CHAPTER 6

Applications with Strings and Text

In the last chapter you were introduced to arrays and you saw how using arrays of numerical values could make many programming tasks much easier. In this chapter you'll extend your knowledge of arrays by exploring how you can use arrays of characters. You'll frequently have a need to work with a text string as a single entity. As you'll see, C doesn't provide you with a string data type as some other languages do. Instead, C uses an array of elements of type `char` to store a string.

In this chapter I'll show you how you can create and work with variables that store strings, and how the standard library functions can greatly simplify the processing of strings.

You'll learn the following:

- How you can create string variables
- How to join two or more strings together to form a single string
- How you compare strings
- How to use arrays of strings
- How you work with wide character strings
- What library functions are available to handle strings and how you can apply them
- How to write a simple password-protection program

What Is a String?

You've already seen examples of string constants—quite frequently in fact. A **string constant** is a sequence of characters or symbols between a pair of double-quote characters. Anything between a pair of double quotes is interpreted by the compiler as a string, including any special characters and embedded spaces. Every time you've displayed a message using `printf()`, you've defined the message as a string constant. Examples of strings used in this way appear in the following statements:

```
printf("This is a string.");
printf("This is on\ntwo lines!");
printf("For \" you write \\\".");
```

These three example strings are shown in Figure 6-1. The decimal value of the character codes that will be stored in memory are shown below the characters.

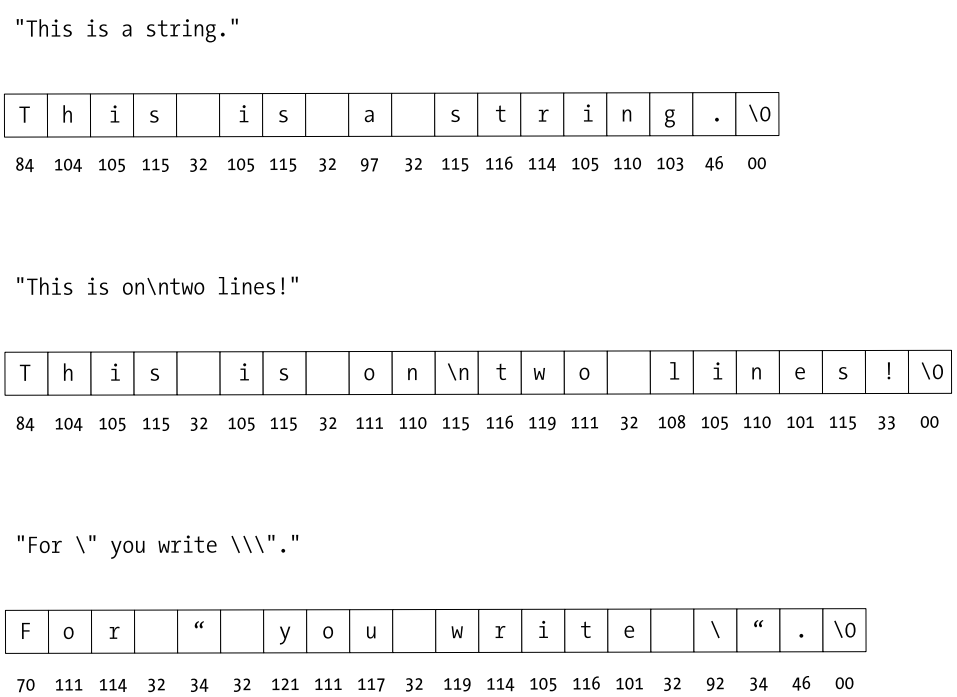

Figure 6-1. *Examples of strings in memory*

The first string is a straightforward sequence of letters followed by a period. The `printf()` function will output this string as the following:

```
This is a string.
```

The second string has a newline character, `\n`, embedded in it so the string will be displayed over two lines:

```
This is on
two lines!
```

The third string may seem a little confusing but the output from `printf()` should make is clearer:

```
For " you write \".
```

You must write a double quote within a string as the escape sequence `\"` because the compiler will interpret an explicit " as the end of the string. You must also use the escape sequence `\\` when you want to include a backslash in a string because a backslash in a string always signals to the compiler the start of an escape sequence.

As Figure 6-1 shows, a special character with the code value 0 is added to the end of each string to mark where it ends. This character is known as the **null character** (not to be confused with `NULL`, which you'll see later), and you write it as `\0`.

■ **Note** Because a string in C is always terminated by a `\0` character, the length of a string is always one greater than the number of characters in the string.

There's nothing to prevent you from adding a \0 character to the end of a string yourself, but if you do, you'll simply end up with two of them. You can see how the null character \0 works with a simple example. Have a look at the following program:

```
/* Program 6.1 Displaying a string */
#include <stdio.h>

int main(void)
{
  printf("The character \0 is used to terminate a string.");
  return 0;
}
```

If you compile and run this program, you'll get this output:

```
The character
```

It's probably not quite what you expected: only the first part of the string has been displayed. The output ends after the first two words because the printf() function stops outputting the string when it reaches the first null character, \0. Even though there's another \0 at the end of string, it will never be reached. The first \0 that's found always marks the end of the string.

String- and Text-Handling Methods

Unlike some other programming languages, C has no specific provision within its syntax for variables that store strings, and because there are no string variables, C has no special operators for processing strings. This is not a problem, though, because you're quite well-equipped to handle strings with the tools you have at your disposal already.

As I said at the beginning of this chapter, you use an array of type char to hold strings. This is the simplest form of string variable. You could declare a char array variable as follows:

```
char saying[20];
```

The variable saying that you've declared in this statement can accommodate a string that has up to 19 characters, because you must allow one element for the termination character. Of course, you can also use this array to store 20 characters that aren't a string.

Caution Remember that you must always declare the dimension of an array that you intend to use to store a string as at least one greater than the number of characters that you want to allow the string to have because the compiler will automatically add \0 to the end of a string constant.

You could also initialize the preceding string variable in the following declaration:

```
char saying[] = "This is a string.";
```

Here you haven't explicitly defined the array dimension. The compiler will assign a value to the dimension sufficient to hold the initializing string constant. In this case it will be 18, which corresponds to 17 elements for the characters in the string, plus an extra one for the terminating \0. You could, of course, have put a value for the dimension yourself, but if you leave it for the compiler to do, you can be sure it will be correct.

You could also initialize just part of an array of elements of type `char` with a string, for example:

```
char str[40] = "To be";
```

Here, the compiler will initialize the first five elements from `str[0]` to `str[4]` with the characters of the specified string in sequence, and `str[5]` will contain the null value `'\0'`. Of course, space is allocated for all 40 elements of the array, and they're all available to use in any way you want.

Initializing a `char` array and declaring it as constant is a good way of handling standard messages:

```
const char message[] = "The end of the world is nigh";
```

Because you've declared `message` as `const`, it's protected from being modified explicitly within the program. Any attempt to do so will result in an error message from the compiler. This technique for defining standard messages is particularly useful if they're used in various places within a program. It prevents accidental modification of such constants in other parts of your program. Of course, if you do need to be able to change the message, then you shouldn't specify the array as `const`.

When you want to refer to the string stored in an array, you just use the array name by itself. For instance, if you want to output the string stored in message using the `printf()` function, you could write this:

```
printf("\nThe message is: %s", message);
```

The `%s` specification is for outputting a null-terminating string. At the position where the `%s` appears in the first argument, the `printf()` function will output successive characters from the message array until it finds the `'\0'` character. Of course, an array with elements of type `char` behaves in exactly the same way as an array of elements of any other type, so you use it in exactly the same way. Only the special string handling functions are sensitive to the `'\0'` character, so outside of that there really is nothing special about an array that holds a string.

The main disadvantage of using `char` arrays to hold a variety of different strings is the potentially wasted memory. Because arrays are, by definition, of a fixed length, you have to declare each array that you intend to use to store strings with its dimension set to accommodate the maximum string length you're likely to want to process. In most circumstances, your typical string length will be somewhat less than the maximum, so you end up wasting memory. Because you normally use your arrays here to store strings of different lengths, getting the length of a string is important, especially if you want to add to it. Let's look at how you do this using an example.

TRY IT OUT: FINDING OUT THE LENGTH OF A STRING

In this example, you're going to initialize two strings and then find out how many characters there are in each, excluding the null character:

```
/* Program 6.2 Lengths of strings  */
#include <stdio.h>
int main(void)
{
  char str1[] = "To be or not to be";
  char str2[] = ",that is the question";
  int count = 0;                /* Stores the string length                */
  while (str1[count] != '\0')   /* Increment count till we reach the string */
    count++;                    /*  terminating character.                 */
  printf("\nThe length of the string \"%s\" is %d characters.", str1, count);
```

```
  count = 0;                    /* Reset to zero for next string           */
  while (str2[count] != '\0')   /* Count characters in second string       */
    count++;
  printf("\nThe length of the string \"%s\" is %d characters.\n", str2, count);
  return 0;
}
```

The output you will get from this program is the following:

```
The length of the string "To be or not to be" is 18 characters.
The length of the string ",that is the question" is 21 characters.
```

How It Works

First you have the inevitable declarations for the variables that you'll be using:

```
char str1[] = "To be or not to be";
char str2[] = ",that is the question";
int count = 0;                  /* Stores the string length                */
```

You declare two arrays of type `char` that are each initialized with a string. The compiler will set the size of each array to accommodate the string including its terminating null. You also declare and initialize a counter, `count`, to use in the loops in the program. Of course, you could have omitted the dimension for each array and left the compiler to figure out what is required, as you saw earlier.

Next, you have a `while` loop that determines the length of the first string:

```
while (str1[count] != '\0')     /* Increment count till we reach the string */
  count++;                      /*  terminating character.                  */
```

Using a loop in the way you do here is very common in programming with strings. To find the length, you simply keep incrementing a counter in the `while` loop as long as you haven't reached the end of string character. You can see how the condition for the continuation of the loop is whether the terminating `'\0'` has been reached. At the end of the loop, the variable `count` will contain the number of characters in the string, excluding the terminating null.

I have shown the `while` loop comparing the value of the `str1[count]` element with `'\0'` so the mechanism for finding the end of the string is clear to you. However, this loop would typically be written like this:

```
while(str1[count])
  count++;
```

The ASCII code value for the `'\0'` character is zero which corresponds to the Boolean value `false`. All other ASCII code values are nonzero and therefore correspond to the Boolean value `true`. Thus the loop will continue as long as `str1[count]` is not `'\0'`, which is precisely what you want.

Now that you've determined the length, you display the string with the following statement:

```
printf("\nThe length of the string \"%s\" is %d characters.", str1, count);
```

This also displays the count of the number of characters that the string contains, excluding the terminating null. Notice that you use the new format specifier, `%s` that we saw earlier. This outputs characters from the string until it reaches the terminating null. If there was no terminating character, it would continue to output characters until it found one somewhere in memory. In some cases, that can mean *a lot* of output. You also use the escape character, `\"`, to include a double quote in the string. If you don't precede the double-quote character with the backslash, the compiler will think it marked the end of the string that is the first argument to the `printf()` function, and the statement will cause an error message to be produced.

You find the length of the second string and display the result in exactly the same way as the first string.

Operations with Strings

The code in the previous example is designed to show you the mechanism for finding the length of a string, but you never have to write such code in practice. As you'll see very soon, the `strlen()` function in the standard library will determine the length of a null-terminated string for you. So now that you know how to find the lengths of strings, how can you manipulate them?

Unfortunately you can't use the assignment operator to copy a string in the way you do with `int` or `double` variables. To achieve the equivalent of an arithmetic assignment with strings, one string has to be copied element by element to the other. In fact, performing any operation on string variables is very different from the arithmetic operations with numeric variables you've seen so far. Let's look at some common operations that you might want to perform with strings and how you would achieve them.

Appending a String

Joining one string to the end of another is a common requirement. For instance, you might want to assemble a single message from two or more strings. You might define the error messages in a program as a few basic text strings to which you append one of a variety of strings to make the message specific to a particular error. Let's see how this works in the context of an example.

TRY IT OUT: JOINING STRINGS

You could rework the last example to append the second string to the first:

```
/* Program 6.3 Joining strings */
#include <stdio.h>

int main(void)
{
  char str1[40] = "To be or not to be";
  char str2[] = ",that is the question";
  int count1 = 0;                /* Length of str1 */
  int count2 = 0;                /* Length of str2 */

  /* find the length of the first string */
  while (str1[count1]        )   /* Increment count till we reach the string */
    count1++;                    /* terminating character.                   */

  /* Find the length of the second string */
  while (str2[count2])           /* Count characters in second string */
    count2++;

  /* Check that we have enough space for both strings  */
  if(sizeof str1 < count1 + count2 + 1)
    printf("\nYou can't put a quart into a pint pot.");
  else
  {  /* Copy 2nd string to end of the first  */
     count2 = 0;                 /* Reset index for str2 to 0   */
    while(str2[count2])          /* Copy up to null from str2   */
      str1[count1++] = str2[count2++];
```

```
    str1[count1] = '\0';      /* Make sure we add terminator  */
    printf("\n%s\n", str1 );  /* Output combined string    */
  }
  return 0;
}
```

The output from this program will be the following:

```
To be or not to be, that is the question
```

How It Works

This program first finds the lengths of the two strings. It then checks that `str1` has enough elements to hold both strings plus the terminating null character:

```
if(sizeof str1 < count1 + count2 + 1)
  printf("\nYou can't put a quart into a pint pot.");
```

Notice how you use the `sizeof` operator to get the total number of bytes in the array by just using the array name as an argument. The value that results from the expression `sizeof str1` is the number of characters that the array will hold, because each character occupies 1 byte.

If you discover that the array is too small to hold the contents of both strings, then you display a message. The program will then end as you fall through the closing brace in `main()`. It's essential that you do not try to place more characters in the array than it can hold, as this will overwrite some memory that may contain important data. This is likely to crash your program. You should never append characters to a string without first checking that there is sufficient space in the array to accommodate them.

You reach the `else` block only if you're sure that both strings will fit in the first array. Here, you reset the variable `count2` to 0 and copy the second string to the first array with the following statements:

```
  else
  {  /* Copy 2nd string to end of the first  */
     count2 = 0;                /* Reset index for str2 to 0   */
    while(str2[count2])         /* Copy up to null from str2   */
      str1[count1++] = str2[count2++];

    str1[count1] = '\0';        /* Make sure we add terminator */
    printf("\n%s\n", str1 );    /* Output combined string      */
  }
```

The variable `count1` starts from the value that was left by the loop that determined the length of the first string, `str1`. This is why you use two separate variables to count the number of characters in each of the two strings. Because the array is indexed from 0, the value that's stored in `count1` will point to the element containing `'\0'` at the end of the first string. So when you use `count1` to index the array `str1`, you know that you're starting at the end of the message proper and that you'll overwrite the null character with the first character of the second string.

You then copy characters from `str2` to `str1` until you find the `'\0'` in `str2`. You still have to add a terminating `'\0'` to `str1` because it isn't copied from `str2`. The end result of the operation is that you've added the contents of `str2` to the end of `str1`, overwriting the terminating null character for `str1` and adding a terminating null to the end of the combined string.

You could replace the three lines of code that did the copying with a more concise alternative:

```
 while ((str1[count1++] = str2[count2++]));
```

This would replace the loop you have in the program as well as the statement to put a `'\0'` at the end of `str1`. This statement would copy the `'\0'` from `str2` to `str1`, because the copying occurs in the loop continuation condition. Let's consider what happens at each stage.

1. Assign the value of `str2[count2]` to `str1[count1]`. An assignment expression has a value that is the value that was stored in the left operand of the assignment operator. In this case it is the character that was copied into `str1[count1]`.
2. Increment each of the counters by 1, using the postfix form of the ++ operator.
3. Check whether the value of the assignment expression—which will be the last character stored in `str1`—is `true` or `false`. The loop ends after the `'\0'` has been copied to `str1`, which will result in the value of the assignment being `false`.

Arrays of Strings

It may have occurred to you by now that you could use a two-dimensional array of elements of type `char` to store strings, where each row is used to hold a separate string. In this way you could arrange to store a whole bunch of strings and refer to any of them through a single variable name, as in this example:

```
char sayings[3][32] = {
                        "Manners maketh man.",
                        "Many hands make light work.",
                        "Too many cooks spoil the broth."
                      };
```

This creates an array of three rows of 32 characters. The strings between the braces will be assigned in sequence to the three rows of the array, `sayings[0]`, `sayings[1]`, and `sayings[2]`. Note that you don't need braces around each string. The compiler can deduce that each string is intended to initialize one row of the array. The last dimension is specified to be 32, which is just sufficient to accommodate the longest string, including its terminating `\0` character. The first dimension specifies the number of strings.

When you're referring to an element of the array—`sayings[i][j]`, for instance—the first index, `i`, identifies a row in the array, and the second index, `j`, identifies a character within a row. When you want to refer to a complete row containing one of the strings, you just use a single index value between square brackets. For instance, `sayings[1]` refers to the second string in the array, `"Many hands make light work."`.

Although you must specify the last dimension in an array of strings, you can leave it to the compiler to figure out how many strings there are:

```
char sayings[][32] = {
                        "Manners maketh man.",
                        "Many hands make light work.",
                        "Too many cooks spoil the broth."
                      };
```

I've omitted the value for the size of the first dimension in the array here so the compiler will deduce this from the initializers between braces. Because you have three initializing strings, the compiler will make the first array dimension 3. Of course, you must still make sure that the last dimension is large enough to accommodate the longest string, including its terminating null character.

You could output the three sayings with the following code:

```
for(int i = 0 ; i<3 ; i++)
```

```
printf("\n%s", sayings[i]);
```

You reference a row of the array using a single index in the expression `sayings[i]`. This effectively accesses the one-dimensional array that is at index position `i` in the `sayings` array.

You could change the last example to use a two-dimensional array.

TRY IT OUT: ARRAYS OF STRINGS

Let's change the previous example so that it stores the two initial strings in a single array and incorporate the more concise coding for finding string lengths and copying strings:

```
/* Program 6.4 Arrays of strings */
#include <stdio.h>

int main(void)
{
  char str[][40] =  {
                      "To be or not to be"   ,
                      ", that is the question"
                    };
  int count[] = {0, 0};                /* Lengths of strings */

  /* find the lengths of the strings */
  for(int i = 0 ; i<2 ; i++)
    while (str[i][count[i]])
      count[i]++;

  /* Check that we have enough space for both strings  */
  if(sizeof str[0] < count[0] + count[1] + 1)
    printf("\nYou can't put a quart into a pint pot.");
  else
  {  /* Copy 2nd string to first */
    count[1] = 0;
    while((str[0][count[0]++] = str[1][count[1]++]));

    printf("\n%s\n", str[0]);    /* Output combined string */
  }
  return 0;
}
```

Typical output from this program is the following:

```
To be or not to be, that is the question
```

How It Works

You declare a single two-dimensional `char` array instead of the two one-dimensional arrays you had before:

```
char str[][40] = {
                      "To be or not to be",
                      ",that is the question"
                    };
```

The first initializing string is stored with the first index value as 0, and the second initializing string is stored with the first index value as 1. Of course, you could add as many initializing strings as you want between the braces, and the compiler would adjust the first array dimension to accommodate them.

The string lengths are now stored as elements in the `count` array. With `count` as an array we are able to find the lengths of both strings in the same loop:

```
for(int i = 0 ; i<2 ; i++)
  while (str[i][count[i]])
    count[i]++;
```

The outer `for` loop iterates of the two strings and the inner `while` loop iterates over the characters in the current string selected by `i`. This approach obviously applies to any number of strings in the `str` array; naturally the number of elements in the `count` array must be the same as the number of strings. A disadvantage of this approach is that if your strings are significantly less than 40 characters long, you waste quite a bit of memory in the array. In the next chapter you'll learn how you can avoid this and store each string in the most efficient manner.

String Library Functions

Now that you've struggled through the previous examples, laboriously copying strings from one variable to another, it's time to reveal that there's a standard library for string functions that can take care of all these little chores. Still, at least you know what's going on when you use the library functions.

The string functions are declared in the `<string.h>` header file, so you'll need to put

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

at the beginning of your program if you want to use them. The library actually contains quite a lot of functions, and your compiler may provide an even more extensive range of string library capabilities than is required by the C standard. I'll discuss just a few of the essential functions to demonstrate the basic idea and leave you to explore the rest on your own.

Copying Strings Using a Library Function

First, let's return to the process of copying the string stored in one array to another, which is the string equivalent of an assignment operation. The `while` loop mechanism you carefully created to do this must still be fresh in your mind. Well, you can do the same thing with this statement:

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

The arguments to the `strcpy()` function are `char` array names. What the function actually does is copy the string specified by the second argument to the string specified by the first argument, so in the preceding example `string2` will be copied to `string1`, replacing what was previously stored in `string1`. The copy operation will include the terminating `'\0'`. It's your responsibility to ensure that the array `string1` has sufficient space to accommodate `string2`. The function `strcpy()` has no way of checking the sizes of the arrays, so if it goes wrong it's all your fault. Obviously, the `sizeof` operator is important because you'll most likely check that everything is as it should be:

```
if(sizeof(string2) <= sizeof (string1))
  strcpy(string1, string2);
```

You execute the `strcpy()` operation only if the length of the `string2` array is less than or equal to the length of the `string1` array.

You have another function available, `strncpy()`, that will copy the first `n` characters of one string to another. The first argument is the destination string, the second argument is the source string, and the third argument is an integer of type `size_t` that specifies the number of characters to be copied. Here's an example of how this works:

```
char destination[] = "This string will be replaced";
char source[] = "This string will be copied in part";
size_t n = 26;                    /* Number of characters to be copied */
strncpy(destination, source, n);
```

After executing these statements, `destination` will contain the string `"This string will be copied"`, because that corresponds to the first 26 characters from `source`. A `'\0'` character will be appended after the last character copied. If `source` has fewer than 26 characters, the function will add `'\0'` characters to make up the count to 26.

Note that when the length of the source string is greater than the number of characters to be copied, no additional `'\0'` character is added to the destination string by the `strncpy()` function. This means that the destination string may not have a termination null character in such cases, which can cause major problems with further operations with the destination string.

Determining String Length Using a Library Function

To find out the length of a string you have the function `strlen()`, which returns the length of a string as an integer of type `size_t`. To find the length of a string in Program 6.3 you wrote this:

```
while (str2[count2])
  count2++;
```

Instead of this rigmarole, you could simply write this:

```
count2 = strlen(str2);
```

Now the counting and searching that's necessary to find the end of the string is performed by the function, so you no longer have to worry about it. Note that it returns the length of the string *excluding* the `'\0'`, which is generally the most convenient result. It also returns the value as `size_t` which corresponds to an unsigned integer type, so you may want to declare the variable to hold the result as `size_t` as well. If you don't, you may get warning messages from your compiler.

Just to remind you, type `size_t` is a type that is defined in the standard library header file `<stddef.h>`. This is also the type returned by the operator `sizeof`. The type `size_t` will be defined to be one of the unsigned integer types you have seen, typically `unsigned int`. The reason for implementing things this way is code portability. The type returned by `sizeof` and the `strlen()` function, among others, can vary from one C implementation to another. It's up to the compiler writer to decide what it should be. Defining the type to be `size_t` and defining `size_t` in a header file enables you to accommodate such implementation dependencies in your code very easily. As long as you define `count2` in the preceding example as type `size_t`, you have code that will work in every standard C implementation, even though the definition of `size_t` may vary from one implementation to another.

So for the most portable code, you should write the following:

```
size_t count2 = 0;
count2 = strlen(str2);
```

As long as you have `#include` directives for `<string.h>` and `<stddef.h>`, this code will compile with the ISO/IEC standard C compiler.

Joining Strings Using a Library Function

In Program 6.3, you copied the second string onto the end of the first using the following rather complicated looking code:

```
count2 = 0;
while(str2[count2])
  str1[count1++] = str2[count2++];
str1[count1] = '\0';
```

Well, the string library gives a slight simplification here, too. You could use a function that joins one string to the end of another. You could achieve the same result as the preceding fragment with the following exceedingly simple statement:

```
strcat(str1, str2);              /* Copy str2 to the end of str1 */
```

This function copies `str2` to the end of `str1`. The `strcat()` function is so called because it performs string catenation; in other words it joins one string onto the end of another. As well as appending `str2` to `str1`, the `strcat()` function also returns `str1`.

If you only want to append part of the source string to the destination string, you can use the `strncat()` function. This requires a third argument of type `size_t` that indicates the number of characters to be copied, for instance

```
strncat(str1, str2, 5);    /* Copy 1st 5 characters of str2 to the end of str1 */
```

As with all the operations that involve copying one string to another, it's up to you to ensure that the destination array is sufficiently large to accommodate what's being copied to it. This function and others will happily overwrite whatever lies beyond the end of your destination array if you get it wrong.

All these string functions return the destination string. This allows you to use the value returned in another string operation, for example

```
size_t length = 0;
length = strlen(strncat(str1, str2, 5));
```

Here the `strncat()` function copies five characters from `str2` to the end of `str1`. The function returns the array `str1`, so this is passed as an argument to the `strlen()` function. This will then return the length of the new version of `str1` with the five characters from `str2` appended.

TRY IT OUT: USING THE STRING LIBRARY

You now have enough tools to do a good job of rewriting Program 6.3:

```
/* Program 6.5 Joining strings - revitalized */
#include <stdio.h>
#include <string.h>
#define STR_LENGTH 40

int main(void)
{
  char str1[STR_LENGTH] = "To be or not to be";
  char str2[STR_LENGTH] = ",that is the question";

  if(STR_LENGTH > strlen(str1) + strlen(str2)) /* Enough space ?             */
    printf("\n%s\n", strcat(str1, str2));   /* yes, so display joined string */
```

```
  else
    printf("\nYou can't put a quart into a pint pot.");
  return 0;
}
```

This program will produce exactly the same output as before.

How It Works

Well, what a difference a library makes. It actually makes the problem trivial, doesn't it? You've defined a symbol for the size of the arrays using a `#define` directive. If you want to change the array sizes in the program later, you can just modify the definition for `STR_LENGTH`. You simply check that you have enough space in your array by means of the `if` statement:

```
if(STR_LENGTH > strlen(str1) + strlen(str2)) /* Enough space ?              */
  printf("\n%s\n", strcat(str1, str2));   /* yes, so display joined string   */
else
  printf("\nYou can't put a quart into a pint pot.");
```

If you do have enough space, you join the strings using the `strcat()` function within the argument to the `printf()`. Because the `strcat()` function returns `str1`, the `printf()` displays the result of joining the strings. If `str1` is too short, you just display a message. Note that the comparison uses the > operator—this is because the array length must be at least one greater than the sum of the two string lengths to allow for the terminating `'\0'` character.

Comparing Strings

The string library also provides functions for comparing strings and deciding whether one string is greater than or less than another. It may sound a bit odd applying such terms as "greater than" and "less than" to strings, but the result is produced quite simply. Successive corresponding characters of the two strings are compared based on the numerical value of their character codes. This mechanism is illustrated graphically in Figure 6-2, in which the character codes are shown as hexadecimal values.

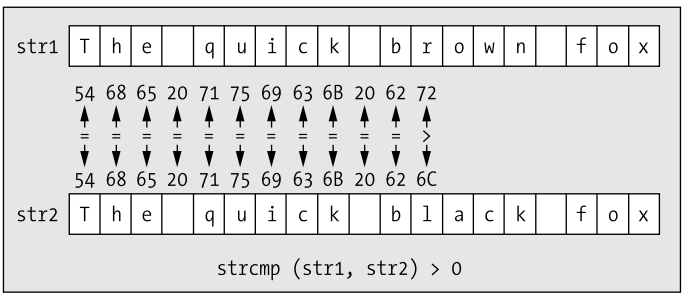

Figure 6-2. *Comparing two strings*

If two strings are identical, then of course they're equal. The first pair of corresponding characters that are different in two strings determines whether the first string is less than or greater than the second. So, for example, if the character code for the character in the first string is less than the character code for the character in the second string, the first string is less than the second. This mechanism for

comparison generally corresponds to what you expect when you're arranging strings in alphabetical order.

The function `strcmp(str1, str2)` compares two strings. It returns a value of type `int` that is less than, equal to, or greater than 0, corresponding to whether `str1` is less than, equal to, or greater than `str2`. You can express the comparison illustrated in Figure 6-2 in the following code fragment:

```
char str1[] = "The quick brown fox";
char str2[] = "The quick black fox";
if(strcmp(str1, str2) < 0)
  printf("str1 is less than str2");
```

The `printf()` statement will execute only if the `strcmp()` function returns a negative integer. This will be when the `strcmp()` function finds a pair of corresponding characters in the two strings that do not match and the character code in `str1` is less than the character code in `str2`.

The `strncmp()` function compares up to `n` characters of the two strings. The first two arguments are the same as for the `strcmp()` function and the number of characters to be compared is specified by a third argument that's an integer of type `size_t`. This function would be useful if you were processing strings with a prefix of ten characters, say, that represented a part number or a sequence number. You could use the `strncmp()` function to compare just the first ten characters of two strings to determine which should come first:

```
if(strncmp(str1, str2, 10) <= 0)
  printf("\n%s\n%s", str1, str2);
else
  printf("\n%s\n%s", str2, str1);
```

These statements output strings `str1` and `str2` arranged in ascending sequence according to the first ten characters in the strings.

Let's try comparing strings in a working example.

TRY IT OUT: COMPARING STRINGS

You can demonstrate the use of comparing strings in an example that compares just two words that you enter from the keyboard:

```
/* Program 6.6 Comparing strings */
#include <stdio.h>
#include <string.h>

int main(void)
{
  char word1[20];                /* Stores the first word  */
  char word2[20];                /* Stores the second word */

  printf("\nType in the first word (less than 20 characters):\n1: ");
  scanf("%19s", word1);          /* Read the first word    */
  printf("Type in the second word (less than 20 characters):\n 2: ");
  scanf("%19s", word2);          /* Read the second word   */

  /* Compare the two words */
  if(strcmp(word1,word2) == 0)
    printf("You have entered identical words");
```

```
  else
    printf("%s precedes %s",
                   (strcmp(word1, word2) < 0) ? word1 : word2,
                   (strcmp(word1, word2) < 0) ? word2 : word1);
  return 0;
}
```

The program will read in two words and then tell you which word comes before the other alphabetically. The output looks something like this:

```
Type in the first word (less than 20 characters):
 1: apple
Type in the second word (less than 20 characters):
 2: banana
apple precedes banana
```

How It Works

You start the program with the `#include` directives for the header files for the standard input and output library, and the string handling library:

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

In the body of `main()`, you first declare two character arrays to store the words that you'll read in from the keyboard:

```
char word1[20];                 /* Stores the first word  */
char word2[20];                 /* Stores the second word */
```

You set the size of the arrays to 20. This should be enough for an example, but there's a risk that this may not be sufficient. As with the `strcpy()` function, it's *your* responsibility to allocate enough space for what the user may key in. The function `scanf()` will limit the number of characters read if you specify a width with the format specification. While this ensures the array limit will not be exceeded, any characters in excess of the width you specify will be left in the input stream and will be read by the next input operation for the stream.

The next task is to get two words from the user; so after a prompt you use `scanf()` twice to read a couple of words from the keyboard:

```
printf("\nType in the first word (less than 20 characters):\n 1: ");
scanf("%19s", word1);           /* Read the first word    */
printf("Type in the second word (less than 20 characters):\n 2: ");
scanf("%19s", word2);           /* Read the second word   */
```

The width specification of 19 characters ensures that the array size of 20 elements will not be exceeded. Notice how in this example you haven't used an & operator before the variables in the arguments to the `scanf()` function. This is because the name of an array by itself is an address. It corresponds to the address of the first element in the array. You could write this explicitly using the & operator like this:

```
scanf("%s", &word1[0]);
```

Therefore, `&word1[0]` is equal to `word1`! I'll go into more detail on this in the next chapter.

Finally, you use the `strcmp()` function to compare the two words that were entered:

```
if(strcmp(word1,word2) == 0)
  printf("You have entered identical words");
```

```
else
  printf("%s precedes %s",
                   (strcmp(word1, word2) < 0) ? word1 : word2,
                   (strcmp(word1, word2) < 0) ? word2 : word1);
```

If the value returned by the `strcmp()` function is 0, the two strings are equal and you display a message to this effect. If not, you print out a message specifying which word precedes the other. You do this using the conditional operator to specify which word you want to print first and which you want to print second.

Searching a String

The `<string.h>` header file declares several string-searching functions, but before I get into these, we'll take a peek at the subject of the next chapter, namely pointers. You'll need an appreciation of the basics of this in order to understand how to use the string-searching functions.

The Idea of a Pointer

As you'll learn in detail in the next chapter, C provides a remarkably useful type of variable called a **pointer**. A pointer is a variable that contains an address—that is, it contains a reference to another location in memory that can contain a value. You already used an address when you used the function `scanf()`. A pointer with the name `pNumber` is defined by the second of the following two statements:

```
int Number = 25;
int *pNumber = &Number;
```

Figure 6-3 illustrates what happens when these two statements are executed.

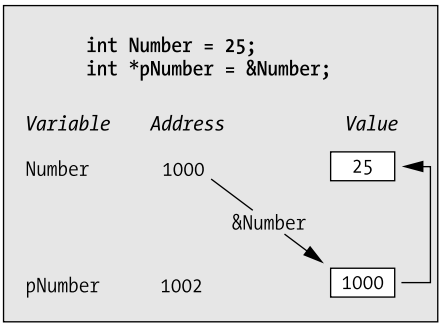

Figure 6-3. *An example of a pointer*

You declare a variable, `Number`, with the value 25, and a pointer, `pNumber`, which contains the address of `Number`. You can now use the variable `pNumber` in the expression `*pNumber` to obtain the value contained in `Number`. The `*` is the dereference operator and its effect is to access the data stored at the address specified by a pointer.

The main reason for introducing this idea here is that the functions I'll discuss in the following sections return pointers, so you could be a bit confused by them if there was no explanation here at all. If you end up confused anyway, don't worry—all will be illuminated in the next chapter.

Searching a String for a Character

The `strchr()` function searches a given string for a specified character. The first argument to the function is the string to be searched (which will be the address of a `char` array), and the second argument is the character that you're looking for. The function will search the string starting at the beginning and return a pointer to the first position in the string where the character is found. This is the address of this position in memory and is of type `char*` described as "pointer to `char`." So to store the value that's returned you must create a variable that can store an address of a character. If the character isn't found, the function will return a special value `NULL`, which is the equivalent of 0 for a pointer and represents a pointer that doesn't point to anything.

You can use the `strchr()` function like this:

```
char str[] = "The quick brown fox";  /* The string to be searched        */
char c = 'q';                        /* The character we are looking for */
char *pGot_char = NULL;              /* Pointer initialized to zero      */
pGot_char = strchr(str, c);          /* Stores address where c is found  */
```

You define the character that you're looking for by the variable `c` of type `char`. Because the `strchr()` function expects the second argument to be of type `int`, the compiler will convert the value of `c` to this type before passing it to the function.

You could just as well define `c` as type `int` like this:

```
int c = 'q';   /* Initialize with character code for q */
```

Functions are often implemented so that a character is passed as an argument of type `int` because it's simpler to work with type `int` than type `char`.

Figure 6-4 illustrates the result of this search using the `strchr()` function.

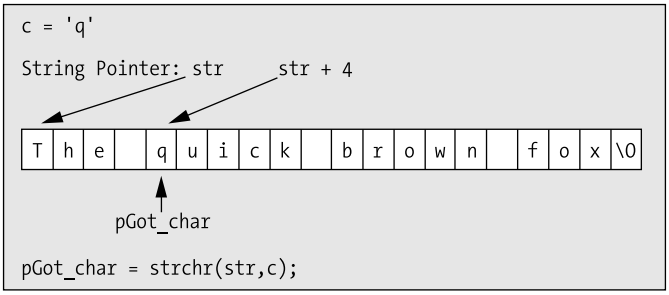

Figure 6-4. *Searching for a character*

The address of the first character in the string is given by the array name `str`. Because `'q'` appears as the fifth character in the string, its address will be `str + 4`, an offset of 4 bytes from the first character. Thus, the variable `pGot_char` will contain the address `str + 4`.

Using the variable name `pGot_char` in an expression will access the address. If you want to access the character that's stored at that address too, then you must dereference the pointer. To do this, you precede the pointer variable name with the dereference operator `*`, for example:

```
printf("Character found was %c.", *pGot_char);
```

I'll go into more detail on using the dereferencing operator further in the next chapter.

Of course, in general it's always possible that the character you're searching for might not be found in the string, so you should take care that you don't attempt to dereference a `NULL` pointer.

If you do try to dereference a NULL pointer, your program will crash. This is very easy to avoid with an if statement, like this:

```
if(pGot_char != NULL)
  printf("Character found was %c.", *pGot_char);
```

Now you only execute the printf() statement when the variable pGot_char isn't NULL.

The strrchr() function is very similar in operation to the strchr() function, except that it searches for the character starting from the end of the string. Thus, it will return the address of the *last* occurrence of the character in the string, or NULL if the character isn't found.

Searching a String for a Substring

The strstr() function is probably the most useful of all the searching functions declared in string.h. It searches one string for the first occurrence of a substring and returns a pointer to the position in the first string where the substring is found. If it doesn't find a match, it returns NULL. So if the value returned here isn't NULL, you can be sure that the searching function that you're using has found an occurrence of what it was searching for. The first argument to the function is the string that is to be searched, and the second argument is the substring you're looking for.

Here is an example of how you might use the strstr() function:

```
char text[] = "Every dog has his day";
char word[] = "dog";
char *pFound = NULL;
pFound = strstr(text, word);
```

This searches text for the first occurrence of the string stored in word. Because the string "dog" appears starting at the seventh character in text, pFound will be set to the address text + 6. The search is case sensitive, so if you search the text string for "Dog", it won't be found.

TRY IT OUT: SEARCHING A STRING

Here's some of what I've been talking about in action:

```
/* Program 6.7 A demonstration of seeking and finding  */
#include <stdio.h>
#include <string.h>
int main(void)
{
  char str1[] = "This string contains the holy grail.";
  char str2[] = "the holy grail";
  char str3[] = "the holy grill";

  /* Search str1 for the occurrence of str2 */
  if(strstr(str1, str2) == NULL)
    printf("\n\"%s\" was not found.", str2);
  else
    printf("\n\"%s\" was found in \"%s\"",str2, str1);

  /* Search str1 for the occurrence of str3 */
  if(strstr(str1, str3) == NULL)
    printf("\n\"%s\" was not found.", str3);
```