Chapter 9
Strings

9.1 Chapter Overview
There is no data type in C called ‘string’; instead, strings are represented by an array of characters. There is an assortment of useful functions for strings that are available in the C library by including the string header file at the top of your program:

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

In this chapter, several of the more commonly performed operations on strings are looked at in detail, including concatenation, string copying, extracting a portion of a string, and determining if two strings are equal.

9.2 Declaring and Initializing a String
A word in C, for example, the following “hello!” might be declared as:
```
char word[] = { 'h', 'e', 'l', 'l', 'o', '!' };
```
which would be a character array containing those 6 characters (and only those 6 characters).

In C, a special character is used to signal the end of a string known as the null character, which is represented by ‘\0’. So, the following would be done to declare the word above including the null character:
```
char word[] = { 'h', 'e', 'l', 'l', 'o', '!', '\0' };
```
which would make it 7 characters long. By using this special character, this allows you to deal with strings without always having to know how long they are (variable length character strings) – as soon as the null character is reached, it is at the end of the string. (One of the functions included in string.h is strlen() – more about that later.)

Defining it that way in the above example is the same as the following:
```
char word[] = { "hello!" };
```
or
```
char word[] = "hello!";
```
each of which results in a 7 character word including the null character at the end.

If you are explicitly specifying the size of the array, don't forget the null character:
```
char word[7] = "hello!";
```
With the above declaration, the compiler will have enough room to place the ‘\0’ character at the end, whereas, the following would not:
```
char word[6] = "hello!";
```
If the compiler does not have enough room for the ‘\0’ character at the end, it just leaves it out, and it doesn’t complain about it either. A character array that does not have the null character ‘\0’ at the end, it no longer ‘acts’ like a string and you may get unpredictable results.

In C, character string constants are automatically terminated by the null character. With the following:
```
printf("I love programming in C! \n");
```
the null character is automatically placed at the end after the newline character, thereby enabling the printf function to determine when it has reached the end of the string.

With this array declaration:
```
char s[10];
```
the c-string variable s will be able to hold up to 9 characters plus one null character. None of the elements of the array have an initial default value:

```
|------|------|------|------|------|------|------|------|------|------|
```
With this array declaration and initialization:
```
char s[10] = “Hi Mom!”;
```
the c-string variable s will be stored as:
```
| H | i | M | o | m | ! | 0 | ? | ? |
```

When declaring a character array, we usually try to specify a size that is large enough to hold the maximum size string we wish to store plus one for the null character. An initial value that is too long to fit in the specified size will result in a syntax error.

### 9.3 Displaying Strings

Another useful feature in C involves the display of strings. You can use the special format character %s inside a printf to display an array of characters that is terminated by the null character. So the following string:
```
char word[7] = “hello!”;
```
can be displayed with the following statement:
```
printf(“%s
”, word);
```

The printf assumes that when it sees a %s, the argument is a character string that is terminated by the null character.

### 9.4 More on Constant Strings

If you are initializing a long character string, you can break it up onto two lines, like the following, using a backslash character at the end of each line that is to be continued:
```
char letters[] = { “abcdefghijklmnopqrstuvwxyz
    ABCDEFGHIJKLMNOPQRSTUVWXYZ” };
```
or by this method, using separate strings, which the compiler automatically concatenates:
```
char letters[] = { “abcdefghijklmnopqrstuvwxyz
    “ABCDEFGHIJKLMNOPQRSTUVWXYZ” };
```

Because the compiler automatically concatenates adjacent strings, the following three printf statements all pass a single argument to printf:
```
printf (“I love programming in C! \n”);
printf (“I love programming “ “in C! \n”);
printf (“I love “ “programming “ “in C! \n”);
```

### 9.5 The NULL String

A character string that contains no characters is called a null string. It has a length of zero. This can be used, for example, as a way for the user to tell the program when he is done entering text by having him hit the enter key an extra time after the last line of text has been entered, as with Program 9.8 in the book, pages 212 - 214.

### 9.6 Inputting Character Strings

The scanf function can be used with the %s format characters to read in a string of characters up to a blank space, tab character, or the end of the line, whichever occurs first. The following has the effect of reading a character string typed into your terminal window and storing it inside the character array string.
```
char string[81];
scanf (“%s”, string);
```

There are two things to note. First, there is no & in front of string. This is because it is an array and the array name already points to the location in memory for that item.
Second, notice the size of the array. You can declare an array like this – perhaps larger than what you think you’ll need, to be sure there is sufficient space for the string. When the string is read and put into that array, if it is shorter than 81 characters, the null character will be put at the end of the string. Then, if you use a `printf` statement to print it out, no matter how much shorter it is, the `printf` stops as soon as it reaches the null character, so it doesn’t matter that the array was declared to be larger than the actual string.

```c
scanf("%s%s%s", string1, string2, string3);
```

The above line will read in three strings and store them into the strings named `string1`, `string2`, and `string3`. So, the user can enter three strings separated by a space, tab, or each on a separate line. It will wait until three strings have been entered.

When handed the `%s` format, `scanf` continues to read until the user hits the space bar, tab key, or enter. If the user were to enter more than 80 characters without pressing the spacebar, tab key, or enter, `scanf` overflows one of the character arrays, which might cause the program to terminate abnormally or cause unpredictable things to happen. If you want to make sure this doesn’t happen, you can place a number after the `%` in the `scanf` format string, which tells `scanf` the maximum number of characters to read, like the following:

```c
scanf("%80s%80s%80s", string1, string2, string3);
```

### 9.7 `getchar` and `gets` Functions

The standard library provides several functions for reading and writing single characters and entire character strings.

`getchar` can be used to **read in a single character** from the terminal. Why do you need this when you can use `scanf` with the format characters `%c`, you ask? `getchar` is defined to read in a single character, so it doesn’t need any arguments. So you can have a variable of type `char` called `character`, and then assign that to `getchar()`, as with the following:

```c
char character;
...
character = getchar();
```

`getchar` actually returns an integer, but assigning it to a variable of type `char` automatically converts it from an `int` to a `char`.

`gets` is another function available from the library, which is used to **read in a single line of text**. The newline character signals the end of the line, and the null character is placed at the end of the array (replacing the newline character) to terminate the character string. If successful, it returns the string and places it into the argument. The `gets` function is dangerous because it does not check the length of the input and will overrun the string if too long. On many machines, `gcc` will produce a warning message when using `gets`.

```c
char input_string[100];
gets(input_string);
```

Program 9.6, page 207, shows how to implement a function similar to the `gets` function using `getchar`. 
9.8 Character Output, `putchar` and `puts`

Using the `%s` format string with `printf` allows you to print the value of a variable that is a c-string to the screen.

```c
char string[81];
scanf ("%s", string);
...
printf ("%s \n", string);
```

`putchar` **writes one character at a time** to standard output.

```c
char msg[] = "dlroW olleH";
int ndx;

// print dlroW olleH
printf ("%s \n", msg);

// print Hello World
for (ndx = (int)strlen(msg) - 1; ndx >= 0; ndx--)
    putchar(msg[ndx]);
printf("\n");
```

`puts` **writes a string** to standard output. A newline character is written in place of the null character. This means you will always get a new line character when using `puts`.

```c
char hello[] = "Hello";
puts (hello);
printf("-----\n");

/* output:
Hello
-----
*/
```

9.9 Escape Characters

Some of the escape characters from page 215:

- \a audible alert (sounds a bell)
- \b backspace
- \f form feed
- \n newline
- \r carriage return
- \t horizontal tab
- \v vertical tab
- \\ backslash
- \" double quotation mark
- \' single quotation mark
- \? question mark

Each escape character is counted as a single character. So the following string contains 9 characters:

```
\033\"Hello\"\n (033 is the ASCII escape character)
9.10 Character Operations
Character constants and variables in C are treated as an integer value. In ASCII, the character ‘a’ has the value of 97, 'b' a value of 98, etc. Therefore the expression c >= ‘a’ is TRUE (nonzero) for any lowercase character contained in c.

The following expression could be used to test if a letter is a lowercase letter:
if c >= 'a' && c <= 'z'
which would be the same as the following:
if c >= 97 && c <= 122

printf ("%i", 'a'); would print 97
printf ("%c", 'a'); would print a

c = 'a' + 1;
printf ("%c \n", c); would print b

This can be very useful to convert characters '0' through '9' to their corresponding numerical values 0 through 9. Because '0' has the numerical value of 48 in ASCII, you can subtract '0' from it to get the numerical value of 0. Same with any other character digit: '1' – '0' would result in the numerical value of 1 (49 – 48); '2' – '0' would result in the numerical value of 2 (50 – 48); etc. This makes it easy to convert a character string consisting of digits to its equivalent numerical representation. There is a function available in the library called atoi() that converts the string sent to it to an integer and returns that integer value. There are also other string conversion functions available as well; Appendix B shows some of the common ones, pages 483 - 484; they are in the <stdlib.h> header file. Program 9.11 on page 228 shows an implementation of a string conversion, called strToInt, demonstrating how a string can be converted to its corresponding integer value.

9.11 String Library
C provides a string library that contains functions for manipulating strings. If you #include <string.h> in your program, you can use functions like: strcpy, strcat, strlen, strcmp, strchr, strtok, memset, memcpy, among others. You can reference your book or Google "string.h" for more information on these and other functions in the string library.

strlen returns the length of the string (not including the null character). The function returns a size_t data type, so you may need to typecast the return type depending on where it appears.

```c
char helloWorld[] = "Hello World";
printf("%d", (int)strlen(helloWorld)); // the above will print number 11
```
**`strcpy` copies a string value into a character array variable.** Strings are not like other variables – you **cannot assign a value to an array:**

```c
char msg[10];
   .   .
msg = ”Hello”;   // CANNOT DO THIS!!
```

To assign a value to an array after it has been declared, you can do one of two things. You can assign each element of the array individually:

```c
char msg[10];
   .   .
msg[0] = ’H’;
msg[1] = ’e’;
msg[2] = ’l’;
msg[3] = ’l’;
msg[4] = ’o’;
msg[5] = ’\0’;
```

Or, you can use the **`strcpy` function:**

```c
char msg[10];
   .   .
strcpy(msg, ”Hello”);
```

There is also a **`strncpy` function: `strncpy(destination, source, limit)```** This has a third argument, which is a limit of the number of characters from the source to copy. If the limit is reached, no null character is placed in the destination, however.
`strcmp` compares two strings to check for equality. You **cannot** use `==` operator to compare two strings:

```c
char hello[] = "Hello";
char help[] = "Help";
if (hello == help)  // CANNOT DO THIS!!!
```

You must use `strcmp`: `strcmp(string1, string2)` The function compares the same two elements from each string (the first character from string1 compared to the first character from string2; the second character from string1 compared to the second character from string2; etc.) This function returns a 0 if there is no difference between the two strings (if they are the same). It returns a value <0 if the first non-matching character from string1 comes before (alphabetically) the one from string2. It returns a value >0 if the first non-matching character from string1 comes after the one from string2.

```c
char hello[] = "Hello";
char help[] = "Help";
if (strcmp(hello, help) == 0)
    printf("They are the same string\n");
else if (strcmp(hello, help) < 0)
    printf("hello comes before help\n");
else if (strcmp(hello, help) > 0)
    printf("hello comes after help\n");

// will print:  hello comes before help
```

`strcat` concatenates two strings together, i.e. appends one string onto the end of another. Be careful when concatenating words – it does not automatically add spaces in between the strings. `strcat(destination, source)`

```c
char msg1[30] = "Hello";
char msg2[30] = "Hello";

strcat(msg1, "World");  // result:  HelloWorld
strcat(msg2, " World"); // result:  Hello World
```