Sams Teach Yourself C in 24 Hours

Hour 13 - Manipulating Strings

I have made this letter longer than usual, because I lack the time to make it short.

—B. Pascal

In the last hour's lesson you learned how to use arrays to collect variables of the same type. You also learned that a character string is actually a character array ended with a null character \0. In this lesson you'll learn more about strings and C functions that can be used to manipulate strings. The following topics are covered:

- Declaring a string
- The length of a string
- Copying strings
- Reading strings with scanf(
- The gets() and puts() functions

Declaring Strings

This section teaches you how to declare and initialize strings, as well as the difference between string constants and character constants. First, let's review the definition of a string.

What Is a String?

As introduced in Hour 12, "Storing Similar Data Items," a *string* is a character array terminated by a null character (\0).

For instance, a character array, array_ch, declared in the following statement, is considered a character string:

```
char array_ch[7] = {`H', `e', `l', `l', `o', `!', `\0'};
```

In C, the null character can be used to mark the end of a string, or to return logical FALSE. C treats \0 as one character. Each character in a string takes only 1 byte.

A series of characters enclosed in double quotes ("") is called a *string constant*. The C compiler can automatically add a null character (\0) at the end of a string constant to indicate the end of the string.

For example, the character string "A character string." is considered a string constant; so is "Hello!".

Initializing Strings

As taught in the last lesson, a character array can be declared and initialized like this:

```
char arr_str[6] = {`H', `e', `l', `l', `o', `!'};
```

Here the array arr_str is treated as a character array. However, if you add a null character (\0) into the array, you can have the following statement:

```
char arr_str[7] = {`H', `e', `l', `l', `o', `!', `\0'};
```

Here the array arr_str is expanded to hold seven elements; the last element contains a null character. Now, the character array arr_str is considered a character string because of the null character that is appended to the array.

You can also initialize a character array with a string constant. For example, the following statement initializes a character array, str, with a string constant, "Hello!":

```
char str[7] = "Hello!";
```

The compiler can automatically append a null character (\0) to the end of the array, and treat the character array as a character string. Note that the size of the array is specified to hold up to seven elements, although the string constant has only six characters enclosed in double quotes. The extra space is reserved for the null character that the compiler will add later.

You can declare an unsized character array if you want the compiler to calculate the total number of elements in the array. For instance, the following statement

```
char str[] = "I like C.";
```

initializes an unsized character array, str, with a string constant. Later, when the compiler sees the statement, it will figure out the total memory space needed to hold the string constant plus an extra null character added by the compiler itself.

If you like, you can also declare a char pointer and then initialize the pointer with a string constant. The following statement is an example:

```
char *ptr_str = "I teach myself C.";
```

WARNING

Don't specify the size of a character array as too small. Otherwise, it cannot hold a string constant plus an extra null character. For instance, the following declaration is considered illegal:

```
char str[4] = "text";
```

Note that many C compilers will not issue a warning or an error message on this incorrect declaration. The runtime errors that could eventually arise as a result could be very difficult to debug. Therefore, it's your responsibility to make sure you specify enough space for a string.

The following statement is a correct one, because it specifies the size of the character array str that is big enough to hold the string constant plus an extra null character:

```
char str[5] = "text";
```

String Constants Versus Character Constants

As you know, a string constant is a series of characters enclosed in double quotes (" "). On the other hand, a character constant is a character enclosed in single quotes (` `).

When a character variable ch and a character array str are initialized with the same character,
x, such as the following,

```
char ch = `x';
char str[] = "x";
```

1 byte is reserved for the character variable ch, and two bytes are allocated for the character array str. The reason that an extra byte is needed for str is that the compiler has to append a null character to the array.

Another important thing is that a string is interpreted as a char pointer. Therefore, you can assign a character string to a pointer variable directly, like this:

```
char *ptr_str;
ptr_str = "A character string.";
```

However, you can not assign a character constant to the pointer variable, as shown in the following:

```
ptr_str = `x';  /* It's wrong. */
```

In other words, the character constant `x' contains a right value, and the pointer variable ptr_str expects a left value. But C requires the same kind of values on both sides of an assignment operator =.

It's legal to assign a character constant to a dereferenced char pointer like this:

```
char *ptr_str;
*ptr_str = `x';
```

Now the values on both sides of the = operator are of the same type.

The program in Listing 13.1 demonstrates how to initialize, or assign character arrays with string constants.

TYPE

Listing 13.1. Initializing strings.

```
1:  /* 13L01.c: Initializing strings */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char str1[] = {`A', ` `,
7:                    `s', `t', `r', `i', `n', `g', ` `,
8:                    `c', `o', `n', `s', `t', `a', `n', `t', `\0'};
9:     char str2[] = "Another string constant";
10:    char *ptr_str;
11:    int i;
12:
13:    /* print out str2 */
14:    for (i=0; str1[i]; i++)
15:       printf("%c", str1[i]);
16:    printf("\n");
17:    /* print out str2 */
18:    for (i=0; str2[i]; i++)
19:       printf("%c", str2[i]);

20:    printf("\n");
21:    /* assign a string to a pointer */
22:    ptr_str = "Assign a string to a pointer.";
23:    for (i=0; *ptr_str; i++)
24:       printf("%c", *ptr_str++);
25:    return 0;
26: }
```

OUTPUT

The following output is displayed on the screen after the executable 13L01.exe of the program in Listing 13.1 is created and run from a DOS prompt:

ANALYSIS

```
C:\app>13L01
A string constant
```

```
Another string constant
Assign a string to a pointer.
C:\app>
```

As you can see from Listing 13.1, there are two character arrays, str1 and str2, that are declared and initialized in lines 6_9. In the declaration of str1, a set of character constants, including a null character, is used to initialize the array. For str2, a string constant is assigned to the array in line 9. The compiler will append a null character to str2 later. Note that both str1 and str2 are declared as unsized arrays for which the compiler can automatically figure out how much memory is needed. The statement in line 10 declares a char pointer variable, ptr_str.

The for loop in lines 14 and 15 then prints out all the elements in str1. Because the last element contains a null character (\0) that is evaluated as FALSE, the str1[i] expression is used in the second field of the for statement. The str1[i] expression returns logical TRUE for each element in str1 except the last one holding the null character. After the execution of the for loop, the string A string constant is shown on the screen.

Likewise, another for loop in lines 18 and 19 displays the string constant assigned to str2 by putting every element of the array on the screen. Because the compiler appends a null character to the array, the str2[i] expression is evaluated in the for statement. The for loop stops iterating when str2[i] returns a logical FALSE. By that time, the content of the string constant, Another string constant, has already been displayed on the screen.

The statement in line 22 assigns a string constant, "Assign a string to a pointer.", to the char pointer variable ptr_str. Also, a for loop is used to print out the string constant by putting every item in the string on the screen (see lines 23 and 24). Note that the dereferenced pointer *ptr_str is used to refer to one of the characters in the string constant. When the null character appended to the string is encountered, *ptr_str returns logical FALSE, which causes the iteration of the for loop to stop. In line 24, the *ptr_str++ expression indicates that the dereferenced pointer moves to the next character of the string after the current character referred to by the pointer is fetched. In Hour 16, "Applying Pointers," you'll learn more about pointer arithmetic.

How Long Is a String?

Sometimes, you need to know how many bytes are taken by a string. In C, you can use a function called strlen() to measure the length of a string.

The strlen() Function

Let's have a look at the syntax of the strlen() function.

The syntax for the strlen() function is

```
#include <string.h>
size_t strlen(const char *s);
```

Here s is a char pointer variable. The return value from the function is the number of bytes. size_t is a data type defined in the string.h header file. The size of the data type depends on the particular computer system. Note that string.h has to be included in your program before you can call the strlen() function.

Listing 13.2 gives an example of using the strlen() function to measure string lengths.

TYPE

Listing 13.2. Measuring string lengths.

```
1:  /* 13L02.c: Measuring string length */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  main()
6:  {
7:     char str1[] = {`A', ` `,
8:                    `s', `t', `r', `i', `n', `g', ` `,
9:                    `c', `o', `n', `s', `t', `a', `n', `t', `\0'};
10:    char str2[] = "Another string constant";
11:    char *ptr_str = "Assign a string to a pointer.";
12:
13:    printf("The length of str1 is: %d bytes\n", strlen(str1));
14:    printf("The length of str2 is: %d bytes\n", strlen(str2));
15:    printf("The length of the string assigned to ptr_str is: %d bytes\n",
16:       strlen(ptr_str));
17:    return 0;
18: }
```

OUTPUT

I get the following output by running the executable 13L02.exe of the program in Listing 13.2 from a DOS prompt:

ANALYSIS

```
C:\app>13L02
The length of str1 is: 17 bytes
The length of str2 is: 23 bytes
The length of the string assigned to ptr_str is: 29 bytes
C:\app>
```

In Listing 13.2, two char arrays, str1 and str2, and one pointer variable, ptr_str, are declared and initialized in lines 7_11, respectively.

Then, the statement in line 13 obtains the length of the string constant held by str1, and prints out the result on the screen. From the result, you can see that the null character (\0) contained by the last element of str1 is not counted by the strlen() function.

In lines 14_16, the lengths of the string constants referenced by str2 and ptr_str are measured and shown on the screen. The results indicate that the strlen() function does not count the null characters appended to the two string constants by the compiler, either.

Copying Strings with strcpy()

If you want to copy a string from one array to another, you can copy each item of the first array to the corresponding element in the second array, or you can simply call the C function strcpy() to do the job for you.

The syntax for the strcpy() function is

```
#include <string.h>
char *strcpy(char *dest, const char *src);
```

Here the content of the string src is copied to the array referenced by dest. The strcpy() function returns the value of src if it is successful. The header file string.h must be included in your program before the strcpy() function is called.

The program in Listing 13.3 demonstrates how to copy a string from one array to another by either calling the strcpy() function or by doing it yourself.

TYPE

Listing 13.3. Copying strings.

```
1:  /* 13L03.c: Copying strings */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  main()
6:  {
7:     char str1[] = "Copy a string.";
8:     char str2[15];
9:     char str3[15];
10:    int  i;
11:
12:    /* with strcpy() */
13:    strcpy(str2, str1);
14:    /* without strcpy() */
15:    for (i=0; str1[i]; i++)
16:       str3[i] = str1[i];
17:    str3[i] = `\0';
18:    /* display str2 and str3 */
19:    printf("The content of str2: %s\n", str2);
20:    printf("The content of str3: %s\n", str3);
21:    return 0;
22: }
```

OUTPUT

After the executable 13L03.exe is created and run, the following output is shown on the screen:

ANALYSIS

```
C:\app>13L03
The content of str2: Copy a string.
The content of str3: Copy a string.
C:\app>
```

Three char arrays, str1, str2, and str3, are declared in Listing 13.3. In addition, str1 is initialized with a string constant, "Copy a string.", in line 7.

The statement in line 13 calls the strcpy() function to copy the content of str1 (including the null character appended by the compiler) to the array referenced by str2.

Lines 15_17 demonstrate another way to copy the content of str1 to an array referenced by str3. To do so, the for loop in lines 15 and 16 keeps copying characters of str1 to the corresponding elements in str3 one after another, until the null character (\0) appended by the compiler is encountered. When the null character is encountered, the str1[i] expression used as the condition of the for statement in line 15 returns logical FALSE, which in turn causes the loop to stop.

Because the for loop does not copy the null character from str1 to str3, the statement in line 17 appends a null character to the array referenced by str3. In C, it's very important to make sure that an array that is used to store a string has a null character as its last element.

To prove that the string constant referenced by str1 has been copied to str2 and str3 successfully, the contents held by str2 and str3 are displayed on the screen. Note that the string format specifier %s and the start addresses of str2 and str3 are invoked in the printf() function in lines 19 and 20 to print out all characters, except the null character, stored in str2 or str3. The results displayed on the screen show that str2 and str3 have the exact same content as str1.

Reading and Writing Strings

Now let's focus on how to read or write strings with the standard input and output streams—that is, stdin and stdout. In C, there are several functions you can use to deal with string reading or writing. The following subsections introduce some of the functions.

The gets() and puts() Functions

The gets() function can be used to read characters from the standard input stream.

The syntax for the gets() function is

```
#include <stdio.h>
char *gets(char *s);
```

Here the characters read from the standard input stream are stored in the character array identified by s. The gets() function stops reading, and appends a null character \0 to the array, when a newline or end-of-file (EOF) is encountered. The function returns s if it concludes successfully. Otherwise, a null pointer is returned.

The puts() function can be used to write characters to the standard output stream (that is, stdout).

The syntax for the puts() function is

```
#include <stdio.h>
int puts(const char *s);
```

Here s refers to the character array that contains a string. The puts() function writes the
string to the stdout. If the function is successful, it returns 0. Otherwise, a nonzero value is returned.

The puts() function appends a newline character to replace the null character at the end of a character array.

Both the gets() and puts() functions require the header file stdio.h. In Listing 13.4, you can see the application of the two functions.

TYPE

Listing 13.4. Using the gets() and puts() functions.

```
1:  /* 13L04.c: Using gets() and puts() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char str[80];
7:     int i, delt = `a' - `A';
8:
9:     printf("Enter a string less than 80 characters:\n");
10:    gets( str );
11:    i = 0;
12:    while (str[i]){
13:      if ((str[i] >= `a') && (str[i] <= `z'))
14:         str[i] -= delt;  /* convert to upper case */
15:      ++i;
16:    }
17:    printf("The entered string is (in uppercase):\n");
18:    puts( str );
19:    return 0;
20: }
```

ANALYSIS

After running the executable 13L04.exe, I enter a line of characters from the keyboard and have the characters (all in uppercase) shown on the screen:

OUTPUT

```
C:\app>13L04
Enter a string less than 80 characters:
This is a test.
The entered string is (in uppercase):
THIS IS A TEST.
C:\app>
```

The program in Listing 13.4 accepts a string of characters entered from the keyboard (that is, stdin), and then converts all lowercase characters to uppercase ones. Finally, the modified string is put back to the screen.

In line 6, a character array (str) is declared that can hold up to 80 characters. The gets() function in line 10 reads any characters the user enters from the keyboard until the user presses the Enter key, which is interpreted as a newline character. The characters read in by the gets() function are stored into the character array indicated by str. The newline character is not saved into str. Instead, a null character is appended to the array as a terminator.

The while loop in lines 12_15 has a conditional expression, str[i]. The while loop keeps iterating as long as str[i] returns logical TRUE. Within the loop, the value of each character represented by str[i] is evaluated in line 13, to find out whether the character is a lowercase character within the range of a through z. If the character is one of the lowercase characters, it is converted into uppercase by subtracting the value of an int variable, delt, from its current value in line 14. The delt variable is initialized in line 7 by the value of the `a' - `A' expression, which is the difference between a lowercase character and its uppercase counterpart. In other words, by subtracting the difference of `a' and `A' from the lower case integer value, we obtain the uppercase integer value.

Then the puts() function in line 18 outputs the string with all uppercase characters to stdout, which leads to the screen by default. A newline character is appended by the puts() function to replace the null character at the end of the string.

Using %s with the printf() Function

We've used the printf() function in many program examples in this book. As you know, many format specifiers can be used with the printf() function to specify different display formats for numbers of various types.

For instance, you can use the string format specifier, %s, with the printf() function to display a character string saved in an array on the screen. (See the example in Listing 13.3.)

In the next section, the scanf() function is introduced as a way to read values of various data types with different format specifiers, including

the format specifier %s.

The scanf() Function

The scanf() function provides another way to read strings from the standard input stream. Moreover, this function can actually be used to read various types of input data. The formats of arguments to the scanf() function are quite similar to those used in the printf() function.

The syntax for the scanf() function is

```
#include <stdio.h>
int scanf(const char *format, …);
```

Here various format specifiers can be included inside the format string referenced by the char pointer variable format. If the scanf() function concludes successfully, it returns the num-ber of data items read from the stdin. If an error occurs, the scanf() function returns EOF (end-of-file).

Note that using the string format specifier %s causes the scanf() function to read characters until a space, a newline, a tab, a vertical tab, or a form feed is encountered. Characters read by the scanf() function are stored into an array referenced by the corresponding argument. The array should be big enough to store the input characters.

A null character is automatically appended to the array after the reading.

The program in Listing 13.5 shows how to use various format specifiers with the scanf() function.

TYPE

Listing 13.5. Using the scanf() function with various format specifiers.

```
1:  /* 13L05.c: Using scanf() */
2:  #include <stdio.h>
3:
4:  main()
5:  {
6:     char str[80];
7:     int x, y;
8:     float z;
9:
10:    printf("Enter two integers separated by a space:\n");
11:    scanf("%d  %d", &x, &y);
12:    printf("Enter a floating-point number:\n");
13:    scanf("%f", &z);
14:    printf("Enter a string:\n");
15:    scanf("%s", str);
16:    printf("Here are what you've entered:\n");
17:    printf("%d  %d\n%f\n%s\n", x, y, z, str);
18:    return 0;
19: }
```

OUTPUT

The following output is a sample displayed on the screen after I run the executable 13L05.exe and enter data (which appears in bold) from my keyboard:

ANALYSIS

```
C:\app>13L05
Enter two integers separated by a space:
10  12345
Enter a floating-point number:
1.234567
Enter a string:
Test
Here are what you've entered:
10  12345
1.234567
Test
C:\app>
```

In Listing 13.5, there are one char array (str), two int variables (x and y), and a float variable (z) declared in lines 6_8.

Then, the scanf() function in line 11 reads in two integers entered by the user and saves them into the memory locations reserved for the integer variables x and y. The statement in line 13 reads and stores a floating-point number into z. Note that the format specifiers, %d and %f, are used to specify proper formats for entered numbers in lines 11 and 13.

Line 15 reads a series of characters entered by the user with the scanf() function by using the format specifier %s, and then saves the characters, plus a null character as the terminator, into the array pointed to by str.

To prove that the scanf() function reads all the numbers and characters entered by the user, the printf() function in line 17 displays the contents saved in x, y, z, and str on the screen. Sure enough, the result shows that the scanf() does a good job.

One thing you need to be aware of is that the scanf() function doesn't actually start reading the input until the Enter key is pressed. Data entered from the keyboard is placed in an input buffer. When the Enter key is pressed, the scanf() function looks for its input in the buffer. You'll learn more about buffered input and output in Hour 21, "Disk File Input and Output: Part I."

Summary

In this lesson you've learned the following:

- A string is a character array with a null character as the terminator at the last element.

- A string constant is a series of characters enclosed by double quotes.
- The C compiler automatically appends a null character to the array that has been initialized by a string constant.
- You cannot assign a string constant to a dereferenced char pointer.
- The strlen() function can be used to measure the length of a string. This function does not count the null character in the last element.
- You can copy a string from one array to another by calling the C function strcpy().
- The gets() function can be used to read a series of characters. This function stops reading when the newline character or end-of-file (EOF) is encountered. A null character is attached to the array that stores the characters automatically after the reading.
- The puts() function sends all characters, except the null character, in a string to the stdout, and appends a newline character to the output.
- You can read different data items with the scanf() function by using various format specifiers.

In the next lesson you'll learn about the concepts of scope and storage in C.

Q&A

Q What is a string? How do you know its length?

A In C, a string is a character array terminated by a null character. Whenever a null character is encountered in a string, functions, such as puts() or strcpy(), will stop printing or copying the next character.

The C function strlen() can be used to measure the length of a string. If it is successful, the strlen() function returns the total number of bytes taken by the string; however, the null character in the string is not counted.

Q What are the main differences between a string constant and a character constant?

A A string constant is a series of characters enclosed by double quotes, while a character constant is a single character surrounded by single quotes. The compiler will append a null character to the array that is initialized with a string constant. Therefore, an extra byte has to be reserved for the null character. On the other hand, a character constant takes only 1 byte in the memory.

Q Does the gets() function save the newline character from the standard input stream?

A No. The gets() function keeps reading characters from the standard input stream until a newline character or end-of-file is encountered. Instead of saving the newline character, the gets() function appends a null character to the array that is referenced by the argument to the gets() function.

Q What types of data can the scanf() function read?

A Depending on the format specifiers indicated in the function, the scanf() function can read various types of data, such as a series of characters, integers, or floating-point numbers. Unlike gets(), scanf() stops reading the current input item (and moves to the next input item, if there is one) when it encounters a space, a newline, a tab, a vertical tab, or a form feed.

Workshop

To help solidify your understanding of this hour's lesson, you are encouraged to answer the quiz questions and finish the exercises provided in the W orkshop before you move to the next lesson. The answers and hints to the questions and exercises are given in Appendix E, "Answers to Quiz Questions and Exercises."

Quiz

1. In the following list, which statements are legal?
    - char str1[5] = "Texas";
    - char str2[] = "A character string";
    - char str3[2] = "A";
    - char str4[2] = "TX";
2. Given a char pointer variable ptr_ch, are the following statements legal?
    - *ptr_ch = `a';
    - ptr_ch = "A character string";
    - ptr_ch = `x';
    - *ptr_ch = "This is Quiz 2.";
3. Can the puts() function print out the null character in a character array?
4. Which format specifier do you use with the scanf() function to read in a string, and which one do you use to read a floating-point number?

Exercises

1. Given a character array in the following statement,

    ```
    char str1[] = "This is Exercise 1.";
    ```

    write a program to copy the string from str1 to another array, called str2.

2. Write a program to measure the length of a string by evaluating the elements in the character array one by one. To prove you get the right result, you can use the strlen() function to measure the same string again.
3. Rewrite the program in Listing 13.4. This time, convert all uppercase characters to their lowercase counterparts.
4. Write a program that uses the scanf() function to read in two integers entered by the user, adds the two integers, and then prints out the sum on the screen.