C-07 Strings in C

An array of characters has a special place in C. **It is used to denote a string** i.e. a word or a sentence or any text. C does not have any special data type for strings and **a string is treated as a collection of characters** with a special marker to denote the end of a particular string.

Just like initialising an integer or a float array, a character array can also be initialised as shown below:

```
char name[6]={'S','a','y','a','n','\0'};
```

In the above initialisation, several points need to be noted. As said earlier, when an array is initialised, usually there is no need to write the index of the array, as the array automatically assumes the index of the number of initialised elements. Moreover you may notice that an **extra special character '\0'** has been introduced at the end of the name Sayan. The character '\0' is a single character called a '**null indicator**' and is used to mark the end of a string. The 'null indicator' should be placed at the end of the set of characters to tell C that a particular string, consisting of several characters, has ended. Without the null indicator it would not be possible to demarcate two or more consecutive strings. One important thing should be kept in mind. While declaring a character array for holding strings, **always remember to keep enough space to include the null terminator**. Thus if a string consists of **n** characters, always declare the array to consist of **n+1** characters as **A[n+1]** and **not as** A[n]. In the latter case the character set will **not** be treated as a string as there would be no space to accommodate the null indicator.

Since an array of characters has a special place in C, it is not surprising that it can be also initialised in a more **convenient manner** as shown below:

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

The above declaration with the name Sayan between two double quotes " " is used to initialise the character array with a string. The double quote indicates that this character array consists of a string and C **automatically inserts the null indicator '\0' after the last character** in the string (note that in the above declaration the index of the array has not been indicated. In case you indicate the index please remember to make it **one more** than the number of characters in the string).

Remember one more thing. If you initialise a character array using single quotes as 'c', then the character is treated as a char type variable and if you initialise it using double quotes as in "c" then it is treated as a string and the null indicator is automatically inserted at the end of the character.

Note: Unlike other type of variables, **an array can be initialised only during the declaration**. C does not allow directly assigning one array to another. Thus the code shown below is **illegal in C**.

```
char name[20];
name = "Sayan";
```

To assign a constant string (like the one shown above) to an array, special functions like **strcpy()** are required in C, as will be shown later.

7.1 Entering and displaying Strings:

Apart from the initialisation, a string can be also input by the user using library functions like **scanf()**, **gets()** or **getchar()**. Similarly we can display a string using the **printf()** function and the **%s** format specifier (**s** for **s**tring), just like displaying any other variable. The following program inputs a string using the **scanf()** function and the **%s** format specifier and determines the number of letters in the string.

```
#include <stdio.h>
#define MAX 20
int main()
{
  int count=0;
  char word[MAX];
  printf("\nEnter a word to determine its length\n");
  scanf("%s", word);          → the string is input using the %s specifier into the array named word

  while(word[count++]);       → the while loop proceeds as long as the contents of word[] ≠ '\0'

  printf("\nThe number of letters in the word = %d", count-1);
  return (0);
}
```

Several points need to be noted in the simple program given above.

First note that **no '&' has been used in the scanf() function** before the variable name. When inputting a string

using **scanf()** and the **%s** specifier there is no need to use the address operator '**&**', as by default the string is stored at the address pointed to by the first element of the array.

Second, note that the variable array **word[]** inside the **scanf()** function **does not include** the square brackets **[]** and only the name **word** is used. This is because for an array, the **name without the square brackets represents the address of the variable** i.e. the address of the first element of the array, and this is what is required by the **scanf()** function as its second argument.

Thirdly, the **%s** specifier **automatically inserts** the null indicator '**\0**' after the entered string.

The **while** loop in the above program is used without any statement following it and may seem strange at first sight. The variable **count** is used to store the count of the number of letters in the word and is been initialised to '0'. The condition of the **while** loop checks the contents of each element of the character array. Initially the value of count is '0' and hence checks the first character of word i.e. **word[0]**. With each pass, the counter gets incremented by 1 by **count++** and the condition subsequently checks for the consecutive values of word as **word[1]**, **word[2]** etc. Any character other than '**\0**' will give a non-zero element and hence will make the **while** loop true and let it continue. When the end of the string is reached, the last array element will be '**\0**' which has a value of '0' and will finally make the **while** loop false and terminate the loop. Thus the final value of count will contain the total number of letters in the string including the null indicator '**\0**' which is not required. Hence the final value displayed using the **printf()** function is one less than the value of **count** to eliminate the count value generated by the null indicator.

The **%s** specifier is perfect as long as we enter a single word. But if we enter multiple words separated by blanks then the string input will not be the same as entered. The %s specifier will read only upto the character before the first blank appears. Thus if someone enters the name "**Sayan Chakraborty**" then only "**Sayan**" will get stored in the character array.

To overcome this problem we use another function, namely the **gets()** function to input a multiple word string. The **gets()** function terminates the input when it receives a newline character '**\n**' (i.e. the Enter key) as input from the user and stores the string by replacing the newline character '**\n**' with a null indicator '**\0**' to mark the end of the string. The argument of the **gets()** function is the array name to input the string. (For 1d array no square bracket is necessary, for 2d array row index to be given within [])

The program below inputs a string using the **gets()** function and determines the number of positions where there is an occurrence of the letter '**e**' in the sentence. (The program with a little modification can also be used to determine the number of words or letters in the string also. Remember that a new word is indicated by a blank followed by any character, except for the first word).

```
#include <stdio.h>
#define MAX 80
int main()
{
  int count=0, i=0;
  char text[MAX];
  printf("\nEnter a sentence to find the number of 'e' in it\n");
  gets(text);                  → No square bracket is used in the actual argument i.e. only text used
  do{ if(text[i]=='e'||text[i]=='E')
        count++;
    } while(text[i++]!='\0');      → While condition continues till text[i] is not equal to '\0'
    printf("\n\nThe number of 'e' in the given text = %d", count);
  return (0);
}
```

From the above program it is seen that the string is input into the array named **text[]**. The **argument** of **gets()** is the name of the array i.e. **text without any square brackets**. Similar to the previous problem, the condition in the **while** loop continues as long as the array element is not a '**\0**'. The index counter **i** is incremented after testing for each array element, while the counter **count** gets incremented whenever the **if** condition is satisfied.

In the above example however we do not have any control over the number of characters entered by the user. In case the user enters more than 80 characters then the remaining characters may get overwritten on some memory locations reserved for some other variables, causing serious programming error or a program crash. In order to overcome this situation we can modify our above program to read one character at a time and as the characters are being read we can simultaneously check the count and make the program stop whenever the input exceeds the

maximum count. To do this we have the **getchar()** function at our disposal. The program given below is the same as the previous one, used to count the number of letters in a word. However here we are using the **getchar()** function to read the letters one by one and terminate in case the user enters more than the maximum specified number of letters.

```
include <stdio.h>
#define MAX 81

int main()
{
  char text[MAX], letter;
  int count=0;

  printf("\nEnter any String less than %d characters\n", MAX);

  while( (letter=getchar())!='\n' && (count<MAX-1) )
       text[count++]=letter;

  if(letter=='\n')          → The new line '\n' is replaced by a null '\0' to mark end of string
       text[count]='\0';
  else
    {printf("\nToo many characters. Try again");
     return (0);
    }

  printf("\nThe sentence \"%s\" has %d characters", text, count);
  return (0);
}
```

Let us now analyse how the above program works. The condition part of the **while** loop is used to input the characters as well as check for the newline character '**\n**'. Moreover the condition part also checks the number of characters entered, by checking the value of the variable **count**. The value of **count** should be less than (**MAX-1**) and not **MAX**, as enough space should be kept for including the '**\0**' at the end of the entered string. When either the user enters a newline (by pressing the Enter key) or the count exceeds (**MAX-1**), the **&&** operator makes the condition inside the loop false and terminates the **while** loop. But before the loop terminates, the character '**\n**' is already assigned to the variable **letter** vide the code segment **letter=getchar()**. Hence when the user presses 'Enter', the condition of the following **if** statement becomes true and the null terminator '**\0**' is assigned to the last character space in the **text[]** array, thus indicating the end of the string. The last value of **count** indicates the last character entered and is used to display the result using the **printf()** function. Note that the string is also displayed by the **printf()** using the **%s** specifier. The corresponding variable **text[]** is placed in the **printf() without the square brackets** as in the **scanf()** of the last example. In case the value of **count** becomes equal to (**MAX-1**), the **while** loop terminates and **else** statement is executed.

7.2 Two dimensional arrays and Strings:

As we have seen, a 1-dimension character array was used to store a single string. The **C language has the ability to treat parts of arrays as arrays**. Thus each row of a 2-dimensional array can be thought of as a 1-dimensional array. Thus a 2-dimensional character array can be used to store multiple strings. The **first dimension defines the number of strings** and the **second dimension indicates the maximum number of characters in each string**.

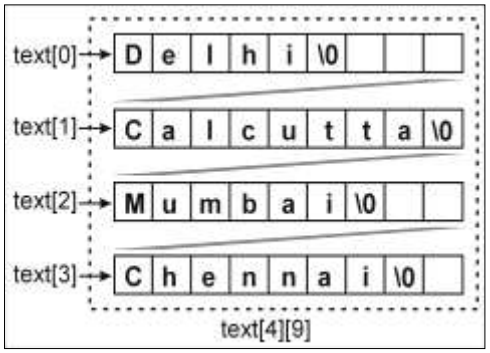
```
char text[4][10] = {   {'D', 'e', 'l', 'h', 'i', '\0'},
                       {'C', 'a', 'l', 'c', 'u', 't', 't', 'a', '\0'},
                       {'M', 'u', 'm', 'b', 'a', 'i', '\0'},
                       {'C', 'h', 'e', 'n', 'n', 'a', 'i', '\0'}
                   };
```

The above declaration indicates a two dimensional character array **text[]**, which can hold 4 strings, with each string containing a maximum of 8 characters (the 9th character is reserved for '\0' to mark the end of each string). Since each word is not exactly 8 characters long (excepting "Calcutta"), the remaining elements in each row of the above array are automatically assigned zero values.

But we have seen earlier that an array of characters to represent a string can be more conveniently written by writing the string within double quotes (" "). We give below the same string initialisation code shown above but using double quotes to enclose the strings.

```
char text[4][9] = {"Delhi", Calcutta", "Mumbai", "Chennai"};
```

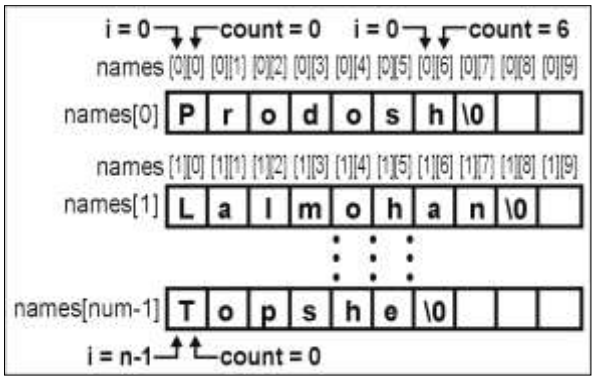
The above code automatically assigns each string to each row of the **text[]** array and inserts the null terminator '\0' at the end of each string. An interesting property of the above initialisation is that **each string can be treated as a single element of a one dimensional array** consisting of 4 elements. Thus each of the 4 strings can be referred to by the row number of the 2-dimensional array as:



text[0]→{"Delhi"}
text[1]→{"Calcutta"}
text[2]→{"Mumbai"}
text[3]→{"Chennai"}

The following program reads several strings and then calculates the length of each string (without using any library functions)

```
#include <stdio.h>
#define LEN 10        → Defines the maximum length of each word
#define MAX 20        → Defines the maximum number of words

int main()
{
  int num, i=0, count=0;
  char names[MAX][LEN], length[MAX];

  printf("\nEnter the number of words to input (Maximum %d): ", MAX);
  scanf("%d", &num);              → Enters the total no. of words (≤ MAX) into count

  /* String input for loop */
  for(i=0; i<num; i++)
      { printf("\nEnter name%d: ",(i+1));
        scanf("%s", names[i]);    → Enters each word into the array names[] in its i th row.
      }

  /* Counting for loop */
  for(i=0; i<num; i++)
      { while(names[i][count++]); → Condition remains true until names[i][count]='\0'
        length[i]=count-1;        → The count is assigned to the array length[] in position i
        count=0;                  → The count is again initialised to 0 for the next word
      }

  /* Display for loop */
  for(i=0; i<num; i++)
      printf("\nThe length of the word \"%-9s\" is %d", names[i], length[i]);

  return (0);
}
```



The words to be entered are stored in the array **names[MAX][LEN]** which can hold a maximum of 20 words (**MAX=20**) with each word containing a maximum of 10 characters (**LEN=10**). Based on this declaration the compiler will reserve **MAX** x **LEN**=200 bytes of memory area to store the words in the array **names[]**. As discussed earlier, during input only the first index of the 2-dimensional array is specified in the **scanf()** function as **names[i]** to indicate the starting position of each string in the array. This is done as each string will be treated as a single element of a 1-dimensional array **names[MAX]**.

After the strings are entered, a **for** loop is used to read the strings one by one and then count the number of characters in each. For each **names[i]**, the second index **[count]** indicates each character in that particular string, as shown ij the diagram above. The counting of the characters is done by using the **while** loop. The condition of the **while** loop checks for the presence of the null indicator **'\0'** at the position **names[i][count]**. When the condition encounters a **'\0'**, it terminates the loop and assigns the

value stored in **count** (minus 1, to cancel the last **++** operation on **count**) to the **i**th position of the array **length[]** to indicate the length of the **i**th string.

Finally the third for loop is used to print the output i.e. the entered words and their lengths. Note how the format specifier %s is used to align and print the words. The code **%-9s** indicates that 9 character spaces will be reserved for each string and the (-) sign before the 9 indicates that the string will be printed left aligned in the reserved space. A sample run of the above program is given below to clarify these points:

```
Enter the number of words to input (Maximum 20): 3
Enter Name1: Arthur
Enter Name2: Edgar
Enter Name3: Hitchcock
The length of the word "Arthur   " is 6
The length of the word "Edgar    " is 5
The length of the word "Hitchcock" is 9
```

7.3 Library functions available to handle Strings:

Just like different library functions that are available to handle mathematical operations, C provides us with several library functions to handle strings. String handling includes common tasks like copying two strings [**strcpy()**], concatenating, i.e. joining two strings [**strcat()**], finding the length of a string [**strlen()**], comparing two strings [**strcmp()**], etc. There are other string handling functions but we will restrict ourselves to the most commonly used above four. To use these library functions you have to include the header file **<string.h>**.

- The **strcat()** function is used to join two strings i.e. one string gets appended or joined to the other. The syntax for the function is:

```
strcat(string1, string2);
```

Both **string1** and **string2** are character arrays and the second string i.e. **string2** gets added after **string1**. Hence care should be taken so that string1 is declared with sufficient size to contain both **string1** and **string2**. The **strcat()** function works by replacing the null indicator '**\0**' at the end of the **string1** with the starting of the second string. The following example concatenates three strings: **wish**, **first_name** and **last_name**. The string **wish** is initialised to a constant string "**Good Morning** ". The remaining strings i.e. **first_name** and **last_name** are getting added to this constant string to form a final greeting. Hence the character array **wish** has been assigned sufficient space to hold the final string.

```
#include <stdio.h>
#include <string.h>
#define MAX 10

int main()
{
  char first_name[MAX], last_name[MAX];
  static char wish[40]="Good Morning ";

  printf("\nEnter first name: ");
  scanf("%s", first_name);
  printf("\nEnter last name: ");
  scanf("%s", last_name);

  strcat(strcat(wish, first_name), " ");
  strcat(wish, last_name);

  printf("\n\"%s\"", wish);
  return (0);
}
```

The first string is entered into **first_name** and the second string is entered into **last_name**. Finally the **strcat()** function is used thrice to form the final string. The first two **strcat()** functions are used in a nested manner. The inner **strcat()** function adds **first_name** to **wish**, while the outer **strcat()** adds a blank " " to the combined string of **wish** and **first_name**. The blank is added to separate the two names in the final string. The **strcat()** in the next line then adds the second string i.e. **last_name** to the combination to create the final string. (If anyone wishes, all the three **strcat()** can be nested together to form the final string, but the code may look clumsy in that case). A sample run of the program is given below:

```
Enter first name: Samasish
```

```
Enter last name: Sengupta

"Good Morning Samasish Sengupta"
```

Note that a blank space has been automatically introduced between the two entered strings **first_name** and **last_name**. Also note the double quotes used to enclose and print the final string in the **printf()** function by using the special character **\"** to enclose the **%s** specifier. If only the double quote were used without the back slash like **"**, then the **printf()** function would have interpreted it wrongly for obvious reasons.

- The **strcpy()** function is used to copy one string into another. The syntax for the function is:

```
strcpy(string1, string2);
```

The function copies the contents of the second string i.e. **string2** into the first string i.e. **string1**. The second string can be either a variable array or can be a constant string. The following program is used to exchange two strings using the **strcpy()** function.

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

#define MAX 10

int main()
{
  char first_name[MAX], last_name[MAX], temp[MAX];

  printf("\nEnter first name: ");
  scanf("%s", first_name);
  printf("\nEnter last name: ");
  scanf("%s", last_name);

  strcpy(temp, first_name);
  strcpy(first_name , last_name);
  strcpy(last_name, temp);

  printf("\n\nThe name is %s %s", first_name, last_name);
  return (0);
}
```

Though the above program is of no special significance, but it illustrates the general procedure to exchange two string variables using the **strcpy()** function. First the string **first_name[]** is copied to the temporary array variable **temp[]** (to save it) using the first **strcpy()** function. Then **last_name** is copied to **first_name** using the second **strcpy()** function. Finally the string stored in **temp[]** is assigned to **last_name** using the third **strcpy()** function. Note how the temporary character array **temp[]** is used to store the contents of **first_name**.

- The **strlen()** function is used to count the number of characters in a string, excluding the null terminator. As seen from the above declaration, it takes only one single string as its argument and **returns an integer** value indicating the length of the string. The argument can be a variable character array or a string constant. The syntax for the function is:

```
strlen(string1);
```

The following program checks each string entered by the user for its length. Only if the string is within the allowable length as specified in the character array declaration **names[][]**, it assigns the string to the **names[][]** array. Otherwise the program prompts to re-enter the string. A temporary intermediate array **temp[]** is used to store the string for checking. The index of **temp[]** has been made sufficiently large to hold a larger string. Apart from the **strlen()** function, the program also uses the **strcpy()** function.

```
#include <stdio.h>
#include <string.h>
#define NUMBER 10
#define LEN 15

int main()
{
  char names[NUMBER][LEN], temp[80];
  int num, i;

  printf("\nEnter the number of names to input (<%d): ", NUMBER);
  scanf("%d", &num);
```

```
  for(i=0; i<num; i++)
     { printf("\nName%d: ", (i+1));
       gets(temp);

       while(strlen(temp)>(LEN-1))
            {printf("\nEnter string with <%d characters. Try again:", LEN);
             printf("\nName%d: ", (i+1));
             gets(temp);
            }

       strcpy(names[i], temp);
     }

  for(i=0; i<num; i++)
     printf("\nName%d is %s", (i+1), names[i]);

  return (0);
}
```

- The **strcmp()** function is used to compare two strings. The syntax for the function is:

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

There are three possibilities while comparing two strings. These are:

1. When both the strings are same or **equal**, the function returns **0** (and not 1, remember this carefully as the inverse of this value can be used in the condition of a **if** statement to test two strings to be same).
2. When **string1** is in a **higher** alphabetical order than **string2**, the function returns a **negative** number.
3. While if **string1** is in a **lower** alphabetical order than **string2**, the function returns a **positive** number. Thus by checking for the sign of the returned value one can determine whether **string1** is equal, higher or lower than **string2**.

The logic of the **strcmp()** function may at first seem confusing. But the way of working of the function will make things clear. The **strcmp()** function in reality finds the difference in the ASCII values of the first non-matching characters in the two strings. Thus if **string1** (say Su**j**oy) is at a higher alphabetical order than **string2** (say Su**m**on), then the first non-matching character '**j**' with ASCII value 106 in **string1** minus the first non-matching character '**m**' with ASCII value 109 in **string2** results in a negative number (**106-109=-3**).

We now give below a program to sort a string array in alphabetical order. The program is a complete one where we will be using the above functions to carry out the sorting. We will be using the **strcmp()** function to compare two strings and then the **strcpy()** function to interchange pairs of strings in case they are not in the alphabetical order. Moreover we will use the **strlen()** function to check for the character count for each input string.

```
/*Sorting of Strings by Selection Method*/
#include <stdio.h>
#include <string.h>
#define ROWMAX 10
#define COLMAX 15

int initialise(char [][COLMAX]);
void ascending(char [][COLMAX], int num);
void display(char names[][COLMAX], int num);


int main()
{ int word_num;
  char names[ROWMAX][COLMAX];

  printf("\n## ENTER the words (MAX %d), to Sort Alphabetically ##\n", ROWMAX);

  word_num = initialise(names);

  ascending(names, word_num);

  display(names, word_num);

  return (0);
}

int initialise(char names[][COLMAX])
{   int i=0;
```

```
    char temp[80];

    printf("\nPress \"ENTER\" after each word");
    printf(" ( ## Enter a \"*\" to terminate the list. ## )\n\n");

    do { if(i==ROWMAX)
            {printf("\nYou were about to exceeded the number of words!!\n");
             i++;
             break;}
         printf("Enter word no.%d: ",(i+1));
         scanf("%s", temp);

         while(strlen(temp)>(COLMAX-1))
            {printf("\nEnter string with <%d characters. Try again:", COLMAX);
             printf("\nEnter word no.%d: ", (i+1));
             scanf("%s", temp);
            }

         strcpy(names[i],temp);
       } while( strcmp(names[i++],"*"));

    printf("\nYou have entered %d words",--i);
    return i;
}

void ascending(char names[][COLMAX], int num)
{ int word=0, compare=0;
  char temp[COLMAX];

  for(word=0; word<num; word++)
     { for(compare=(word+1); compare<num; compare++)
          { if(strcmp(names[word],names[compare])>0)
             { strcpy(temp, names[word]);
               strcpy(names[word], names[compare]);
               strcpy(names[compare], temp);
             }
          }
      }
}

void display(char names[][COLMAX], int num)
{   int i=0;
    printf("\n\nThe Sorted Word List is given below\n");

    for(i=0; i<num; i++)
         printf("\n%s", names[i]);
}
```

Let us now analyse the above program and see how the different functions used carry out their tasks. Apart from **main()** and some library functions, three other user defined functions are used to carry out the task. The working of the different functions are given below:

1. **main()**: Here we have declared the character array '**char names[ROWMAX][COLMAX]**' to hold the strings input during the execution of the program. The array can hold **ROWMAX** number of strings with each string containing a maximum of (**COLMAX-1**) characters. Another variable '**word_num**' is declared to count the number of words input. **Main()** then calls the **initialise()** function to input the strings and count the number of words. The argument of the **initialise()** function is the array **names**. The function initialises the array and returns the number of words entered, which is stored in the variable **word_num**. Next the function **ascending()** is called and the array **names** and the word count **word_num** are passed to the function, which arranges the input strings in the ascending order. Finally **main()** calls the function **display()** with the arguments names and **word_count** to display the names in ascending order.
2. **int initialise(char names[][COLMAX])**: This function is used to initialise the character array **names[][]** by inputting the names into the different rows of the array. The maximum number of rows or strings that can be input is given by **ROWMAX** while the maximum number of characters per string is given by **COLMAX**. To terminate the input, the user has to enter the character "*****" as indicated in the input prompt. The **do-while** loop is used to input the strings. The condition at the end of the **do-while** loop i.e. '**while(strcmp(names[i++],"*"))**' checks whether the input string is "*****" in which case the **strcmp()**

returns '**0**' and terminates the loop. The **i++** operation increments the counter **i** each time the loop is executed and *after* comparing the two strings.

Two types of checking are done while inputting the strings. The first **if** statement checks whether the user has exceeded the maximum allowable rows. In that case it breaks from the **do-while** loop and stops further entry of strings.

The **scanf()** inputs the entered string into the temporary array **temp[]**. The **while** loop following **scanf()** then checks whether the entered string contains more than **COLMAX-1** number of characters. The **strlen()** function within the condition of the **while** loop is used to do this. In case the length is more than **COLMAX-1**, the loop asks the user to re-enter the string and continues till a proper string is entered. In case a proper length string is entered, it is copied from **temp[]** to **names[i]** using the **strcpy()** function. Finally the word count **i** is returned to **main()**.

3. **void ascending(char names[][COLMAX],int num)**: This function is used to sort the strings using the **selection sort** procedure. The function works in the same manner as the earlier program to sort an array of integers. Though the logic is the same but certain modifications are done to accommodate strings, as strings cannot be compared or assigned like integers.

   As usual we have declared a temporary array called **temp[]** to hold a string during the exchange. The first **for** loop takes each word by turn and compares it with all the remaining words using the second **for** loop. The **strcmp()** function is used to compare the two strings. If the first word (say monitor) is at a higher alphabetical order than the second word (say joystick), then the function returns a positive value (as m–j = 109 – 106 = 3 > 0) and satisfies the condition of the **if** statement and indicates that the strings need to be interchanged. In that case the **strcpy()** function is used to store the first word in **temp[]**. The next **strcpy()** then assigns the second string to the first and the final **strcpy()** reassigns the string stored in **temp** to the position of the second string. This process continues till all the strings are sorted.

4. **void display(char names[][COLMAX], int num)**: This is the last function called by **main()** and is used to display the strings in the sorted order. The **printf()** function is used to print each string stored in **names[i]** using the **for** loop.

A sample run of the above program is given below for reference. Note how the program responds when a word with more than the maximum allowable length is entered.

```
## ENTER the words (MAX 10), to Sort Alphabetically ##
Press "ENTER" after each word ( ## Enter a "*" to terminate the list. ## )

Enter word no.1: jaundice
Enter word no.2: influenza
Enter word no.3: diarrhoea
Enter word no.4: pneumonoultramicroscopicsilicovolcanoconiosis

Enter string with <15 characters. Try again:
Enter word no.4: pneumonia
Enter word no.5: tuberculosis
Enter word no.6: *

You have entered 5 words

The Sorted Word List is given below
diarrhoea
influenza
jaundice
pneumonia
tuberculosis
```

The following program is used to count the number of votes received by each candidate in an election. The number of candidates is used as an input and the number of voters is taken as constant say VOTERS. The logic of the program is that, the candidates are assigned identification numbers like 0, 1, 2, 3 etc. and the voter has to cast his vote by pressing the ID number of the candidate. Depending upon the ID number the vote count will get added against the respective candidate's ID number.

```
#include <stdio.h>
#define MAX 10
#define VOTERS 100

int main()
{
  int candidate[MAX]={0};
```

```
  int vote, invalid=0, n, v, i;
  printf("\nEnter the total number of candidates (<=%d): ", MAX);
  do{ scanf("%d", &n);
    } while(n>MAX||n<0);
  for(i=0; i<n; i++) printf("\nCandidate%d has ID number %d", i+1, i+1);
  printf("\n\nEnter total number of voters (<=%d): ", VOTERS);
  do{ scanf("%d", &v);
    } while(n>VOTERS||n<0);
  for(i=0; i<v; i++)
      { printf("Enter Vote%d: ", i+1);
        scanf("%d", &vote);
        if(vote>0&&vote<=n)
          candidate[vote-1]++;
        else
          invalid++;
      }
  printf("\nThe vote received by each candidate is given below:");
  for(i=0; i<n; i++)
      printf("\nVote received by Candidate%d is %d", i+1, candidate[i]);
      printf("\nInvalid votes received = %d", invalid);
  return 0;
}
```

Output of the above program is given below:

```
Enter the total number of candidates (<=10): 3

Candidate1 has ID number 1
Candidate2 has ID number 2
Candidate3 has ID number 3

Enter total number of voters (<=100): 8

Enter Vote1: 2
Enter Vote2: 1
Enter Vote3: 1
Enter Vote4: 2
Enter Vote5: 3
Enter Vote6: 5
Enter Vote7: 1
Enter Vote8: 1

The vote received by each candidate is given below:
Vote received by Candidate1 is 4
Vote received by Candidate2 is 2
Vote received by Candidate3 is 1
Invalid votes received = 1
```