```
  else
    printf("\nWe shouldn't get to here!");
  return 0;
}
```

This program produces the following output:

```
"the holy grail" was found in "This string contains the holy grail."
"the holy grill" was not found.
```

How It Works

Note the `#include` directive for `<string.h>`. This is necessary when you want to use any of the string processing functions.

You have three strings defined: `str1`, `str2`, and `str3`:

```
char str1[] = "This string contains the holy grail.";
char str2[] = "the holy grail";
char str3[] = "the holy grill";
```

In the first `if` statement, you use the library function `strstr()` to search for the occurrence of the second string in the first string:

```
if(strstr(str1, str2) == NULL)
  printf("\n\"%s\" was not found.", str2);
else
  printf("\n\"%s\" was found in \"%s\"",str2, str1);
```

You display a message corresponding to the result by testing the returned value of `strstr()` against `NULL`. If the value returned is equal to `NULL`, this indicates the second string wasn't found in the first, so a message is displayed to that effect. If the second string *is* found, the `else` is executed. In this case, a message is displayed indicating that the string was found.

You then repeat the process in the second `if` statement and check for the occurrence of the third string in the first:

```
if(strstr(str1, str3) == NULL)
  printf("\n\"%s\" was not found.", str3);
else
  printf("\nWe shouldn't get to here!");
```

If you get output from the first or the last `printf()` in the program, something is seriously wrong.

Analyzing and Transforming Strings

If you need to examine the internal contents of a string, you can use the set of standard library functions that are declared the `<ctype.h>` header file that I introduced in Chapter 3. These provide you with a very flexible range of analytical functions that enable you to test what kind of character you have. They also have the advantage that they're independent of the character code on the computer you're using. Just to remind you, Table 6-1 shows the functions that will test for various categories of characters.

Table 6-1. *Character Classification Functions*

Function	Tests For
`islower()`	Lowercase letter
`isupper()`	Uppercase letter
`isalpha()`	Uppercase or lowercase letter
`isalnum()`	Uppercase or lowercase letter or a digit
`iscntrl()`	Control character
`isprint()`	Any printing character including space
`isgraph()`	Any printing character except space
`isdigit()`	Decimal digit (`'0'` to `'9'`)
`isxdigit()`	Hexadecimal digit (`'0'` to `'9'`, `'A'` to `'F'`, `'a'` to `'f'`)
`isblank()`	Standard blank characters (space, `'\t'`)
`isspace()`	Whitespace character (space, `'\n'`, `'\t'`, `'\v'`, `'\r'`, `'\f'`)
`ispunct()`	Printing character for which `isspace()` and `isalnum()` return `false`

The argument to a function is the character to be tested. All these functions return a nonzero value of type `int` if the character is within the set that's being tested for; otherwise, they return 0. Of course, these return values convert to `true` and `false` respectively so you can use them as Boolean values. Let's see how you can use these functions for testing the characters in a string.

TRY IT OUT: USING THE CHARACTER CLASSIFICATION FUNCTIONS

The following example determines how many digits and letters there are in a string that's entered from the keyboard:

```
/* Program 6.8 Testing characters in a string */
#include <stdio.h>
#include <ctype.h>

int main(void)
{
  char buffer[80];              /* Input buffer              */
  int i = 0;                    /* Buffer index              */
  int num_letters = 0;          /* Number of letters in input */
  int num_digits = 0;           /* Number of digits in input  */

  printf("\nEnter an interesting string of less than 80 characters:\n");
  gets(buffer);                 /* Read a string into buffer  */


  while(buffer[i] != '\0')
  {
    if(isalpha(buffer[i]))
      num_letters++;            /* Increment letter count     */
```

```
    if(isdigit(buffer[i++]))
      num_digits++;             /* Increment digit count      */
  }
  printf("\nYour string contained %d letters and %d digits.\n",
                                          num_letters, num_digits);
  return 0;
}
```

The following is typical output from this program:

```
Enter an interesting string of less than 80 characters:
I was born on the 3rd of October 1895

Your string contained 24 letters and 5 digits.
```

How It Works

This example is quite straightforward. You read the string into the array, `buffer`, with the following statement:

```
gets(buffer);
```

The string that you enter is read into the array `buffer` using a new standard library function, `gets()`. So far, you've used only `scanf()` to accept input from the keyboard, but it's not very useful for reading strings because it interprets a space as the end of an input value. The `gets()` function has the advantage that it will read all the characters entered from the keyboard, including blanks, up to when you press the Enter key. This is then stored as a string into the area specified by its argument, which in this case is the `buffer` array. A `'\0'` will be appended to the string automatically.

As with any input or output operation, things can go wrong. If an error of some kind prevents the `gets()` function from reading the input successfully, it will return `NULL` (normally, it returns the address passed as the argument—`buffer`, in this case). You could therefore check that the read operation was successful using the following code fragment:

```
if(gets(buffer) == NULL)
{
  printf("Error reading input.");
  return 1;                       /* End the program */
}
```

This will output a message and end the program if the read operation fails for any reason. Errors on keyboard input are relatively rare, so you won't include this testing when you're reading from the keyboard in your examples; but if you are reading from a file, verifying that the read was successful is essential.

A disadvantage of the `gets()` function is that it will read a string of any length and attempt to store it in `buffer`. There is no check that `buffer` has sufficient space to store the string so there's another opportunity to crash the program. To avoid this you could use the `fgets()` function, which allows you to specify the maximum length of the input string. This is a function that is used for any kind of input stream, as opposed to `gets()` which only reads from the standard input stream `stdin`; so you also have to specify a third argument to `fgets()` indicating the stream that is to be read. Here's how you could use `fgets()` to read a string from the keyboard:

```
if(fgets(buffer, sizeof(buffer), stdin) == NULL)
{
  printf("Error reading input.");
  return 1;                       /* End the program */
}
```

The `fgets()` function reads a maximum of one less than the number of characters specified by the second argument. It then appends a `\0` character to the end of the string in memory, so the second argument in this case is `sizeof(buffer)`. Note that there is another important difference between `fgets()` and `gets()`. For both functions, reading a newline character ends the input process, but `fgets()` stores a `'\n'` character when a newline is entered, whereas `gets()` does not. This means that if you are reading strings from the keyboard, strings read by `fgets()` will be one character longer than strings read by `gets()`. It also means that just pressing the Enter key as the input will result in an empty string `"\0"` with `gets()`, but will result in the string `"\n\0"` with `fgets()`. You'll use `fgets()` in the next example in this chapter, Program 6.9, where you have to take account of the newline character that is stored as part of the string. You'll also see more about the `fgets()` function in Chapter 12.

The statements that analyze the string are as follows:

```
while(buffer[i] != '\0')
{
  if(isalpha(buffer[i]))
    num_letters++;                /* Increment letter count     */
  if(isdigit(Buffer[i++]))
    num_digits++;                 /* Increment digit count      */
}
```

The input string is tested character by character in the `while` loop. Checks are made for alphabetic characters and digits in the two `if` statements. When either is found, the appropriate counter is incremented. Note that you increment the index to the `buffer` array in the second `if`. Remember, because you're using the postfix form of the increment operator, the check is made using the current value of `i`, and then `i` is incremented.

You could implement this without using `if` statements:

```
while(buffer[i] != '\0')
{
  num_letters += isalpha(buffer[i]) != 0;
  num_digits += isdigit(buffer[i++]) != 0;
}
```

The test functions return a nonzero value (not necessarily 1, though) if the argument belongs to the group of characters being tested for. The value of the logical expressions to the right of the assignment operators will be `true` if the character does belong to the category you're testing for; otherwise, it will be `false`.

The way you've coded the example isn't a particularly efficient way of doing things, because you test for a digit even if you've already discovered the current character is alphabetic. You could try to improve on this if the TV is really bad one night.

Converting Characters

You've already seen that the standard library also includes two conversion functions that you get access to through `<ctype.h>`. The `toupper()` function converts from lowercase to uppercase, and the `tolower()` function does the reverse. Both functions return either the converted character or the same character for characters that are already in the correct case. You can therefore convert a string to uppercase using this statement:

```
for(int i = 0 ; (buffer[i] = toupper(buffer[i])) != '\0' ; i++);
```

This loop will convert the entire string to uppercase by stepping through the string one character at a time, converting lowercase to uppercase and leaving uppercase characters unchanged. The loop stops when it reaches the string termination character `'\0'`. This sort of pattern in which everything is done inside the loop control expressions is quite common in C.

Let's try a working example that applies these functions to a string.

TRY IT OUT: CONVERTING CHARACTERS

You can use the function `toupper()` in combination with the `strstr()` function to find out whether one string occurs in another, ignoring case. Look at the following example:

```
/* Program 6.9 Finding occurrences of one string in another  */
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main(void)
{
  char text[100];                /* Input buffer for string to be searched */
  char substring[40];            /* Input buffer for string sought         */

  printf("\nEnter the string to be searched (less than 100 characters):\n");
  fgets(text, sizeof(text), stdin);

  printf("\nEnter the string sought (less than 40 characters):\n");
  fgets(substring, sizeof(substring), stdin);

  /* overwrite the newline character in each string */
  text[strlen(text)-1] = '\0';
  substring[strlen(substring)-1] = '\0';

  printf("\nFirst string entered:\n%s\n", text);
  printf("\nSecond string entered:\n%s\n", substring);

  /* Convert both strings to uppercase. */
  for(int i = 0 ; (text[i] = toupper(text[i])) ; i++);
  for(int i = 0 ; (substring[i] = toupper(substring[i])) ; i++);

    printf("\nThe second string %s found in the first.",
            ((strstr(text, substring) == NULL) ? "was not" : "was"));
  return 0;
}
```

Typical operation of this example will produce the following:

```
Enter the string to be searched(less than 100 characters):
Cry havoc, and let slip the dogs of war.

Enter the string sought (less than 40 characters ):
The Dogs of War

First string entered:
Cry havoc, and let slip the dogs of war

Second string entered:
The Dogs of War

The second string was found in the first.
```

How It Works

This program has three distinct phases: getting the input strings, converting both strings to uppercase, and searching the first string for an occurrence of the second.

First of all, you use `printf()` to prompt the user for the input, and you use the `fgets()` function introduced in the discussion of the previous example to read the input into `text` and `substring`:

```
printf("\nEnter the string to be searched(less than 100 characters):\n");
fgets(text. sizeof(text), stdin);
printf("\nEnter the string sought (less than 40 characters ):\n");
gets(substring, sizeof(substring), stdin);
```

You use the `fgets()` function here because it will read in any string from the keyboard, including spaces, the input being terminated when the Enter key is pressed. The input process will only allow 99 characters to be entered for the first string, `text`, and 39 characters for the second string, `substring`. If more characters are entered they will be ignored so the operation of the program is safe.

You'll recall that `fgets()` stores the newline character that ends the input process. This doesn't matter particularly for the first string but it matters a lot for the second string you are searching for. For example, if the string you want to find is `"dogs"`, the `fgets()` function will actually store `"dogs\n"`, which is not the same at all. You therefore remove the newline from each string by overwriting it with a `'\0'` character:

```
text[strlen(text)-1] = '\0';
substring[strlen(substring)-1] = '\0';
```

The newline character is the next to last character in each string and the index for this position is the string length less 1.

Of course, if you exceed the limits for input, the strings will be truncated and the results are unlikely to be correct. This will be evident from the listing of the two strings that is produced by the following:

```
printf("\nFirst string entered:\n%s\n", text);
printf("\nSecond string entered:\n%s\n", substring);
```

The conversion of both strings to uppercase is accomplished using the following statements:

```
for(int i = 0 ; (text[i] = toupper(text[i])) ; i++);
for(int i = 0 ; (substring[i] = toupper(substring[i])) ; i++);
```

You use `for` loops to do the conversion and the work is done entirely within the control expressions for the loops. The first `for` loop initializes `i` to 0, and then converts the `i`th character of `text` to uppercase in the loop condition and stores that result back in the same position in `text`. The loop continues as long as the character code stored in `text[i]` in the second loop control expression is nonzero, which will be for any character except `NULL`. The index `i` is incremented in the third loop control expression. This ensures that there's no confusion as to when the incrementing of `i` takes place. The second loop works in exactly the same way to convert `substring` to uppercase.

With both strings in uppercase, you can test for the occurrence of `substring` in `text`, regardless of the case of the original strings. The test is done inside the output statement that reports the result:

```
printf("\nThe second string %s found in the first.",
         ((strstr(text, substring) == NULL) ? "was not" : "was"));
```

The conditional operator chooses either `"was not"` or `"was"` to be part of the output string, depending on whether the `strstr()` function returns `NULL`. You saw earlier that the `strstr()` function returns `NULL` when the string specified by the second argument isn't found in the first. Otherwise, it returns the address where the string was found.

Converting Strings to Numerical Values

The <stdlib.h> header file declares functions that you can use to convert a string to a numerical value. Each of the functions in Table 6-2 requires an argument that's a pointer to a string or an array of type char that contains a string that's a representation of a numerical value.

Table 6-2. *Functions That Convert Strings to Numerical Values*

Function	Returns
atof()	A value of type double that is produced from the string argument
atoi()	A value of type int that is produced from the string argument
atol()	A value of type long that is produced from the string argument
atoll()	A value of type long long that is produced from the string argument

These functions are very easy to use, for example

```
char value_str[] = "98.4";
double value = 0;
value = atof(value_str);          /* Convert string to floating-point */
```

The value_str array contains a string representation of a value of type double. You pass the array name as the argument to the atof() function to convert it to type double. You use the other three functions in a similar way.

These functions are particularly useful when you need to read numerical input in the format of a string. This can happen when the sequence of the data input is uncertain, so you need to analyze the string in order to determine what it contains. Once you've figured out what kind of numerical value the string represents, you can use the appropriate library function to convert it.

Working with Wide Character Strings

Working with wide character strings is just as easy as working with the strings you have been using up to now. You store a wide character string in an array of elements of type wchar_t and a wide character string constant just needs the L modifier in front of it. Thus you can declare and initialize a wide character string like this:

```
wchar_t proverb[] = L"A nod is as good as a wink to a blind horse.";
```

As you saw back in Chapter 2, a wchar_t character occupies 2 bytes. The proverb string contains 44 characters plus the terminating null, so the string will occupy 90 bytes.

If you wanted to write the proverb string to the screen using printf() you must use the %S format specifier rather than %s that you use for ASCII string. If you use %s, the printf() function will assume the string consists of single-byte characters so the output will not be correct. Thus the following statement will output the wide character string correctly:

```
printf("The proverb is:\n%S", proverb);
```

Operations on Wide Character Strings

The <wchar.h> header file declares a range of functions for operating on wide character strings that parallel the functions you have been working with that apply to ordinary strings. Table 6-3 shows the functions declared in <wchar.h> that are the wide character equivalents to the string functions I have already discussed in this chapter.

Table 6-3. *Functions That Operate on Wide Character Strings*

Function	Description
`wcslen(const wchar_t* ws)`	Returns a value of type `size_t` that is the length of the wide character string `ws` that you pass as the argument. The length excludes the termination `L'\0'` character.
`wcscpy(wchar_t* destination, const wchar_t source)`	Copies the wide character string `source` to the wide character string `destination`. The function returns `source`.
`wcsncpy(wchar_t* destination, const wchar_t source, size_t n)`	Copies `n` characters from the wide character string `source` to the wide character string `destination`. If `source` contains less than `n` characters, `destination` is padded with `L'\0'` characters. The function returns `source`.
`wcscat(whar_t* ws1, whar_t* ws2)`	Appends a copy of `ws2` to `ws1`. The first character of `ws2` overwrites the terminating null at the end of `ws1`. The function returns `ws1`.
`wcsncmp(const wchar_t* ws1, const wchar_t* ws2)`	Compares the wide character string pointed to by `ws1` with the wide character string pointed to by `ws2` and returns a value of type `int` that is less than, equal to, or greater than 0 if the string `ws1` is less than, equal to, or greater than the string `ws2`.
`wcscmp(const wchar_t* ws1, const wchar_t* ws2, size_t n)`	Compares up to `n` characters from the wide character string pointed to by `ws1` with the wide character string pointed to by `ws2`. The function returns a value of type `int` that is less than, equal to, or greater than 0 if the string of up to `n` characters from `ws1` is less than, equal to, or greater than the string of up to `n` characters from `ws2`.
`wcschr(const wchar_t* ws, wchar_t wc)`	Returns a pointer to the first occurrence of the wide character, `wc`, in the wide character string pointed to by `ws`. If `wc` is not found in `ws`, the `NULL` pointer value is returned.
`wcsstr(const wchar_t* ws1, const wchar_t* ws2)`	Returns a pointer to the first occurrence of the wide character string `ws2` in the wide character string `ws1`. If `ws2` is not found in `ws1`, the `NULL` pointer value is returned.

As you see from the descriptions, all these functions work in essentially the same way as the string functions you have already seen. Where the `const` keyword appears in the specification of the type of argument you can supply to a function, it implies that the argument will not be modified by the function. This forces the compiler to check that the function does not attempt to change such arguments. You'll see more on this in Chapter 7 when you explore how you create your own functions in more detail.

The `<wchar.h>` header also declares the `fgetws()` function that reads a wide character string from a stream such as `stdin`, which by default corresponds to the keyboard. You must supply three arguments to the `fgetws()` function, just like the `fgets()` function you use for reading for single-byte strings:

- The first argument is a pointer to an array of `wchar_t` elements that is to store the string.
- The second argument is a value `n` of type `size_t` that is the maximum number of characters that can be stored in the array.
- The third argument is the stream from which the data is to be read, which will be `stdin` when you are reading a string from the keyboard.

The function reads up to `n-1` characters from the stream and stores them in the array with an `L'\0'` appended. Reading a newline in less than `n-1` characters from the stream signals the end of input. The function returns a pointer to the array containing the string.

Testing and Converting Wide Characters

The `<wchar.h>` header also declares functions to test for specific subsets of wide characters, analogous to the functions you have seen for characters of type `char`. These are shown in Table 6.4.

Table 6-4. *Wide Character Classification Functions*

Function	Tests For
`iswlower()`	Lowercase letter
`iswupper()`	Uppercase letter
`iswalnum()`	Uppercase or lowercase letter
`iswcntrl()`	Control character
`iswprint()`	Any printing character including space
`iswgraph()`	Any printing character except space
`iswdigit()`	Decimal digit (`L'0'` to `L'9'`)
`iswxdigit()`	Hexadecimal digit (`L'0'` to `L'9'`, `L'A'` to `L'F'`, `L'a'` to `L'f'`)
`iswblank()`	Standard blank characters (space, `L'\t'`)
`iswspace()`	Whitespace character (space, `L'\n'`, `L'\t'`, `L'\v'`, `L'\r'`, `L'\f'`)
`iswpunct()`	Printing character for which `iswspace()` and `iswalnum()` return `false`

You also have the case-conversion functions, `towlower()` and `towupper()`, that return the lowercase or uppercase equivalent of the `wchar_t` argument.

You can see some of the wide character functions in action with a wide character version of Program 6.9.

TRY IT OUT: CONVERTING WIDE CHARACTERS

This example uses the wide character equivalents of `fgets()`, `toupper()`, and `wcsstr()`. The code that has changed from Program 6.9 is shown in bold type.

```
/* Program 6.9A Finding occurrences of one wide character string in another  */
#include <stdio.h>
#include <wchar.h>

int main(void)
{
  wchar_t text[100];             /* Input buffer for string to be searched */
  wchar_t substring[40];         /* Input buffer for string sought         */

  printf("\nEnter the string to be searched(less than 100 characters):\n");
  fgetws(text, 100, stdin);
  printf("\nEnter the string sought (less than 40 characters ):\n");
  fgetws(substring, 40, stdin);

  /* overwrite the newline character in each string */
  text[wcslen(text)-1] = L'\0';
  substring[wcslen(substring)-1] = L'\0';

  printf("\nFirst string entered:\n%S\n", text);
  printf("\nSecond string entered:\n%S\n", substring);

  /* Convert both strings to uppercase. */
  for(int i = 0 ; (text[i] = towupper(text[i])) ; i++);
  for(int i = 0 ; (substring[i] = towupper(substring[i])) ; i++);

    printf("\nThe second string %s found in the first.",
              ((wcsstr(text, substring) == NULL) ? "was not" : "was"));
  return 0;
}
```

The output will be the same as for the previous example.

How It Works

This works in the same way as the previous example except that it stores the input as wide character strings and makes use of wide character functions. The example is so similar there is not much to say about it. Of course, the arrays now have elements of type `wchar_t` and the names of the functions are slightly different. Reading from the keyboard into the wide character arrays is accomplished by the `fgetws()` function where you supply the limit on the number of characters that can be stored and the name of the stream as the second and third arguments. We replace the newline character in each string with the wide character version of the null terminator, `L'\0'`. Prefixing a character literal with `L` makes it a literal of type `wchar_t`. Of course, the statements that output the strings use `%S` because we are outputting wide character strings.

Designing a Program

You've almost come to the end of this chapter. All that remains is to go through a larger example to use some of what you've learned so far.

The Problem

You are going to develop a program that will read a paragraph of text of an arbitrary length that is entered from the keyboard, and determine the frequency of which each word in the text occurs, ignoring case. The paragraph length won't be completely arbitrary, as you'll have to specify some limit for the array size within the program, but you can make the array that holds the text as large as you want.

The Analysis

To read the paragraph from the keyboard, you need to be able to read input lines of arbitrary length and assemble them into a single string that will ultimately contain the entire paragraph. You don't want lines truncated either, so `fgets()` looks like a good candidate for the input operation. If you define a symbol at the beginning of the code that specifies the array size to store the paragraph, you will be able to change the capacity of the program by changing the definition of the symbol.

The text will contain punctuation, so you will have to deal with that somehow if you are to be able to separate one word from another. It would be easy to extract the words from the text if each word is separated from the next by one or more spaces. You can arrange for this by replacing all characters that are not characters that appear in a word with spaces. You'll remove all the punctuation and any other odd characters that are lying around in the text. We don't need to retain the original text, but if you did you could just make a copy before eliminating the punctuation.

Separating out the words will be simple. All you need to do is extract each successive sequence of characters that are not spaces as a word. You can store the words in another array. Since you want to count word occurrences, ignoring case, you can store each word as lowercase. As you find a new word, you'll have to compare it with all the existing words you have found to see if it occurs previously. You'll only store it in the array if it is not already there. To record the number of occurrences of each word, you'll need another array to store the word counts. This array will need to accommodate as many counts as the number of words you have provided for in the program.

The Solution

This section outlines the steps you'll take to solve the problem. The program boils down to a simple sequence of steps that are more or less independent of one another. At the moment, the approach to implementing the program will be constrained by what you have learned up to now, and by the time you get to Chapter 9 you'll be able to implement this much more efficiently.

Step 1

The first step is to read the paragraph from the keyboard. As this is an arbitrary number of input lines it will be necessary to involve an indefinite loop. Let's first define the variables that we'll be using to code up the input mechanism:

```
/* Program 6.10 Analyzing text */
#include <stdio.h>
#include <string.h>

#define TEXTLEN  10000      /* Maximum length of text            */
#define BUFFERSIZE 100      /* Input buffer size                 */

int main(void)
{
  char text[TEXTLEN+1];
  char buffer[BUFFERSIZE];
  char endstr[] = "*\n";          /* Signals end of input       */

  printf("Enter text on an arbitrary number of lines.");
  printf("\nEnter a line containing just an asterisk to end input:\n\n");

  /* Read an arbitrary number of lines of text */
  while(true)
  {
    /* A string containing an asterisk followed by newline */
    /* signals end of input                                */
    if(!strcmp(fgets(buffer, BUFFERSIZE, stdin), endstr))
      break;

    /* Check if we have space for latest input */
    if(strlen(text)+strlen(buffer)+1 > TEXTLEN)
      {
        printf("Maximum capacity for text exceeded. Terminating program.");
        return 1;
      }
    strcat(text, buffer);
  }

  /* Plus the rest of the program code ... */

  return 0;
}
```

You can compile and run this code as it stands if you like. The symbols `TEXTLEN` and `BUFFERSIZE` specify the capacity of the `text` array and the `buffer` array respectively. The `text` array will store the entire paragraph, and the `buffer` array stores a line of input. We need some way for the user to tell the program when he is finished entering text. As the initial prompt for input indicates, entering a single asterisk on a line will do this. The single asterisk input will be read by the `fgets()` function as the string `"*\n"` because the function stores newline characters that arise when the Enter key is pressed. The `endstr` array stores the string that marks the end of the input so you can compare each input line with this array.

The entire input process takes place within the indefinite `while` loop that follows the prompt for input. A line of input is read in the `if` statement:

```
if(!strcmp(fgets(buffer, BUFFERSIZE, stdin), endstr))
  break;
```

The `fgets()` function reads a maximum of `BUFFERSIZE-1` characters from `stdin`. If the user enters a line longer than this, it won't really matter. The characters that are in excess of `BUFFERSIZE-1` will be left in the input stream and will be read on the next loop iteration. You can check that this works by setting BUFFERSIZE at 10, say, and entering lines longer than ten characters.

Because the fgets() function returns a pointer to the string that you pass as the first argument, you can use fgets() as the argument to the strcmp() function to compare the string that was read with endstr. Thus, the if statement not only reads a line of input, it also checks whether the end of the input has been signaled by the user.

Before you append the new line of input to what's already stored in text, you check that there is still sufficient free space in text to accommodate the additional line. To append the new line, just use the strcat() library function to concatenate the string stored in buffer with the existing string in text.

Here's an example of output that results from executing this input operation:

```
Enter text on an arbitrary number of lines.
Enter a line containing just an asterisk to end input:

Mary had a little lamb,
Its feet were black as soot,
And into Mary's bread and jam,
His sooty foot he put.
  *
```

Step 2

Now that you have read all the input text, you can replace the punctuation and any newline characters recorded by the fgets() function by spaces. The following code goes immediately before the return statement at the end of the previous version of main():

```
/* Replace everything except alpha and single quote characters by spaces */
for(int i = 0 ; i < strlen(text) ; i++)
{
  if(text[i] == quote || isalnum(text[i]))
    continue;
  text[i] = space;
}
```

The loop iterates over the characters in the string stored in the text array. We are assuming that words can only contain letters, digits, and single-quote characters, so anything that is not in this set is replaced by a space character. The isalnum() that returns true for a character that is a letter or a digit is declared in the <ctype.h> header file so you must add an #include statement for this to the program. You also need to add declarations for the variables quote and space, following the declaration for endstr:

```
const char space = ' ';
const char quote = '\'';
```

You could, of course, use character literals directly in the code, but defining variables like this helps to make the code a little more readable.

Step 3

The next step is to extract the words from the text array and store them in another array. You can first add a couple more definitions for symbols that relate to the array you will use to store the words. These go immediately after the definition for BUFFERSIZE:

```
#define MAXWORDS   500      /* Maximum number of different words */
#define WORDLEN     15      /* Maximum word length               */
```

You can now add the declarations for the additional arrays and working storage that you'll need for extracting the words from the text, and you can put these after the existing declarations at the beginning of main():

```
char words[MAXWORDS][WORDLEN+1];
int nword[MAXWORDS];             /* Number of word occurrences */
char word[WORDLEN+1];            /* Stores a single word       */
int wordlen = 0;                 /* Length of a word           */
int wordcount = 0;               /* Number of words stored     */
```

The words array stores up to MAXWORDS word strings of length WORDLEN, excluding the terminating null. The nword array hold counts of the number of occurrences of the corresponding words in the words array. Each time you find a new word, you'll store it in the next available position in the words array and set the element in the nword array that is at the same index position to 1. When you find a word that you have found and stored previously in words, you just need to increment the corresponding element in the nword array.

You'll extract words from the text array in another indefinite while loop because you don't know in advance how many words there are. There is quite a lot of code in this loop so we'll put it together incrementally. Here's the initial loop contents:

```
/* Find unique words and store in words array */
int index = 0;
while(true)
{
  /* Ignore any leading spaces before a word */
  while(text[index] == space)
    ++index;

  /* If we are at the end of text, we are done */
  if(text[index] == '\0')
    break;

  /* Extract a word */
  wordlen = 0;          /* Reset word length */
  while(text[index] == quote || isalpha(text[index]))
  {
    /* Check if word is too long */
    if(wordlen == WORDLEN)
    {
      printf("Maximum word length exceeded. Terminating program.");
      return 1;
    }
    word[wordlen++] = tolower(text[index++]);  /* Copy as lowercase     */
  }
  word[wordlen] = '\0';                        /* Add string terminator */
}
```

This code follows the existing code in main(), immediately before the return statement at the end.

The index variable records the current character position in the text array. The first operation within the outer loop is to move past any spaces that are there so that index refers to the first character of a word. You do this in the inner while loop that just increments index as long as the current character is a space.

It's possible that the end of the string in `text` has been reached, so you check for this next. If the current character at position `index` is `'\0'`, you exit the loop because all words must have been extracted.

Extracting a word just involves copying any character that is alphanumeric or a single quote. The first character that is not one of these marks the end of a word. You copy the characters that make up the word into the `word` array in another `while` loop, after converting each character to lowercase using the `tolower()` function from the standard library. Before storing a character in `word`, you check that the size of the array will not be exceeded. After the copying process, you just have to append a terminating null to the characters in the `word` array.

The next operation to be carried out in the loop is to see whether the word you have just extracted already exists in the `words` array. The following code does this and goes immediately before the closing brace for the `while` loop in the previous code fragment:

```
/* Check for word already stored */
bool isnew = true;
for(int i = 0 ; i< wordcount ; i++)
  if(strcmp(word, words[i]) == 0)
  {
    ++nword[i];
    isnew = false;
    break;
  }
```

The `isnew` variable records whether the word is present and is first initialized to indicate that the latest word you have extracted is indeed a new word. Within the `for` loop you compare `word` with successive strings in the `words` array using the `strcmp()` library function that compares two strings. The function returns 0 if the strings are identical; as soon as this occurs you set `isnew` to `false`, increment the corresponding element in the `nword` array, and exit the `for` loop.

The last operation within the indefinite loop that extracts words from `text` is to store the latest word in the `words` array, but only if it is new, of course. The following code does this:

```
if(isnew)
{
  /* Check if we have space for another word */
  if(wordcount >= MAXWORDS)
  {
    printf("\n Maximum word count exceeded. Terminating program.");
    return 1;
  }

  strcpy(words[wordcount], word);   /* Store the new word  */
  nword[wordcount++] = 1;           /* Set its count to 1  */
}
```

This code also goes after the previous code fragment, but before the closing brace in the indefinite `while` loop. If the `isnew` indicator is `true`, you have a new word to store, but first you verify that there is still space in the `words` array. The `strcpy()` function copies the string in `word` to the element of the `words` array selected by `wordcount`. You then set the value of the corresponding element of the `nword` array that holds the count of the number of times a word has been found in the text.

Step 4

The last code fragment that you need will output the words and their frequencies of occurrence. Following is a complete listing of the program with the additional code from steps 3 and 4 highlighted in bold font:

```
/* Program 6.10 Analyzing text */
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
#include <ctype.h>

#define TEXTLEN  10000      /* Maximum length of text            */
#define BUFFERSIZE 100      /* Input buffer size                 */
#define MAXWORDS   500      /* Maximum number of different words */
#define WORDLEN     15      /* Maximum word length               */

int main(void)
{
  char text[TEXTLEN+1];
  char buffer[BUFFERSIZE];
  char endstr[] = "*\n";          /* Signals end of input       */

  const char space = ' ';
  const char quote = '\'';

  char words[MAXWORDS][WORDLEN+1];
  int nword[MAXWORDS];            /* Number of word occurrences */
  char word[WORDLEN+1];           /* Stores a single word       */
  int wordlen = 0;                /* Length of a word           */
  int wordcount = 0;              /* Number of words stored     */

  printf("Enter text on an arbitrary number of lines.");
  printf("\nEnter a line containing just an asterisk to end input:\n\n");

  /* Read an arbitrary number of lines of text */
  while(true)
  {
    /* A string containing an asterisk followed by newline */
    /* signals end of input                                */
    if(!strcmp(fgets(buffer, BUFFERSIZE, stdin), endstr))
      break;

    /* Check if we have space for latest input */
    if(strlen(text)+strlen(buffer)+1 > TEXTLEN)
      {
        printf("Maximum capacity for text exceeded. Terminating program.");
        return 1;
      }
    strcat(text, buffer);
  }

  /* Replace everything except alpha and single quote characters by spaces */
  for(int i = 0 ; i < strlen(text) ; i++)
  {
    if(text[i] == quote || isalnum(text[i]))
      continue;
    text[i] = space;
  }
```

```
/* Find unique words and store in words array */
int index = 0;
while(true)
{
  /* Ignore any leading spaces before a word */
  while(text[index] == space)
    ++index;

  /* If we are at the end of text, we are done */
  if(text[index] == '\0')
    break;

  /* Extract a word */
  wordlen = 0;          /* Reset word length */
  while(text[index] == quote || isalpha(text[index]))
  {
    /* Check if word is too long */
    if(wordlen == WORDLEN)
    {
      printf("Maximum word length exceeded. Terminating program.");
      return 1;
    }
    word[wordlen++] = tolower(text[index++]);  /* Copy as lowercase     */
  }
  word[wordlen] = '\0';                        /* Add string terminator */

  /* Check for word already stored */
  bool isnew = true;
  for(int i = 0 ; i< wordcount ; i++)
    if(strcmp(word, words[i]) == 0)
    {
      ++nword[i];
      isnew = false;
      break;
    }

  if(isnew)
  {
    /* Check if we have space for another word */
    if(wordcount >= MAXWORDS)
    {
      printf("\n Maximum word count exceeded. Terminating program.");
      return 1;
    }

    strcpy(words[wordcount], word);   /* Store the new word  */
    nword[wordcount++] = 1;           /* Set its count to 1  */
  }
}
```

```
  /* Output the words and frequencies */
  for(int i = 0 ; i<wordcount ; i++)
  {
    if( !(i%3) )                       /* Three words to a line */
      printf("\n");
    printf("  %-15s%5d", words[i], nword[i]);
  }

  return 0;
}
```

The seven lines highlighted in bold output the words and corresponding frequencies. This is very easily done in a `for` loop that iterates over the number of words. The loop code arranges for three words plus frequencies to be output per line by writing a newline character to `stdout` if the current value of `i` is a multiple of 3. The expression `i%3` will be zero when `i` is a multiple of 3, and this value maps to the `bool` value `false`, so the expression `!(i%3)` will be `true`.

The program ends up as a `main()` function of more than 100 statements. When you learn the complete C language you would organize this program very differently with the code segmented into several much shorter functions. By Chapter 9 you'll be in a position to do this, and I would encourage you to revisit this example when you reach the end of Chapter 9. Here's a sample of output from the complete program:

```
Enter text on an arbitrary number of lines.
Enter a line containing just an asterisk to end input:

When I makes tea I makes tea, as old mother Grogan said.
And when I makes water I makes water.
Begob, ma'am, says Mrs Cahill, God send you don't make them in the same pot.
*

  when               2  i                  4  makes              4
  tea                2  as                 1  old                1
  mother             1  grogan             1  said               1
  and                1  water              2  begob              1
  ma'am              1  says               1  mrs                1
  cahill             1  god                1  send               1
  you                1  don't              1  make               1
  them               1  in                 1  the                1
  same               1  pot                1
```

Summary

In this chapter, you applied the techniques you acquired in earlier chapters to the general problem of dealing with character strings. Strings present a different, and perhaps more difficult, problem than numeric data types.

Most of the chapter dealt with handling strings using arrays, but I also mentioned pointers. These will provide you with even more flexibility in dealing with strings, and many other things besides, as you'll discover as soon as you move on to the next chapter.

Exercises

The following exercises enable you to try out what you've learned in this chapter. If you get stuck, look back over the chapter for help. If you're still stuck, you can download the solutions from the Source Code/Downloads section of the Apress web site (`http://www.apress.com`), but that really should be a last resort.

Exercise 6-1. Write a program that will prompt for and read a positive integer less than 1000 from the keyboard, and then create and output a string that is the value of the integer in words. For example, if **941** is entered, the program will create the string `"Nine hundred and forty one"`.

Exercise 6-2. Write a program that will allow a list of words to be entered separated by commas, and then extract the words and output them one to a line, removing any leading or trailing spaces. For example, if the input is

```
John  ,  Jack ,   Jill
```

then the output will be

```
John
Jack
Jill
```

Exercise 6-3. Write a program that will output a randomly chosen thought for the day from a set of at least five thoughts of your own choosing.

Exercise 6-4. A **palindrome** is a phrase that reads the same backward as forward, ignoring whitespace and punctuation. For example, "Madam, I'm Adam" and "Are we not drawn onward, we few? Drawn onward to new era?" are palindromes. Write a program that will determine whether a string entered from the keyboard is a palindrome.