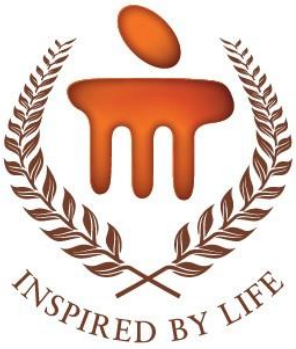


MASTER OF COMPUTER APPLICATIONS

SEMESTER 1

PROGRAMMING & PROBLEM-SOLVING USING C

Unit 7

Arrays and Strings

Table of Contents

1. INTRODUCTION

In the previous unit, you studied about the various types of storage classes that are used in C. You studied how those storage classes are used in different situations in C. In this unit, you will study about the arrays and strings. You will learn how arrays and strings are formed and manipulated.

Many applications require processing of multiple data items that have common characteristics. In such situations it is always convenient to place the data items into an array, where they will share the same name. An array is a collection of similar type of elements. All elements in the array are referred with the array name. Since arrays hold a group of data, it is very easy to perform looping and arithmetic operations on group of data. This chapter covers the processing of both one-dimensional and two-dimensional arrays.

1.1 Objectives

After studying this unit, you should be able to:

- *declare, initialize and process one-dimensional and two-dimensional arrays*
- *explain about strings and how to process them*
- *describe the library functions available in C to process strings*

2. ONE DIMENSIONAL ARRAYS

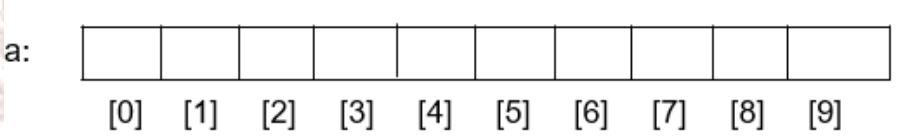
So far, we've been declaring simple variables: the declaration

int i;

declares a single variable, named i, of type **int**. It is also possible to declare an array of several elements. The declaration

int a[10];

declares an array, named a, consisting of ten elements, each of type **int.** Simply speaking, an array is a variable that can hold more than one value. You specify which of the several values you're referring to at any given time by using a numeric subscript. (Arrays in programming are similar to vectors or matrices in mathematics.) We can represent the array a above with a picture like this:



In C, arrays are *zero-based:* the ten elements of a 10-element array are numbered from 0 to 9. The subscript which specifies a single element of an array is simply an integer expression in square brackets. The first element of the array is a[0], the second element is a[1], etc. You can use these “array subscript expressions'' anywhere you can use the name of a simple variable, for example:

a[0] = 10;
a[1] = 20;
a[2] = a[0] + a[1];

Notice that the subscripted array references (i.e. expressions such as a[0] and a[1]) can appear on either side of the assignment operator.

The subscript does not have to be a constant like 0 or 1; it can be any integral expression. For example, it's common to loop over all elements of an array:

```
int i;
for(i = 0; i < 10; i = i + 1)
        a[i] = 0;
```

This loop sets all ten elements of the array a to 0.

Arrays are a real convenience for many problems, but there is not a lot that C will do with them for you automatically. In particular, you can neither set all elements of an array at once nor assign one array to another; both of the assignments

```
        a = 0;                  /* WRONG */
```

and

```
        int b[10];
        b = a;                  /* WRONG */
```

are illegal.

To set all of the elements of an array to some value, you must do so one by one, as in the loop example above. To copy the contents of one array to another, you must again do so one by one:

```
int b[10];
for(i = 0; i < 10; i = i + 1)
        b[i] = a[i];
```

Remember that for an array declared

```
int a[10];
```

there is no element a[10]; the topmost element is a[9]. This is one reason that zero-based loops are also common in C. Note that the for loop

```
for(i = 0; i < 10; i = i + 1)
  ...
```

does just what you want in this case: it starts at 0, the number 10 suggests (correctly) that it goes through 10 iterations, but the less-than comparison means that the last trip through the loop has i set to 9. (The comparison i <= 9 would also work, but it would be less clear and therefore poorer style.)

In the little examples so far, we've always looped over all 10 elements of the sample array a. It's common, however, to use an array that's bigger than necessarily needed, and to use a second variable to keep track of how many elements of the array are currently in use. For example, we might have an integer variable

```
int na;                /* number of elements of a[] in use */
```

Then, when we wanted to do something with a (such as print it out), the loop would run from 0 to na, not 10 (or whatever a's size was):

```
for(i = 0; i < na; i = i + 1)
        printf("%d\n", a[i]);
```

Naturally, we would have to ensure that na's value was always less than or equal to the number of elements actually declared in a.

Arrays are not limited to type **int;** you can have arrays of **char** or **double** or any other type.

Here is a slightly larger example of the use of arrays. Suppose we want to investigate the behavior of rolling a pair of dice. The total roll can be anywhere from 2 to 12, and we want to count how often each roll comes up. We will use an array to keep track of the counts: a[2] will count how many times we've rolled 2, etc.

We'll simulate the roll of a die by calling C's random number generation function, rand(). Each time you call rand(), it returns a different, pseudo-random integer. The values that rand() returns typically span a large range, so we'll use C's modulus (or "remainder'') operator % to produce random numbers in the range we want. The expression rand() % 6 produces random numbers in the range 0 to 5, and rand() % 6 + 1 produces random numbers in the range 1 to 6.

Program 7.1: Program to simulate the roll of a die

```
#include <stdio.h>
#include <stdlib.h>
main()
{
        int i;
        int d1, d2;
        int a[13]; /* uses [2..12] */

        for(i = 2; i <= 12; i = i + 1)
                a[i] = 0;

        for(i = 0; i < 100; i = i + 1)
                {
                d1 = rand() % 6 + 1;
                d2 = rand() % 6 + 1;
                a[d1 + d2] = a[d1 + d2] + 1;
                }

        for(i = 2; i <= 12; i = i + 1)
                printf("%d: %d\n", i, a[i]);

        return 0;
}
```

We include the header <stdlib.h> because it contains the necessary declarations for the rand() function. We declare the array of size 13 so that its highest element will be a[12]. (We're wasting a[0] and a[1]; this is no great loss.) The variables d1 and d2 contain the rolls of the two individual dice; we add them together to decide which cell of the array to increment, in the line

```
a[d1 + d2] = a[d1 + d2] + 1;
```

After 100 rolls, we print the array out. Typically, we'll see mostly 7's, and relatively few 2's and 12's.

2.1 Passing Arrays to Functions

An array name can be used as an argument to a function, thus permitting the entire array to be passed to the function. To pass an array to a function, the array name must appear by itself, without brackets or subscripts, as an actual argument within the function call. The corresponding formal argument is written in the same manner, though it must be declared as an array within the formal argument declarations. 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 specified within the formal argument declaration.

Program 7.2: The following program illustrates the passing of an array from the main to a function. This program is used to find the average of n floating point numbers.

```
#include<stdio.h>
main()
{
        int n, i;
        float avg;
        float list[100];
        float average(int, float[]); /* function prototype */
        printf("How many numbers:");
        scanf("%d",&n);
        printf(" Enter  the numbers:");
        for(i=1;i<=n;i++)
                scanf("%f", &list[i]);
        avg=average(n, list); /* Here list and n are actual arguments */
        printf("Average=%f\n", avg);
}
float average(int a, float x[ ])
{
        float avg;
        float sum=0;
        int i;
        for(i=0;i<a;i++)
                sum=sum+x[i]; /* find sum of all the numbers */
        avg=sum/a;                      /* find average */
        return avg;
}
```

SELF-ASSESSMENT QUESTIONS – 1

1. In C, an array subscript starts from _____
2. An array name is a pointer. (True/False)
3. Will there be a compilation error for the following program segment?(Yes/No)

```
int a[5] = {1, 2, 3, 4, 5};
int b[5] = {5, 4, 3, 2, 1};
int c[5][5];
…
c=a+b;
…
```

3. MULTIDIMENSIONAL ARRAYS

The C language allows arrays of any dimension to be defined. In this section, we will take a look at two-dimensional arrays. One of the most natural applications for a two-dimensional array arises in the case of a matrix. In C, the two-dimensional matrix can be declared as follows:

```
int array[3][6];
```

Following is the way of declaring as well as initializing two-dimensional arrays.

```
int array[3][6] = {
{4,5,6,7,8,9},
{1,5,6,8,2,4},
{0,4,4,3,1,1}
};
```

Such arrays are accessed like so:

```
array[1][4]= -2;
if (array[2][1] > 0) {
printf ("Element [2][1] is %d", array[2][1]);
}
```

Remember that, like ordinary arrays, two-dimensional arrays are numbered from 0. Therefore, the array above has elements from array[0][0] to array[2][5].

Program 7.3: Program to add two matrices

```
#include <stdio.h>
main()
{
        int a[5][5], b[5][5], c[5][5];
        int i, j, m, n;
        printf("Enter the order of the matrices:");
        scanf("%d%d", &m, &n);
        printf(" Enter the elements of A matrix:\n");
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        scanf("%d", &a[i][j]);
        printf("Enter the elements of B matrix:\n");
```

```
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        scanf("%d", &b[i][j]);
        /* Add the matrices */
        for(i=0;i<m;i++)
                for(j=0;j<n;j++)
                        c[i][j] = a[i][j]+b[i][j];
        /* Print the sum */
        printf("The sum of matrices:\n");
        for(i=0;i<m;i++)
        {
                for(j=0;j<n;j++)
                        printf("%d\t", c[i][j]);
                        printf("\n");
        }
}
```

Multidimensional arrays are processed in the same manner as one-dimensional arrays, on an element-by-element basis. However, some care is required when passing multidimensional arrays to a function. In particular, the formal argument declarations within the function definition must include explicit size specifications in all of the subscript positions except the first. These size specifications must be consistent with the corresponding size specifications in the calling program. The first subscript position may be written as an empty pair of square brackets, as with a one-dimensional array. The corresponding function prototypes must be written in the same manner. But while calling the function the array name may be passed as the actual argument as in the case of one-dimensional arrays. E.g:

```
void process_array (int [][6]); /* function prototype */
void process_array (int array[][6])/*function definition */ {
…
}
```

SELF-ASSESSMENT QUESTIONS – 2

4. In a two-dimensional matrix, the first subscript in the declaration specifies number of _____.
5. A two-dimensional array is considered as an array of one-dimensional arrays. (True/False)

4. STRINGS

Strings in C are represented by arrays of characters. The end of the string is marked with a special character, the *null character*, which is simply the character with the value 0. (The null character has no relation except in name to the *null pointer.* In the ASCII character set, the null character is named NULL.) The null or string-terminating character is represented by another character escape sequence, \0.

Because C has no built-in facilities for manipulating entire arrays (copying them, comparing them, etc.), it also has very few built-in facilities for manipulating strings.

In fact, C's only truly built-in string-handling is that it allows us to use *string constants* (also called *string literals*) in our code. Whenever we write a string, enclosed in double quotes, C automatically creates an array of characters for us, containing that string, terminated by the \0 character. For example, we can declare and define an array of characters, and initialize it with a string constant:

```
char string[ ] = "Hello, world!";
```

In this case, we can leave out the dimension of the array, since the compiler can compute it for us based on the size of the initializer (14, including the terminating \0). This is the only case where the compiler sizes a string array for us, however; in other cases, it will be necessary that we decide how big the arrays and other data structures we use to hold strings are.

To do anything else with strings, we must typically call functions. The C library contains a few basic string manipulation functions, and to learn more about strings, we'll be looking at how these functions might be implemented.

Since C never lets us assign entire arrays, we use the **strcpy** function to copy one string to another:

```
#include <string.h>
char string1[ ] = "Hello, world!";
char string2[20];
strcpy(string2, string1);
```

The destination string is **strcpy's** first argument, so that a call to **strcpy** mimics an assignment expression (with the destination on the left-hand side). Notice that we had to allocate string2 big enough to hold the string that would be copied to it. Also, at the top of any source file where we're using the standard library's string-handling functions (such as strcpy) we must include the line

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

which contains external declarations for these functions.

Since C won't let us compare entire arrays, either, we must call a function to do that, too. The standard library's **strcmp** function compares two strings, and returns 0 if they are identical, or a negative number if the first string is alphabetically “less than'' the second string, or a positive number if the first string is “greater.'' (Roughly speaking, what it means for one string to be “less than'' another is that it would come first in a dictionary or telephone book, although there are a few anomalies.) Here is an example:

```
char string3[] = "this is";
char string4[] = "a test";
if(strcmp(string3, string4) == 0)
printf("strings are equal\n");
else printf("strings are different\n");
```

This code fragment will print “strings are different''. Notice that **strcmp** does not return a Boolean, true/false, zero/nonzero answer, so it's not a good idea to write something like

```
if(strcmp(string3, string4))
    ...
```

because it will behave backwards from what you might reasonably expect. (Nevertheless, if you start reading other people's code, you're likely to come across conditionals like if(strcmp(a, b)) or even if(!strcmp(a, b)). The first does something if the strings are unequal; the second does something if they're equal. You can read these more easily if you pretend for a moment that **strcmp's** name were **strdiff,** instead.)

Another standard library function is **strcat,** which concatenates strings. It does not concatenate two strings together and give you a third, new string; what it really does is append one string onto the end of another. (If it gave you a new string, it would have to allocate memory for it somewhere, and the standard library string functions generally never do that for you automatically.) Here's an example:

```
char string5[20] = "Hello, ";
char string6[] = "world!";
printf("%s\n", string5);
strcat(string5, string6);
printf("%s\n", string5);
```

The first call to **printf prints** “Hello, '', and the second one prints “Hello, world!'', indicating that the contents of string6 have been tacked on to the end of string5. Notice that we declared string5 with extra space, to make room for the appended characters.

If you have a string and you want to know its length (perhaps so that you can check whether it will fit in some other array you've allocated for it), you can call **strlen**, which returns the length of the string (i.e. the number of characters in it), not including the \0:

```
char string7[ ] = "abc";
int len = strlen(string7);
printf("%d\n", len);
```

Finally, you can print strings out with printf using the %s format specifier, as we've been doing in these examples already (e.g. printf("%s\n", string5);).

Since a string is just an array of characters, all of the string-handling functions we've just seen can be written quite simply, using no techniques more complicated than the ones we already know. In fact, it's quite instructive to look at how these functions might be implemented. Here is a version of **strcpy:**

```
mystrcpy(char dest[ ], char src[ ])
{
int i = 0;
while(src[i] != '\0')
        {
        dest[i] = src[i];
        i++;
        }
dest[i] = '\0';
}
```

We've called it mystrcpy instead of strcpy so that it won't clash with the version that's already in the standard library. Its operation is simple: it looks at characters in the src string one at a time, and as long as they're not \0, assigns them, one by one, to the corresponding positions in the dest string. When it's done, it terminates the dest string by appending a \0. (After exiting the while loop, i is guaranteed to have a value one greater than the subscript of the last character in src.) For comparison, here's a way of writing the same code, using a for loop:

```
for(i = 0; src[i] != '\0'; i++)
        dest[i] = src[i];
dest[i] = '\0';
```

Yet a third possibility is to move the test for the terminating \0 character out of the for loop header and into the body of the loop, using an explicit if and break statement, so that we can perform the test after the assignment and therefore use the assignment inside the loop to copy the \0 to dest, too:

```
for(i = 0; ; i++)
        {
        dest[i] = src[i];
        if(src[i] == '\0')
                break;
        }
```

(There are in fact many, many ways to write strcpy. Many programmers like to combine the assignment and test, using an expression like (dest[i] = src[i]) != '\0')

Here is a version of strcmp:

```
mystrcmp(char str1[ ], char str2[ ])
{
int i = 0;

while(1)
        {
        if(str1[i] != str2[i])
                return str1[i] - str2[i];
        if(str1[i] == '\0' || str2[i] == '\0')
                return 0;
        i++;
        }
}
```

Characters are compared one at a time. If two characters in one position differ, the strings are different, and we are supposed to return a value less than zero if the first string (str1) is alphabetically less than the second string. Since characters in C are represented by their numeric character set values, and since most reasonable character sets assign values to characters in alphabetical order, we can simply subtract the two differing characters from each other: the expression str1[i] - str2[i] will yield a negative result if the i'th character of str1 is less than the corresponding character in str2. (As it turns out, this will behave a bit strangely when comparing upper and lower-case letters, but it's the traditional approach, which the standard versions of strcmp tend to use.) If the characters are the same, we continue around the loop, unless the characters we just compared were (both) \0, in which case we've reached the end of both strings, and they were both equal. Notice that we used

what may at first appear to be an infinite loop--the controlling expression is the constant 1, which is always true. What actually happens is that the loop runs until one of the two return statements breaks out of it (and the entire function). Note also that when one string is longer than the other, the first test will notice this (because one string will contain a real character at the [i] location, while the other will contain \0, and these are not equal) and the return value will be computed by subtracting the real character's value from 0, or vice versa. (Thus the shorter string will be treated as “less than'' the longer.)

Finally, here is a version of strlen:

```
int mystrlen(char str[ ])
{
int i;
for(i = 0; str[i] != '\0'; i++)
        { }
return i;
}
```

In this case, all we have to do is find the \0 that terminates the string, and it turns out that the three control expressions of the for loop do all the work; there's nothing left to do in the body. Therefore, we use an empty pair of braces { } as the loop body. Equivalently, we could use a *null statement*, which is simply a semicolon:

```
for(i = 0; str[i] != '\0'; i++)
    ;
```

Everything we've looked at so far has come out of C's standard libraries. As one last example, let's write a substr function, for extracting a substring out of a larger string. We might call it like this:

```
char string8[ ] = "this is a test";
char string9[10];
substr(string9, string8, 5, 4);
printf("%s\n", string9);
```

The idea is that we'll extract a substring of length 4, starting at character 5 (0-based) of string8, and copy the substring to string9. Just as with strcpy, it's our responsibility to declare the destination string (string9) big enough. Here is an implementation of substr. Not surprisingly, it's quite similar to strcpy:

```
substr(char dest[ ], char src[ ], int offset, int len)
{
int i;
for(i = 0; i < len && src[offset + i] != '\0'; i++)
        dest[i] = src[i + offset];
dest[i] = '\0';
}
```

If you compare this code to the code for mystrcpy, you'll see that the only differences are that characters are fetched from src[offset + i] instead of src[i], and that the loop stops when len characters have been copied (or when the src string runs out of characters, whichever comes first).

When working with strings, it's important to keep firmly in mind the differences between characters and strings. We must also occasionally remember the way characters are represented, and about the relation between character values and integers.

As we have had several occasions to mention, a character is represented internally as a small integer, with a value depending on the character set in use. For example, we might find that 'A' had the value 65, that 'a' had the value 97 and that '+' had the value 43. (These are, in fact, the values in the ASCII character set, which most computers use. However, you don't need to learn these values, because the vast majority of the time, you use character constants to refer to characters, and the compiler worries about the values for you. Using character constants in preference to raw numeric values also makes your programs more portable.)

As we may also have mentioned, there is a big difference between a character and a string, even a string which contains only one character (other than the \0). For example, 'A' is not the same as "A". To drive home this point, let's illustrate it with a few examples.

If you have a string:

char string[ ] = "hello, world!";

you can modify its first character by saying

```
string[0] = 'H';
```

(Of course, there's nothing magic about the first character; you can modify any character in the string in this way. Be aware, though, that it is not always safe to modify strings in-place like this) Since you're replacing a character, you want a character constant, 'H'. It would not be right to write

```
string[0] = "H";        /* WRONG */
```

because "H" is a string (an array of characters), not a single character. (The destination of the assignment, string[0], is a char, but the right-hand side is a string; these types don't match.)

On the other hand, when you need a string, you must use a string. To print a single newline, you could call

```
printf("\n");
```

It would not be correct to call

```
printf('\n');          /* WRONG */
```

printf always wants a string as its first argument. (As one final example, putchar wants a single character, so putchar('\n') would be correct, and putchar("\n") would be incorrect.)

We must also remember the difference between strings and integers. If we treat the character '1' as an integer, perhaps by saying

```
int i = '1';
```

we will probably not get the value 1 in i; we'll get the value of the character '1' in the machine's character set. (In ASCII, it's 49.) When we do need to find the numeric value of a digit character (or to go the other way, to get the digit character with a particular value) we can make use of the fact that, in any character set used by C, the values for the digit characters, whatever they are, are contiguous. In other words, no matter what values '0' and

'1' have, '1' - '0' will be 1 (and, obviously, '0' - '0' will be 0). So, for a variable c holding some digit character, the expression

```
c - '0'
```

gives us its value. (Similarly, for an integer value i, i + '0' gives us the corresponding digit character, as long as 0 <= i <= 9.)

Just as the character '1' is not the integer 1, the string "123" is not the integer 123. When we have a string of digits, we can convert it to the corresponding integer by calling the standard function **atoi:**

```
char string[] = "123";
int i = atoi(string);
int j = atoi("456");
```

SELF-ASSESSMENT QUESTIONS – 3

6. Will there be a compilation error for the following program? (Yes/No). char str1[10];

str1=”Hello, world”; printf(“%s”, str1);

7. The library function used to copy one string to another is _____.
8. The library function atoi can be used for any string. (True/False)

5. SUMMARY

An array is a variable that can hold more than one value. In C, arrays are zero-based. An array name can be used as an argument to a function, thus permitting the entire array to be passed to the function. The C language allows arrays of any dimension to be defined. One of the most natural applications for a two-dimensional array arises in the case of a matrix. Strings in C are represented by arrays of characters. C has built in library functions to perform some operations on strings.

6. TERMINAL QUESTIONS

1. Write a program for 10 times summation of square of a number.
2. How many elements can the array in the following declaration accommodate?
   int a[3][4][5];
3. Is the following array declaration and initialization correct? int a[2][2]={1,2,3,4};
4. When you pass an array as a parameter to a function, the entire array is copied and is available to function. Justify whether this statement is true or false.
5. Write a Program that uses loops for array processing.

7. ANSWERS

Self-Assessment Questions

1. 0
2. True
3. Yes
4. rows
5. true
6. Compilation error
7. strcpy
8. false

Terminal Questions

1. #include<stdio.h>
   main()
   {
   int i=0, sum=0, x;
   printf('Enter a number:");
   scanf("%d", &x);
   while(i<10)
   {
   sum+=x*x;
   i++;
   }
   printf("Sum=%d", sum);
   }

2. 60
3. Yes, It is correct.
4. Statement is False.
5. // loops for array processing #include <stdio.h> #define SIZE 10 #define PAR 72
   int main(void)
   {

```
int index, score[SIZE]; int sum = 0;
float average;
printf("Enter %d golf scores:\n", SIZE); for (index = 0; index < SIZE; index++) scanf("%d", &score[index]); */read in the ten scores printf("The scores read in are as follows:\n");
for (index = 0; index < SIZE; index++) printf("%5d", score[index]); */verify input
printf("\n");
for (index = 0; index < SIZE; index++)
sum += score[index];   */add them up
average = (float) sum / SIZE; */ time-honored method printf("Sum of scores = %d, average = %.2f\n", sum, average); printf("That's a handicap of %.0f.\n", average - PAR);
return 0;
}
```

9. EXERCISES

1. Write a program to count the number of vowels and consonants in a given string.
2. Write a program to arrange a list of numbers in ascending order.
3. Write a program to multiply two matrices.
4. Write a program to rewrite a given string in the alphabetical order.
5. Write a program to transpose a given matrix.