Chapter 7: Strings and Arrays

WHAT IS A STRING ?

A string is a group of characters, usually letters of the alphabet. In order to format your printout in such a way that it looks nice, has meaningful names and titles, and is aesthetically pleasing to you and the people using the output of your program, you need the ability to output text data. Actually you have already been using strings, because the second program in this tutorial, way back in Chapter 2, output a message that was handled internally as a string. A complete definition of a string is a series of **char** type data terminated by a NULL character.

When C is going to use a string of data in some way, either to compare it with another string, output it, copy it to another string, or whatever, the functions are set up to do what they are called to do until a NULL, which is a zero, is detected. Such a string is often called an ASCII-Z string. We will use a few ASCII-Z strings in this chapter.

WHAT IS AN ARRAY ?

An array is a series of homogeneous pieces of data that are all identical in type, but the type can be quite complex as we will see when we get to the chapter of this tutorial discussing structures. A string is simply a special case of an array, a series of **char** type data.

The best way to see these principles is by use of an example, so load the program chrstrg.c and display it on your monitor. The first thing new is in line 6 which defines a **char** type of data entity. The square brackets define an array subscript in C, and in the case of the data definition statement, the 5 in the brackets defines 5 data fields of type **char** all defined as part of the variable. In the C language, all subscripts start at 0 and increase by 1 each step up to the maximum which in this case is 4. We therefore have 5 char type variables named, **name[0], name[1], name[2], name[3],** and **name[4]**. You must keep in mind that in C, the subscripts actually go from 0 to one less than the number defined in the definition statement. This is due to the original definition of C and these limits cannot be changed or redefined by the programmer.

HOW DO WE USE THE STRING ?

The variable **name** is therefore a string which can hold up to 5 characters, but since we need room for the NULL terminating character which counts as one of the five characters, there are actually only four usable characters. To load something useful into the string, we have 5 assignment statements, each of which assigns one alphabetical character to one of the string characters. Finally, the last place in the string is filled with the numeral 0 as the end indicator and the string is complete. (A define would allow us to use NULL instead of a zero, and this would add greatly to the clarity of the program. It would be very obvious that this was a NULL and not simply a zero for some other purpose.) Now that we have the string, we will print it out with some other string data in the output statement, line 14.

The **%s** in the format portion of the **printf()** statement is the output definition used to output a string. The system will output characters starting with the first one in the string **name** until it comes to the NULL character, where it will quit. Notice that in the **printf()** statement, only the variable name which happens to be **name** needs to be given, with no subscript since we are interested in starting at

the beginning. (There is actually another reason that only the variable name is given without brackets. The discussion of that topic will be given in the next chapter.) It is important to realize that **name** by itself refers to the entire string, but **name[]** with some value in the square braces refers to only a single character in the string.

OUTPUTTING PART OF A STRING

The **printf()** in line 15 illustrates that we can output any single character of the string by using the **%c** and naming the particular character of the variable **name** we want by including the subscript. Notice that the term with the square brackets refers to only a single character in the string, so we output it with the **%c** notation which is used to format and output a single character. The last **printf()** illustrates how we can output part of the string by stating the starting point by using a subscript. The & specifies the address of **name[1]**. We will study this in the next chapter but I thought you would benefit from a little glimpse ahead, so don't worry about this construct yet. You will notice, however, that we can print only part of the string by starting later in the string and printing until we reach the terminating null.

This example may make you feel that strings are rather cumbersome to use since you have to set up each character one at a time. Strings would be very difficult to use if they had to be defined like we defined the string in this program, but we only did this so you could see the internal structure of the string. The next example program will illustrate that strings are very easy to use. Be sure to compile and run this program.

SOME STRING FUNCTIONS

Load the example program strings.c for an example of some ways to use strings. First we define four strings in lines 7 and 8. Next we come to a new function that you will find very useful, the **strcpy()** function, or string copy. It copies from one string to another until it comes to the NULL character in the source string. Remember that the NULL is actually a zero and is added to the character string by the system. It is easy to remember which one gets copied to which if you think of them like an assignment statement. Thus if you were to say, for example, `x = 23;`, the data is copied from the right entity to the left one. In the **strcpy()** function, the data is also copied from the right entity to the left, so that after execution of the first statement, the string variable **name1** will contain the string "Rosalinda", but without the double quotes, they are the compiler's way of knowing that you are defining a string. The term "Rosalinda" is actually a string constant in exactly the same way that 23 is an integer constant as used in the expression index = 23. Line 10 is copying a string constant into a string variable.

Likewise, the string "Zeke" is copied into **name2** in line 11, then the title string is copied into the string named **title**. The title and both names are then printed out. Note that it is not necessary for the destination string to be exactly the same size as the string it will be called upon to store, only that it is at least as long as the source string plus one more character for the terminating NULL.

ALPHABETICAL SORTING OF STRINGS

The next function we will look at is the **strcmp()** or the string compare function illustrated in line 18. It will return a 1 if the first string is larger than the second, zero if they are the same length and have the same characters, and -1 if the first string is smaller than the second. One of the strings, depending on the result of the compare is copied into the string variable **mixed**, and the largest name alphabetically is printed out. It should come as no surprise to you that Zeke wins because it is alphabetically larger, length doesn't matter, only relative position in the alphabet. It might be wise to mention that the result would also depend on whether the letters were upper or lower case. There are also functions available with your C compiler to change the case of a string to all upper or all lower case if you desire. These will be used in an example program later in this tutorial.

COMBINING STRINGS

Lines 25 through 28 illustrate another new feature, the **strcat(),** or string concatenation function. This function simply adds the characters from one string onto the end of another string taking care to adjust the NULL so everything is still all right. In this case, **name1** is copied into **mixed**, then two blanks are concatenated to **mixed**, and finally **name2** is concatenated to the combination. The result is printed out with both names in the one string variable **mixed**.

Strings are not difficult to use and are extremely useful, but they do require some care in their use. It is possible to copy a long string into a string that has been defined as shorter, but this is an error and will overwrite some portion of your program. There is no way for the compiler to warn you of this.

A quick check of the documentation for one compiler revealed about 24 string functions available for use. Some are used for copying strings with upper limits on how many characters can be copied. There are string functions to search for certain characters in a string, and others for adding characters to the front, middle, or end of a string. And of course you can remove characters from anywhere also. It would pay you to read your compiler documentation to see just what string functions are available for your use. It could greatly simplify something you will be doing in the near future if you know what is available. You should spend some time getting familiar with strings before proceeding on to the next topic. We included the file named string.h in line 3 because it contains prototypes for all of the string functions. A little time spent examining this file would be time well spent.

Compile and run this program and observe the results for compliance with this definition.

AN ARRAY OF INTEGERS

Load the file intarray.c and display it on your monitor for an example of an array of integers. Notice that the array is defined in much the same way we defined an array of **char** in order to do the string manipulations in the last example program. We have 12 integer variables to work with, plus one more named **index**. The names of the variables are **values[0], values[1],** ... , and **values[11]**. In lines 9 and 10 we have a loop to assign nonsense, but well defined, data to each of the 12 variables, then print all 12 out in lines 12 and 13. Note carefully that each element of the array is simply an **int** type variable capable of storing an integer value. The only difference between the variables **index** and **values[2],** for example, is in the way you address them. You should have no trouble following this program, but be sure you understand it. Compile and run it to see if it does what you expect it to do.

AN ARRAY OF FLOATING POINT DATA

Load and display the program named bigarray.c for an example of a program with an array of **float** type data. This program has an extra feature to illustrate how strings can be initialized. Line 4 of the program illustrates how to initialize a string of characters. Notice that the square brackets are empty leaving it up to the compiler to count the characters and allocate enough space for our string plus the terminating NULL. Another string is initialized in line 11 of the body of the program but it must be declared **static** here. This prevents it from being allocated as an automatic variable and allows it to retain the string once the program is started. There is nothing else new here, the variables are assigned nonsense data and the results of all the nonsense are printed out along with a header. This program should also be easy for you to follow, so study it until you are sure of what it is doing before going on to the next topic. Once again, the **float** array can corrupt a program if it is used to write past the end of the array.

GETTING DATA BACK FROM A FUNCTION

Back in chapter 5 when we studied functions, I hinted to you that there was a way to get data back from a function by using an array, and that is true. Examine the program passback.c for an example of doing that. In this program, we define an array of 20 variables named **matrix** in line 8, then assign some nonsense data to the variables, and print out the first five. In line 16 we call the function **dosome()** taking along the entire array by putting the name of the array in the parentheses.

The function **dosome()** beginning in line 22 has a name in its parentheses also but it prefers to call the array **list** internally. The function needs to be told that it is really getting an array passed to it and that the array is of type **int**. Line 22 does that by defining **list** as an integer type variable and including the square brackets to indicate an array. It is not necessary to tell the function how many elements are in the array. Generally a function works with an array until some end-of-data marker is found, such as a NULL for a string, or some other previously defined data or pattern. Many times, another piece of data is passed to the function with a count of how many elements to work with. In our present illustration, we will use a fixed number of elements to keep it simple.

So far nothing is different from the previous functions we have called except that we have passed more data points to the function this time than we ever have before, having passed 20 integer values, the entire array. We print out the first 5 again in lines 26 and 27 to see if they did indeed get passed here. In lines 29 and 30 we add ten to each of the elements and print out the new values. Finally we return to the main program and print out the same 5 data points. We find that we have indeed modified the data stored in the calling program from within the function, and when we returned to the main program, we brought the changes back. Compile and run this program to verify this conclusion.

ARRAYS PASS DATA BOTH WAYS

We stated during our study of functions that when we passed data to a function, the system made a copy to use in the function which was thrown away when we returned. This is not the case with arrays. The actual array is passed to the function and the function can modify it any way it wishes to. The result of the modifications will be available back in the calling program. This may seem strange to you that arrays are handled differently from single point data, but they are. It really does make sense, but you will have to wait until we get to pointers to understand it.

A HINT AT A FUTURE LESSON

Another way of getting data back from a function to the calling program is by using a pointer which we will discuss in the next chapter. When we get there we will find that an array is in reality a pointer to a list of values. Don't let that worry you now, it will make sense when we get there. In the meantime, concentrate on arrays and understand the basics of them because when we get to the study of structures we will be able to define some pretty elaborate arrays.

MULTI-DIMENSIONAL ARRAYS

Load and display the file named multiary.c for an example of a program with doubly dimensioned arrays. The variable **big** is an 8 by 8 array that contains 8 times 8 or 64 elements total. The first element is **big[0][0]**, and the last is **big[7][7]**. Another array named **large** is also defined which is not square to illustrate that the array need not be square. Both are filled with data, one representing a multiplication table, and the other being formed into an addition table.

To illustrate that individual elements can be modified at will, one of the elements of **big** is assigned the value from one of the elements of **large** after being multiplied by 22 in line 17. Next **big[2][2]** is assigned the arbitrary value of 5, and this value is used for the subscripts of the assignment statement in line 19. The assignment statement in line 19 is in reality **big[5][5]** = 177; because each of the

subscripts contain the value 5. This is only done to illustrate that any valid expression can be used for a subscript. It must only meet two conditions, it must be an integer (although a **char** will work just as well), and it must be within the range of the subscript it is being used for.

The entire matrix variable **big** is printed out in a square form in lines 21 through 25 so you can check the values to see if they did get set the way you expected them to.

Programming Exercise:

---

1. Write a program with three short strings, about 6 characters each, and use **strcpy ()** to copy the string literals "one", "two", and "three" into them. Concatenate the three strings into one larger string defined with 30 characters and print the result out 10 times.
2. Define two integer arrays, each 10 elements long, called **array1** and **array2**. Using a loop, put some kind of nonsense data in each and add them term for term into another 10 element array named **arrays**. Finally, print all results in a table with an index number.
3. Define a string of some selected length, assign a word or phrase to it, and print it out as a string, then print it out as individual characters. Finally, print it out backwards by using a for loop with a decrementing third term. A useful function for the first term of the for loop is **strlen()** which returns the length of the string by counting characters up to, but not including, the terminating null.
