to the individual array element which ranges from `0` to `(n-1)` where `n` is the size of the array. e.g. `int a[5];` is a declaration of one dimensional array of type `int`. Its elements can be illustrated as



> The first element of an array is indexed `zero`, so the last element is one less than the size of the array. But C doesn't warn when an array subscript exceeds the size of the array

The elements of an integer array `a[5]` are stored in continuous memory locations. It is assumed that the starting memory location is `2000`. As each integer element requires `2` bytes, subsequent element appears after gap of `2` locations.

8.1.1 Declaration of 1-D array

Syntax:

```
storage_class   data_type   array_name[size];
```

i. `storage_class` refers to the storage class of the array. It may be `auto`, `static`, `extern` and `register`. It is optional.

ii. `data_type` is the data type of array. It may be `int`, `float`, `char` ....etc. The array elements are all values of the type `data_type`. If `int` is used, this means that this array stores all data items of integer types.

iii. `array_name` is name of the array. It is user defined name for array. The name of array may be any name valid for name of variable.

iv. `size` of the array is the number of elements in the array. The `size` is mentioned within `[ ]` bracket. The size must be an integer constant like `10, 15, 100,...` or a constant expression like symbolic constant (i.e. if symbolic constant `SIZE` is defined as `#define SIZE 80`, the array can be defined as `int a[size];`). The size of the array must be specified (i.e. should not be blank) except in array initialization.

Example:

An array named `nums` can be defined as

```
int nums [5];
```

This statement tells to the compiler that 'nums' is an array of type `int` and can store five integers. The compiler reserves `2` bytes of memory for each integer array element. Thus, total memory size allocated by an integer array of size `5` is `10` bytes. It's individual elements are recognized by `nums[0]`, `nums[1]`, `nums[2]`, `nums[3]` and `nums[4]`. The integer value within square bracket (i.e. `[ ]`) is called `index` of array. Index of an array always starts from `0` and ends with one less than size of array.

The other valid names of arrays are

`long marks[5];` i.e. marks is a `long integer` array of size `5`. It can store `5` long values.

`char name[10];` i.e. name is a `char` array of size `10` and it can store `10` characters.

`float salary[50];` i.e. salary is a `float` array of size `50` and it can store `50` fractional numbers.

8.1.2 Initialization of 1-D array

Till the array elements are not given any specific values, they are supposed to contain garbage values. The values can be assigned to elements at the time of array declaration, which is called array initialization. Since an array has multiple elements, braces are used to denote the entire array and commas are used to separate the individual values assigned to the elements in the array initialization statements. The syntax for initialization is

```
storage_class  data_type  array_name[size]={vaue1, value2,...value n};
```

Where `value1` is value of first element, `value2` is that of second and so on.

The array initialization may be of three types as below.

a) `int a[5]= {1,2,3,5,8};`

Here, `a` is integer type array which has 5 elements. Their values are assigned as `1, 2, 3, 5, 8` (i.e. `a[0]=1, a[1]=2,a[2]=3,a[3]=5` and `a[4]=8`). The array elements are stored sequentially in separate memory locations.

b) `int b[ ]= {2,5,7};`

Here, size of array is not given, the compiler automatically sets it size according to the number of values given. The size of array `b` is `3` in this array initialization as three values `2, 5` and `7` are assigned at the time of initialization. Thus, writing `int b[ ]= {2,5,7};` is equivalent to writing `int b[3]= {2,5,7};`

c) `int c[10]={45,89,54,8,9};`

In this example, the size of array `c` has been set `10` but only `5` elements are assigned at the time of initialization. In this situation, all individual elements that are not assigned explicitly contains `zero` as initial values. Thus, the value of `c[5], c[6], c[7], c[8]` and `c[9]` is `zero` in this example.

8.1.3 Accessing array elements

The single operation, which involves entire array, are not permitted in C. Thus, if `num` and `list` are two similar arrays (i.e. same data type, dimension and size), then assignment operations, comparison operations etc., involving these two arrays must be carried out on an element-by-element basis. All the elements of an array can neither be set at once nor one array may be assigned to another.

For example:

```
int a[5], b[5];
a=0;  /* wrong */
b=a;  /* wrong */
if(a<b)
{...}  /* wrong */
```

The particular array element in an array is accessed by specifying the name of array, followed by square bracket enclosing an integer, which is called the array index. The array index indicates the particular element of the array which we want to access. The numbering of elements starts from zero and ends to a number one less than the size of the array. For example, if an array `marks` is defined as

```
float marks[5]
```

Then, the array elements are `marks[0], marks[1], marks[2], marks[3]` and `marks[4]`. We can assign float values to these individual elements directly.

i.e.

```
marks[0]=90.7;
marks[1]=45;
marks[2]=78.5;
marks[3]=0;
marks[4]=45.8;
```

A loop can be used to input and output the elements of array.

Example 8.1

Write a program that reads 10 integers from keyboard, stores in an array and displays entered numbers in the screen.

```
#include<stdio.h>
#include<conio.h>
int main()
    {
    int a[10], i;
    printf("Enter 10 numbers:\t");
    for(i=0;i<10;i++)
    scanf("%d",&a[i]);  /* array input */
    printf("\n You have entered these 10 numbers:\n");
    for(i=0;i<10;i++)
    printf("\ta[%d]=%d",i,a[i]);  /* array output*/
    getch();
    return 0;
    }
```

Output

```
Enter 10 numbers:   10  30  45   23   45   68   90   78   34   32
You have entered these 10 numbers:
a[0]=10    a[1]=30    a[2]=45    a[3]= 23    a[4]= 45    a[5]=68
a[6]= 90   a[7]=78    a[8]= 34   a[9]= 32
```

Explanation: In this example, the variable `i` is varying from `0` to `9` and the statement `scanf("%d",&a[i]);` implies that the entered number is stored in address of array element `a[i]` where `i` is an integer variable. Thus, address of `i`th element of array (i.e. `&a[i]`) is passed in `scanf()` function to store the value in this address. Hence, here `a[i]` represents individual array element and array element depends on the value of `i`, which acts as index of the array.

Example 8.2

Write a program to illustrate the memory locations allocated by each array elements.

```
#include<stdio.h>
#include<conio.h>
int main()
    {
    float a[]={10.4,45.9,5.5,0,10.6};
    int i;
    printf("The continuous memory locations are:\n");
    for(i=0;i<5;i++)
        printf("\t%u",&a[i]);  /* address of array elements*/
    getch();
    return 0;
    }
```

Output

```
The continuous memory locations are:
65506   65510   65514   65518   65522
```

Explanation: As memory reserved by each array element of type float is 4 bytes, subsequent element appears after gap of 4 memory locations. Here, conversion character u (i.e. unsigned) is used to display the address of the array element as the address values like 65506 does not lie on the range of signed integers.

Example 8.3

Write a program that reads mark's percentage in an examination of 10 students. Calculate and display the average percentage and deviation percentage from average of each student.

```
#include<stdio.h>
#include<conio.h>
int main()
    {
    float marks[10],avg,dev,sum=0;
    int i;
    printf("\nEnter percentage of 10 students\n");
    for(i=0;i<10;i++)
        {
        printf("\tmarks[%d]=",i);
        scanf("%f",&marks[i]);
        sum+=marks[i];
        }
    avg=sum/10;
    printf("\nThe average marks is\t%.2f",avg);
    printf("\nThe deviation of each student form average\n");
    for(i=0;i<10;i++)
    {
     dev=marks[i]-avg;
     printf("\nmarks[%d]=%.2f\tdeviation=%.2f",i,marks[i],dev);
     }
  getch();
  return 0;
  }
```

Output

```
Enter percentage of 10 students
        marks[0]=45
        marks[1]=67.5
        marks[2]=45
        marks[3]=89
        marks[4]=90
        marks[5]=34
        marks[6]=56
        marks[7]=76.5
        marks[8]=43
        marks[9]=48
```

```
The average marks is    59.40
The deviation of each student form average
marks[0]=45.00  deviation=-14.40
marks[1]=67.50  deviation=8.10
marks[2]=45.00  deviation=-14.40
marks[3]=89.00  deviation=29.60
marks[4]=90.00  deviation=30.60
marks[5]=34.00  deviation=-25.40
marks[6]=56.00  deviation=-3.40
marks[7]=76.50  deviation=17.10
marks[8]=43.00  deviation=-16.40
marks[9]=48.00  deviation=-11.40
```

Example 8.4

Write a program to sort n numbers in ascending order.

```
#include<stdio.h>
#include<conio.h>
int main()
    {
     int nums[50],i,j,n,temp;
     printf("\nHow many numbers are there?\t");
     scanf("%d",&n);
     printf("\nEnter %d numbers:\n",n);
     for(i=0;i<n;i++)
          scanf("%d",&nums[i]);

      for(i=0;i<n-1;i++)
        {
       for(j=i+1;j<n;j++)
          {
          if(nums[i]>nums[j])
              {
               temp=nums[i];
               nums[i]=nums[j];
               nums[j]=temp;
              }
          }
        }
      printf("\nThe numbers in ascending order:\n");
      for(i=0;i<n;i++)
          printf("\t%d",nums[i]);
      getch();
      return 0;
    }
```

Output

```
How many numbers are there?     10
Enter 10 numbers:
12          56      3       90      234     78      235     97      22      46
The numbers in ascending order:
 3   12      22      46      56  78      90      97      234     235
```

8.2 Characteristics of Array

1) The declaration `int a [5]` is nothing but creation of `5` variables of integer types in the memory. Instead of declaring five variables for five values, the programmer can define them in the array.
2) All the elements of an array share the same name, and they are distinguished from one another with the help of an element number or array index.
3) The element number or array index in the array plays an important role for calling each element.
4) Any particular element of an array can be modified separately without disturbing other elements.
5) The single operation, which involves entire arrays, are not permitted in C. Thus, if `num` and `list` are two similar arrays (i.e. same data type, dimension and size), then assignment operations, comparison operations etc., involving these two arrays must be carried out on an element-by-element basis.
6) Any element of an array can be assigned/equated to another ordinary variable or array variable of its type.

```
e.g. int b, a[10],c=90;
     b=a[5];   /* ok */
     a[2]=a[3];  /* ok */
     a[3]=c;    /* ok */
```

8.3 Multi-Dimension Arrays

Multi-dimensional arrays are those which have more than one dimensions. Multi-dimensional arrays are defined in much the same manner as one dimensional array, except that a separate pair of square brackets is required for each subscript or dimension or index. Thus, two dimensional arrays will require two pairs of square brackets; three dimensional arrays will require three pairs of square brackets and so on. The two dimensional array is also called matrix. Multi-dimensional array can be defined as following

```
storage_class data_type  array_name[dim1] [dim2] ...[dimn];
```

Here, `dim1, dim2.....dimn` are positive valued integer expressions that indicate the number of array elements associated with each subscript. An `m*n` two dimensional array can be thought as tables of values having `m` rows and `n` columns. For example: `int X[3][3]` can be shown as follows

	Col 1	Col 2	Col 3
Row 1	`X[0][0]`	`X[0][1]`	`X[0][2]`
Row 2	`X[1][0]`	`X[1][1]`	`X[1][2]`
Row 3	`X[2][0]`	`X[2][1]`	`X[2][2]`

8.3.1 Declaration of two-dimensional array

Like one dimensional array, two dimensional arrays must also be declared before using it. The syntax for declaration is

```
[storage class] data_type  array_name[row_size][col_size];
```

Example:

```
int matrix[2][3]; /* matrix is 2-D array which has 2 rows and 3 columns
i.e. it contains 6 elements*/
float m[10][20];
char students[10][15];
```

8.3.2 Initialization of multi-dimensional array

Like one dimensional array, multi-dimensional arrays can be initialized at the time of array definition. Some examples of 2-D array initialization are:

```
1) int marks[2][3]={   {2, 4, 6}, {8, 10, 12}  };
```

is equivalent to following assignments

```
marks[0][0]=2;    marks[0][1]=4;    marks[0][2]=6;
marks[1][0]=8;    marks[1][1]=10;   marks[1][2]=12;
```

```
2) int marks1[ ][3]={   {2, 4, 6},  {8, 10, 12}  };
```

is equivalent to

```
marks[0][0]=2;    marks[0][1]=4;    marks[0][2]=6;
marks[1][0]=8;    marks[1][1]=10;   marks[1][2]=12;
```

First dimension of 2-D array may be empty while initializing 2-D array.

```
3) int marks2[2][3]={ 2,4,6,8, 10, 12};
```

is equivalent to 1 or 2

```
4) int marks3[ ][3]={ 2,4,6,8, 10, 12};
```

is equivalent to 1 or 2 or 3

```
5) int marks4[ ][3]={ 2,4,6,8, 10};
```

is equivalent to

```
marks[0][0]=2;    marks[0][1]=4;    marks[0][2]=6;
marks[1][0]=8;    marks[1][1]=10;   marks[1][2]=0;
```

i.e. If number of values provided while initialization is less than size of array, zero is assigned to the remaining elements.

All above examples are valid. While initializing an array, it is necessary to mention the second dimension (column) whereas the first dimension (row) is optional. Thus, following are invalid initializations.

```
6) int marks3[3][ ]={ 2,4,6,8, 10,. 12};  /*error*/
7) int marks4[ ][ ]={ 2,4,6,8, 10};    /* error!!!*/
```

The different tables with particular rows and colums may be treated as three dimensional (3-D) array and shall be initialized as

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

Example 8.5

Write a program to initialize 2-D array and display its elements.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int mat[2][3]={{1,2,3},{10,20,40}};
int i,j;
for(i=0;i<2;i<i++)
{
  printf("\nRow number %d:",i);
  for(j=0;j<3;j++)
     printf("\t%d",mat[i][j]);
  }
getch();
return 0;
}
```

Example 8.6

Write a program to initialize 3-D array and disply its elements.

```
#include<stdio.h>
#include<conio.h>
int main()
{
int marks[3][2][4] =
     {
     {{ 2, 4, 6, 8},{1, 3, 5, 7}},
     {{9, 2, 5, 8},{5, 7, 8, 2}},
     {{1, 4, 8, 2},{2, 4, 5, 3}}
     };
int i,j,k;
for(i=0;i<3;i<i++)
 {
  printf("\n\nTable number:%d",i);
  for(j=0;j<2;j++)
     {
     printf("\nRow number:%d",j);
     for(k=0;k<4;k++)
        printf("\t%d",marks[i][j][k]);
     }
 }
getch();
```

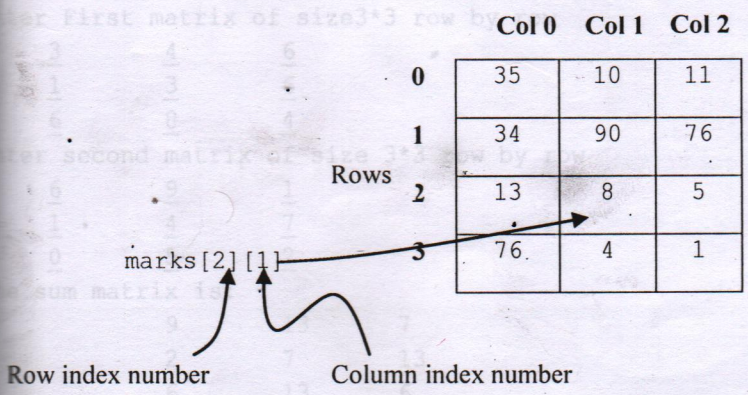
```
return 0;
}
Output
Table number:0
Row number:0    2    4    6    8
Row number:1    1    3    5    7

Table number:1
Row number:0    9    2    5    8
Row number:1    5    7    8    2

Table number:2
Row number:0    1    4    8    2
Row number:1    2    4    5    3
```

8.3.3 Accessing 2-D array elements

In 2-D array, the first dimension specifies number of rows and second specifies number of columns. Each row contains elements of many columns. Thus, a row is 1-D array. 2-D array contains multiple rows (i.e. 1-D arrays). Thus, 2-D array is an `array of 1-D arrays`. As each row will contain elements of many columns, the column index runs from `0 to size-1` inside every row in the one-dimensional representation (where `size` is the number of columns in the array). So, the column index changes faster than the row index as the one-dimensional representation of the array inside the computer is traversed. 2-D array is traversed row by row (i.e. every column elements in first row are traversed first and then column elements of second row are traversed and so on). The nested loop is used to traverse the 2-D array. Let us consider a following 2-D array `marks` of size `4*3` (i.e. matrix having `4` rows and `3` columns).

```
35    10    11
34    90    76
13    8     5
76    4     1
```

The matrix can be illustrated with row and column number as below.



This array is traversed in the following order:

35 => 10 => 11 => 34 => 90 => 76 => 13 => 8 =>5 => 76 => 4 =>1

i.e.marks[0][0]=>marks[0][1]=>marks[0][2]=>marks[1][0]=>marks[1][1]=> marks[1][2]=> marks[2][0]=> marks[2][1]=> marks[2][2]=> marks[3][0]=> marks[3][1]=> marks[3][2].

To access a particular element of 2-D array, we have to specify the array name, followed by two square brackets with row and column number inside it. For example: `Marks[0][0]` represents 35, `marks[1][1]` (i.e. 1 in both braces represents second row and second columns as array index starts from 0) represents 90, `marks[2][2]` represents 5 and `marks[1][2]` represents 76 in above matrix.

Example 8.7

Write a program to read a matrix of size 2*3 from user and display it to screen.

```
#include<stdio.h>
#include<conio.h>
int main()
   {
   int matrix[2][3],i,j;
   for(i=0;i<2;i++)
    for(j=0;j<3;j++)
     {
      printf("Enter matrix[%d][%d]:\t",i,j);
      scanf("%d",&matrix[i][j]);
     }
    printf("\nThe entered matrix is:\n");
    for(i=0;i<2;i++)
      {
       for(j=0;j<3;j++)
         printf("%d\t",matrix[i][j]);
      printf("\n");
     }
   getch();
   return 0;
   }
```

Output

```
Enter matrix[0][0]:     10
Enter matrix[0][1]:     20
Enter matrix[0][2]:     80
Enter matrix[1][0]:     4
Enter matrix[1][1]:     6
Enter matrix[1][2]:     7
The entered matrix is:
    10      20      80
    4       6       7
```

Example 8.8

Write a program to read two m*n matrices and display their sum.

```
#include<stdio.h>
#include<conio.h>
#define m 3
#define n 3
int main()
    {
      int a[m][n],b[m][n],sum[m][n],i,j;
      printf("Enter First matrix of size%d*%d row by row\n",m,n);
      for(i=0;i<m;i++)
      for(j=0;j<n;j++)
      scanf("%d",&a[i][j]);
      printf("Enter second matrix of size %d*%d row by row\n",m,n);
      for(i=0;i<m;i++)
      for(j=0;j<n;j++)
      scanf("%d",&b[i][j]);
      for(i=0;i<m;i++)
      for(j=0;j<n;j++)
      sum[i][j]=a[i][j]+b[i][j];
      printf("\nThe sum matrix is:\n");
      for(i=0;i<m;i++)
      {
      for(j=0;j<n;j++)
      printf("\t%d",sum[i][j]);
      printf("\n");
      }
      getch();
      return 0;
    }
```

Output

```
Enter First matrix of size3*3 row by row
3       4       6
1       3       6
6       8       4
Enter second matrix of size 3*3 row by row
6       9       1
1       4       7
0       5       2
The sum matrix is:
        9       13      7
        2       7       13
        6       13      6
```

8.4 Passing Arrays to Function

Like wise ordinary variables and values, it is possible to pass the value of an array element and even an entire array as an argument to a function. To pass an entire array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument in function call statement. The corresponding formal argument in function definition is written in the same manner, though it must be declared as an array. When declaring a one-dimensional array as a formal argument, the array name is written with a pair of empty square brackets. The size of the array is not necessarily specified within the formal argument declaration.

Syntax for function call passing array as argument:

```
function_name(array_name);
```

Syntax for function prototype which accepts an array as argument:

```
return_type function_name (data_type  array_name[]);
```

or

```
return_type function_name(data_type *pointer_variable);
```

[The second version uses pointer. The detail will be used in next chapter: Pointer]

When an array is passed to a function, the values of the array elements are not passed to the function. Rather, the array name is interpreted as the address of the first array element. This address is assigned to the corresponding formal argument when the function is called. The formal argument therefore becomes a pointer to the first element of the array. Thus, array is passed to a function using `call by reference`. Thus, the change in the values of formal arguments within function is also recognized in actual argument in the `main()` function. Thus, the passing an `array-name` to a function doesn't create a copy of of the array. Rather the formal argument and actual argument represents the same array..

Example 8.9

Write a program to illustrate passing array to a function one element at a time.

```
void display(int n)
 {
  printf("\t%d",n);
 }
int main()
 {
  int nums[5]={100,20,3,40,15},i;
  printf("\nThe content of array is:\n");
  for(i=0;i<5;i++)
     display(nums[i]);  /* pass array element to function*/
  getch();
  return 0;
 }
```

Output

```
The content of array is:
    100     20      3       40      15
```

Example 8.10

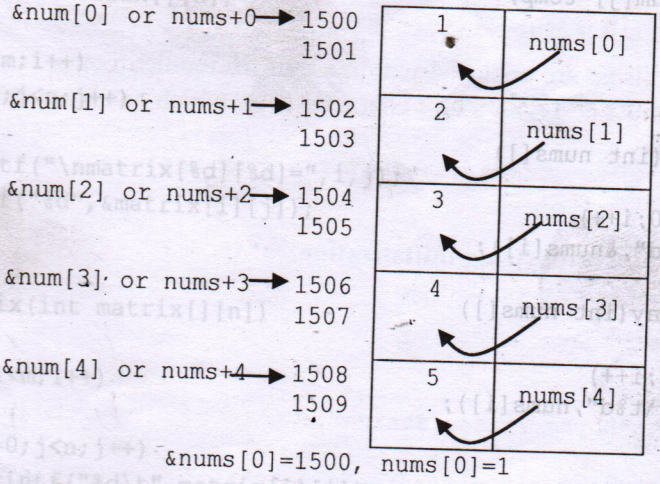
Write a program to illustrate passing an entire array to a function.

```
void change(int a[])
{
a[0]=10;
a[1]=20;
a[2]=30;
a[3]=40;
a[4]=50;
}
int main()
 {
  int nums[5]={1,2,3,4,5},i;
  printf("Before function call:\n");
  for(i=0;i<5;i++)
     printf("\t%d",nums[i]);
  change(nums); /* passing array nums to function */
  printf("\nAfter function call:\n");
  for(i=0;i<5;i++)
      printf("\t%d",nums[i]);
  getch();
  return 0;
}
```

Output:

```
Before function call:
1       2       3       4       5
After function call:
10      20      30      40      50
```

It can be illustrated with following diagramatical representation.



An array is referred to by its address, which is represented by the name of that array, used without subscripts.

Example 8.11

Write a program to read 10 numbers and reorders them in ascending order using function.

```
void sort(int []);
void readArray(int []);
void displayArray(int []);
int i;
int main()
 {
  int nums[10],j;
  printf("Enter 10 numbers:\n");
  readArray(nums);
  sort(nums);
  printf("\nThe numbers in ascending order as:\n");
  displayArray(nums);
  getch();
  return 0;
 }
 void sort(int num[])
     {
     int j,temp;
     for(i=0;i<9;i++)
        for(j=i+1;j<10;j++)
          {
          if(num[i]>num[j])
              {
               temp=num[i];
               num[i]=num[j];
               num[j]=temp;
               }
          }
     }
 void readArray(int nums[])
   {
    for(i=0;i<10;i++)
       scanf("%d",&nums[i]);
   }
void displayArray(int nums[])
   {
   for(i=0;i<10;i++)
      printf("\t%d",nums[i]);
   }
```

Output

```
Enter 10 numbers:
21  67  09  45  33  89  100  89  23  23
The numbers in ascending order as:
9   21  23  23  33  45  67  89  89 100
```

Example 8.12

Write a program to define three user defined functions for reading, processing and displaying of 2X3 matrix. Doube the matrix elements and display it.

```
#define m 2
#define n 3
void processMatrix(int [][n]);
void readMatrix(int [][n]);
void displayMatrix(int [][n]);
int i,j;
int main()
  {
    int matx[m][n];
    printf("Enter matrix of order %d*%d\n",m,n);
    readMatrix(matx);
    processMatrix(matx);
    printf("\nThe resultant matrix is:\n");
    displayMatrix(matx);
    getch();
    return 0;
  }
void processMatrix(int matrix[][n])
  {
    for(i=0;i<m;i++)
      for(j=0;j<n;j++)
        matrix[i][j]=2*matrix[i][j];
  }
void readMatrix(int matrix[][n])
    {
     for(i=0;i<m;i++)
       for(j=0;j<n;j++)
         {
          printf("\nmatrix[%d][%d]=",i,j);
          scanf("%d",&matrix[i][j]);
         }
      }
void displayMatrix(int matrix[][n])
    {
      for(i=0;i<m;i++)
         {
         for(j=0;j<n;j++)
             printf("%d\t",matrix[i][j]);
        printf("\n");
        }
    }
```

Output

```
Enter matrix of order 2*3
matrix[0][0]=3
matrix[0][1]=6
matrix[0][2]=7
matrix[1][0]=8
matrix[1][1]=2
matrix[1][2]=3
The resultant matrix is:
6       12      14
16      4       6
```

8.5 Arrays and Strings

String is an array of characters (i.e. characters are arranged one after another in memory). In other words, a character-array is also known as a string. The strings are used in C programming to manipulate text such as words and sentences. A string is always terminated by a NULL character (i.e \0). The terminating null character '\0' is important because it is the only way for string handling functions to know the end of string. Normally each character is stored in one byte of memory, and successive characters of the string are stored in successive bytes.

8.5.1 Initializing Strings

A string is initialized as:

```
char name[ ]={'R','A','M','\0'};
```

Here, the string name is initialized to RAM. This technique is also valid. But C offers special way to initialize the string as:

```
char name[]="RAM";   [it is same as char name[ ]={'R','A','M','\0'}; ]
```

The characters of the string are enclosed within a pair of double quotes. It is important to note down that all strings must end with a null character (\0), which has a numerical value of 0 (i.e. ASCII code).

Example 8.13

Write a program to illustrate string initialization.

```
#include<stdio.h>
#include<conio.h>
int main()
    {
     char name[]="Shyam Prasad Sharma";
     int i=0;
     printf("\nThe name in discrete form:\n");
     while(name[i]!='\0')
       {
       printf("\t%c",name[i]);
       i++;
```

```
    }
    getch();
    return 0;
}
```

Output

```
The name in discrete form:
S       h       y       a       m       P       r       a       s       a       d
S       h       a       r       m       a
```

Explanation: Here, the example checks the end of string using terminating character '\0'. C inserts the null character automatically at the end of string. Thus, it is not necessary to enter this character by the user.

8.5.2 Arrays of Strings

String is itself an array of characters. Thus, an array of strings is 2-D array of characters (i.e. array of character-arrays). The array of strings is declared as:

```
char string_name[size1][size2];
```

Example: `char st[2][5];`

It represents 2 strings with 5 chracters in each string.

Example 8.14

Write a program to read name of 5 different persons using array of strings and display them.

```
#include<stdio.h>
#include<conio.h>
int main()
    {
     char names[5][10];
     int i;
     printf("\nEnter name of 5 persons\n");
     for(i=0;i<5;i++)
         scanf("%s",names[i]);
     printf("\nThe names you have entered are:\n");
     for(i=0;i<5;i++)
     printf("%s\t",names[i]);
     getch();
     return 0;
    }
```

Output

```
    Enter name of 5 persons
    Ram
    Hari
    Krishan
    Gopal
    Sita
The names you have entered are:
    Ram     Hari    Krishan Gopal   Sita
```

8.5.3 String Handling Functions

The library or built-in functions `strlen()`, `strcpy()`, `strcat()`, `strcmp()`, `strrev()` etc are used for string manipulation in C. These functions are defined in header file `string.h`.

i. strlen() function

The function `strlen()` receives a string as argument and returns an integer which represents the length of string passed. The length of a string is the number of characters present in it, excluding the terminating null character. Its syntax is:

```
integer_variable= strlen(string);
```

Example 8.15

Write a program to find out length of string input from user using library function `strlen()`.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
  {
  char name[30];
  int seq, len;
  printf("Enter your name:");
  gets(name);
  len=strlen(name);
  printf("The Name you entered is:%s, have length: %d", name,len);
  printf("\nDisplaying each name character with address & ASCII:\n");
  for(seq=0;seq<len;seq++)
      printf("addr: %5u\t char='%c'=%3u\n",
   &name[seq],name[seq],name[seq]);
  return 0;
  }
```

Output

```
Enter your name:Michael
The Name you entered is:Michael, have length: 7
Displaying each name character with address & ASCII:
addr: 2293498     char='M'= 77
addr: 2293499     char='i'=105
addr: 2293500     char='c'= 99
addr: 2293501     char='h'=104
addr: 2293502     char='a'= 97
addr: 2293503     char='e'=101
addr: 2293504     char='l'=108
```

ii. strcpy() function

The `strcpy()` function copies one string to another. The function accepts two strings as parameters and copies the `second` string into the `first` string character by character including the null character of the second string. Its syntax is:

```
strcpy(destination_string, source_string);
```

i.e. `strcpy(s1,s2)` means the content of `s2` is copied to `s1`.

Example 8.16

Write a program to copy one string to another using strcpy() function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
    {
    char name[]="Saroj Bhatta",s[15];
    strcpy(s,name);
    printf("\nThe value in two strings:\n s=%s \t name=%s",s,name);
    getch();
    return 0;
    }
```

Output

```
The value in two strings:
s=Saroj Bhatta          name=Saroj Bhatta
```

iii. strcat() function

The `strcat()` function concatenates two strings (i.e. it appends one string at the end of another). The function accepts two strings as parameters and stores the contents of the `second` string at the end of the `first`. Its syntax is:

```
strcat(string1, string2);// 
```
i.e. it is equivalent to `string1=string1+string2;`

Example 8.17

Write a program to concatenate two strings using strcat() function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
    {
    char firstName[20]="Saroj",lastName[]="  Bhatta";
    strcat(firstName,lastName);
    printf("\nThe full name is %s",firstName);
    return 0;
    }
```

Output

```
The full name is Saroj Bhatta
```

iv. strcmp() function

The `strcmp()` function compares two strings to find out whether they are same or different. This function is useful for constructing and searching strings as arranged in a dictionary. The function accepts `two strings` as parameters and returns an `integer` whose value is

i) less than 0 (or -1 in some compiler) if the first string is less than the second

ii) equal to 0 if the both are same

iii) greater than 0 (or 1 in some compiler) if the first string is greater than the second

The two strings are compared character by character until there is a mismatch or end of one of stings is reached. Whenever two characters in two strings differ, the string which has the character with a higher ASCII value is treated as greater string. For example, consider two strings "ram" and "rajesh". The first two characters are same but third character in string ram and that is in rajesh are different. Since ASCII value of third character m in string ram is greater than that of j in string rajesh, the string ram is greater than rajesh.

Its syntax is:

```
integer_variable= strcmp(string1, string2);
```

Example 8.18

Write a program to illustrate the use of strcmp() function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
{
char name1[15],name2[15];
int diff;
printf("Enter first string\t:");
gets(name1);
printf("Enter second string\t:");
gets(name2);
diff=strcmp(name1,name2);
if(diff>0)
   printf("%s is greater than %s by value %d",name1, name2, diff);
else if(diff<0)
   printf("%s is greater than %s by value %d",name2,name1,diff);
else
    printf("%s is same as %s ",name1,name2);
getch();
return 0;
}
```

Output

```
a) Enter first string      :ram
   Enter second string     :rajesh
   ram is greater than rajesh by value 3
b) Enter first string      :Hello
   Enter second string     :hello
   hello is greater than Hello by value -32
c) Enter first string      :Binod
   Enter second string     :Binod
   Binod is same as Binod
```

v. strrev() function

The function `strrev()` is used to reverse all characters in a string except null character at the end of string. The reverse of string "abc" is "cba". Its syntax is

```
strrev(string);
```

For example: `strrev(s)` means it reverses the characters in string `s` and stores reversed string in `s`. Thus, the original string is reversed.

Example 8.19

Write a program to illustrate `strrev()` function.

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
int main()
    {
     char name[15]="Shyam Shrama",name2[15];
     strcpy(name2,name);
     strrev(name);
     printf("The reversed string of original string %s is %s",name2,name);
     getch();
     return 0;
    }
```

Output

```
The reversed string of original string Shyam Shrama is amarhS mayhS
```

vi. strncat() function

The function appends one string into another string upto specified number of characters.

Syntax: `char *strncat(char *str1, const char *str2, size_t n);`

It appends string `str2` into end of string `str1` upto n characters long of `str1`.

vii. strchr() function

Syntax: `char *strchr(const char *str, int c);`

It searches for the first occurance of the character `c` in the string `str`. The function returns a pointer pointing to the first matching character or NULL if not matched.

viii. strrchr() function

Syntax: `char *strrchr(const char *str, int c);`

It searches for the first occurance of the character `c` beginning at the rear end of string `str`.

ix. strncmp() function

Syntax: `int strncmp(const char *str1, const char *str2, size_t n);`

It searches at most the first `n` bytes of `str1` and `str2` and the function returns `zero` if the first bytes of the strings are equal. It returns less than or greater than zero value according to `str1` is less than or greater than `str2`, respectively.

x. strstr() function

Syntax: `char *strstr(const char *str1, const char *str2);`

It finds the first occurance of the string `str2` in the string `str1`. It returns a pointer to the first ocurance of `str2` in `str1`. If no match is found, then a NULL pointer is returned.

xi. strspn() function

Syntax: `size_t strspn(const char *str1, const char *str2);`

It returns the index of the first character in `str1` that doesn't match any character in `str2`.

xii. strcspn() function

Syntax: `size_t strcspn(const char *str1, const char *str2);`

It returns the index of the first character in `str1` that matches any of the characters in `str2`.

xiii. strtol() function

Syntax: `long strtol(const char *str1, char **end, int base);`

It converts the string pointed by `str1` to long value. It skips leading white space and stops when it encounters the non-numeric character. It stores the address of the first invalid character in `str1` in `*end`. We may pass `NULL` instead of `*end`. If we do not want to store the invalid characters anywhere. The argument base specifies whether the number is in hexdecimal, octal, binary or decimal representation.