Chapter 17 Manipulating Strings

17-1. String Length and Storage

You should remember from earlier chapters that, in C programs, a string is a sequence of characters, with its beginning indicated by a pointer and its end marked by the null character \0. At times, you need to know the length of a string (the number of characters between the start and the end of the string). This length is obtained with the library function strlen(). Its prototype, in STRING.H, is

```
size_t strlen(char *str);
```

You might be puzzling over the size_t return type. This type is defined in STRING.H as unsigned, so the function strlen() returns an unsigned integer. The size_t type is used with many of the string functions. Just remember that it means unsigned.

The argument passed to strlen is a pointer to the string of which you want to know the length. The function strlen() returns the number of characters between str and the next null character, not counting the null character. Listing 17.1 demonstrates strlen().

Listing 17.1. Using the strlen() function to determine the length of a string.

```
1:  /* Using the strlen() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      size_t length;
9:      char buf[80];
10:
11:     while (1)
12:     {
13:         puts("\nEnter a line of text; a blank line terminates.");
14:         gets(buf);
15:
16:         length = strlen(buf);
17:
18:         if (length != 0)
19:             printf("\nThat line is %u characters long.", length);
20:         else
21:             break;
22:     }
23:      return(0);
24: }
Enter a line of text; a blank line terminates.
Just do it!
That line is 11 characters long.
Enter a line of text; a blank line terminates.
```

ANALYSIS:: This program does little more than demonstrate the use of strlen(). Lines 13 and 14 display a message and get a string called buf. Line 16 uses strlen() to assign the length of buf to the variable length. Line 18 checks whether the string was blank by checking for a length of 0. If the string wasn't blank, line 19 prints the string's size.

17-2. Copying Strings

The C library has three functions for copying strings. Because of the way C handles strings, you can't simply assign one string to another, as you can in some other computer languages. You must copy the source string from its location in memory to the memory location of the destination string. The string-copying functions are strcpy(), strncpy(), and strdup(). All of the string-copying functions require the header file STRING.H.

17-2-1. The strcpy() Function

The library function strcpy() copies an entire string to another memory location. Its prototype is as follows:

```
char *strcpy( char *destination, char *source );
```

The function strcpy() copies the string (including the terminating null character \0) pointed to by source to the location pointed to by destination. The return value is a pointer to the new string, destination.

When using strcpy(), you must first allocate storage space for the destination string. The function has no way of knowing whether destination points to allocated space. If space hasn't been allocated, the function overwrites strlen(source) bytes of memory, starting at destination; this can cause unpredictable problems. The use of strcpy() is illustrated in Listing 17.2.

NOTE: When a program uses malloc() to allocate memory, as Listing 17.2 does, good programming practice requires the use of the free() function to free up the memory when the program is finished with it. You'll learn about free() on Day 20, "Working with Memory."

Listing 17.2. Before using strcpy(), you must allocate storage space for the destination string.

```
1:  /* Demonstrates strcpy(). */
2:  #include <stdlib.h>
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char source[] = "The source string.";
7:
8:  main()
9:  {
10:     char dest1[80];
11:     char *dest2, *dest3;
12:
13:     printf("\nsource: %s", source );
14:
15:     /* Copy to dest1 is okay because dest1 points to */
16:     /* 80 bytes of allocated space. */
17:
```

```
18:     strcpy(dest1, source);
19:     printf("\ndest1:  %s", dest1);
20:
21:     /* To copy to dest2 you must allocate space. */
22:
23:     dest2 = (char *)malloc(strlen(source) +1);
24:     strcpy(dest2, source);
25:     printf("\ndest2:  %s\n", dest2);
26:
27:     /* Copying without allocating destination space is a no-no. */
28:     /* The following could cause serious problems. */
29:
30:     /* strcpy(dest3, source); */
31:     return(0);
32: }
source: The source string.
dest1:  The source string.
dest2:  The source string.
```

ANALYSIS: This program demonstrates copying strings both to character arrays such as dest1 (declared on line 10) and to character pointers such as dest2 (declared along with dest3 on line 11). Line 13 prints the original source string. This string is then copied to dest1 with strcpy() on line 18. Line 24 copies source to dest2. Both dest1 and dest2 are printed to show that the function was successful. Notice that line 23 allocates the appropriate amount of space for dest2 with the malloc() function. If you copy a string to a character pointer that hasn't been allocated memory, you get unpredictable results.

17-2-2. The strncpy() Function

The strncpy() function is similar to strcpy(), except that strncpy() lets you specify how many characters to copy. Its prototype is

```
char *strncpy(char *destination, char *source, size_t n);
```

The arguments destination and source are pointers to the destination and source strings. The function copies, at most, the first n characters of source to destination. If source is shorter than n characters, enough null characters are added to the end of source to make a total of n characters copied to destination. If source is longer than n characters, no terminating \0 is added to destination. The function's return value is destination.

Listing 17.3 demonstrates the use of strncpy().

Listing 17.3. The strncpy() function.

```
1:  /* Using the strncpy() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char dest[] = ".........................";
7:  char source[] = "abcdefghijklmnopqrstuvwxyz";
8:
9:  main()
10: {
11:     size_t n;
```

```
12:
13:     while (1)
14:     {
15:         puts("Enter the number of characters to copy (1-26)");
16:         scanf("%d", &n);
17:
18:         if (n > 0 && n< 27)
19:             break;
20:     }
21:
22:     printf("\nBefore strncpy destination = %s", dest);
23:
24:     strncpy(dest, source, n);
25:
26:     printf("\nAfter strncpy destination = %s\n", dest);
27:     return(0);
28: }
Enter the number of characters to copy (1-26)
15
Before strncpy destination = ........................
After strncpy destination = abcdefghijklmno...........
```

ANALYSIS: In addition to demonstrating the strncpy() function, this program also illustrates an effective way to ensure that only correct information is entered by the user. Lines 13 through 20 contain a while loop that prompts the user for a number from 1 to 26. The loop continues until a valid value is entered, so the program can't continue until the user enters a valid value. When a number between 1 and 26 is entered, line 22 prints the original value of dest, line 24 copies the number of characters specified by the user from source to dest, and line 26 prints the final value of dest.

WARNING: Be sure that the number of characters copied doesn't exceed the allocated size of the destination.

17-2-3. The strdup() Function

The library function strdup() is similar to strcpy(), except that strdup() performs its own memory allocation for the destination string with a call to malloc(). In effect, it does what you did in Listing 17.2, allocating space with malloc() and then calling strcpy(). The prototype for strdup() is

```
char *strdup( char *source );
```

The argument source is a pointer to the source string. The function returns a pointer to the destination string--the space allocated by malloc()--or NULL if the needed memory couldn't be allocated. Listing 17.4 demonstrates the use of strdup(). Note that strdup() isn't an ANSI-standard function. It is included in the Microsoft, Borland, and Symantec C libraries, but it might not be present (or it might be different) in other C compilers.

Listing 17.4. Using strdup() to copy a string with automatic memory allocation.

```
1:  /* The strdup() function. */
2:  #include <stdlib.h>
```

```
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char source[] = "The source string.";
7:
8:  main()
9:  {
10:     char *dest;
11:
12:     if ( (dest = strdup(source)) == NULL)
13:     {
14:         fprintf(stderr, "Error allocating memory.");
15:         exit(1);
16:     }
17:
18:     printf("The destination = %s\n", dest);
19:     return(0);
20: }
The destination = The source string.
```

ANALYSIS: In this listing, strdup() allocates the appropriate memory for dest. It then makes a copy of the passed string, source. Line 18 prints the duplicated string.

17-3. Concatenating Strings

If you're not familiar with the term *concatenation,* you might be asking, "What is it?" and "Is it legal?" Well, it means to join two strings--to tack one string onto the end of another--and, in most states, it is legal. The C standard library contains two string concatenation functions--strcat() and strncat()--both of which require the header file STRING.H.

17-3-1. The strcat() Function

The prototype of strcat() is

```
char *strcat(char *str1, char *str2);
```

The function appends a copy of str2 onto the end of str1, moving the terminating null character to the end of the new string. You must allocate enough space for str1 to hold the resulting string. The return value of strcat() is a pointer to str1. Listing 17.5 demonstrates strcat().

Listing 17.5. Using strcat() to concatenate strings.

```
1:  /* The strcat() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char str1[27] = "a";
7:  char str2[2];
8:
9:  main()
10: {
11:     int n;
```

```
12:
13:     /* Put a null character at the end of str2[]. */
14:
15:     str2[1] = `\0';
16:
17:     for (n = 98; n< 123; n++)
18:     {
19:         str2[0] = n;
20:         strcat(str1, str2);
21:         puts(str1);
22:     }
23:      return(0);
24: }
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

ANALYSIS: The ASCII codes for the letters b through z are 98 to 122. This program uses these ASCII codes in its demonstration of strcat(). The for loop on lines 17 through 22 assigns these values in turn to str2[0]. Because str2[1] is already the null character (line 15), the effect is to assign the strings "b", "c", and so on to str2. Each of these strings is concatenated with str1 (line 20), and then str1 is displayed on-screen (line 21).

17-3-2. The strncat() Function

The library function strncat() also performs string concatenation, but it lets you specify how many characters of the source string are appended to the end of the destination string. The prototype is

```
char *strncat(char *str1, char *str2, size_t n);
```

If str2 contains more than n characters, the first n characters are appended to the end of str1. If str2 contains fewer than n characters, all of str2 is appended to the end of str1. In either case, a terminating null character is added at

the end of the resulting string. You must allocate enough space for str1 to hold the resulting string. The function returns a pointer to str1. Listing 17.6 uses strncat() to produce the same output as Listing 17.5.

Listing 17.6. Using the strncat() function to concatenate strings.

```
1:  /* The strncat() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  char str2[] = "abcdefghijklmnopqrstuvwxyz";
7:
8:  main()
9:  {
10:     char str1[27];
11:     int n;
12:
13:     for (n=1; n< 27; n++)
14:     {
15:         strcpy(str1, "");
16:         strncat(str1, str2, n);
17:         puts(str1);
18:     }
19: }
a
ab
abc
abcd
abcde
abcdef
abcdefg
abcdefgh
abcdefghi
abcdefghij
abcdefghijk
abcdefghijkl
abcdefghijklm
abcdefghijklmn
abcdefghijklmno
abcdefghijklmnop
abcdefghijklmnopq
abcdefghijklmnopqr
abcdefghijklmnopqrs
abcdefghijklmnopqrst
abcdefghijklmnopqrstu
abcdefghijklmnopqrstuv
abcdefghijklmnopqrstuvw
abcdefghijklmnopqrstuvwx
abcdefghijklmnopqrstuvwxy
abcdefghijklmnopqrstuvwxyz
```

ANALYSIS: You might wonder about the purpose of line 15, strcpy(str1, "");. This line copies to str1 an empty string consisting of only a single null character. The result is that the first character in str1--str1[0]--is set equal to 0 (the null character). The same thing could have been accomplished with the statements str1[0] = 0; or str1[0] = `\0';.

17-4. Comparing Strings

Strings are compared to determine whether they are equal or unequal. If they are unequal, one string is "greater than" or "less than" the other. Determinations of "greater" and "less" are made with the ASCII codes of the characters. In the case of letters, this is equivalent to alphabetical order, with the one seemingly strange exception that all uppercase letters are "less than" the lowercase letters. This is true because the uppercase letters have ASCII codes 65 through 90 for A through Z, while lowercase a through z are represented by 97 through 122. Thus, "ZEBRA" would be considered to be less than "apple" by these C functions.

The ANSI C library contains functions for two types of string comparisons: comparing two entire strings, and comparing a certain number of characters in two strings.

17-4-1. Comparing Two Entire Strings

The function strcmp() compares two strings character by character. Its prototype is

```
int strcmp(char *str1, char *str2);
```

The arguments str1 and str2 are pointers to the strings being compared. The function's return values are given in Table 17.1. Listing 17.7 demonstrates strcmp().

Table 17.1. The values returned by strcmp().

Return Value	Meaning
< 0	str1 is less than str2.
0	str1 is equal to str2.
> 0	str1 is greater than str2.

Listing 17.7. Using strcmp() to compare strings.

```
1:  /* The strcmp() function. */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char str1[80], str2[80];
9:      int x;
10:
11:     while (1)
12:     {
13:
14:         /* Input two strings. */
15:
16:         printf("\n\nInput the first string, a blank to exit: ");
17:         gets(str1);
```

```
18:
19:         if ( strlen(str1) == 0 )
20:             break;
21:
22:         printf("\nInput the second string: ");
23:         gets(str2);
24:
25:         /* Compare them and display the result. */
26:
27:         x = strcmp(str1, str2);
28:
29:         printf("\nstrcmp(%s,%s) returns %d", str1, str2, x);
30:     }
31:     return(0);
32: }
Input the first string, a blank to exit: First string
Input the second string: Second string
strcmp(First string,Second string) returns -1
Input the first string, a blank to exit: test string
Input the second string: test string
strcmp(test string,test string) returns 0
Input the first string, a blank to exit: zebra
Input the second string: aardvark
strcmp(zebra,aardvark) returns 1
Input the first string, a blank to exit:
```

NOTE: On some UNIX systems, string comparison functions don't necessarily return -1 when the strings aren't the same. They will, however, always return a nonzero value for unequal strings.

ANALYSIS: This program demonstrates strcmp(), prompting the user for two strings (lines 16, 17, 22, and 23) and displaying the result returned by strcmp() on line 29. Experiment with this program to get a feel for how strcmp() compares strings. Try entering two strings that are identical except for case, such as Smith and SMITH. You'll see that strcmp() is case-sensitive, meaning that the program considers uppercase and lowercase letters to be different.

17-4-2. Comparing Partial Strings

The library function strncmp() compares a specified number of characters of one string to another string. Its prototype is

```
int strncmp(char *str1, char *str2, size_t n);
```

The function strncmp() compares n characters of str2 to str1. The comparison proceeds until n characters have been compared or the end of str1 has been reached. The method of comparison and return values are the same as for strcmp(). The comparison is case-sensitive. Listing 17.8 demonstrates strncmp().

Listing 17.8. Comparing parts of strings with strncmp().

```
1:  /* The strncmp() function. */
2:
3:  #include <stdio.h>
4:  #include[Sigma]>=tring.h>
5:
6:  char str1[] = "The first string.";
7:  char str2[] = "The second string.";
8:
9:  main()
10: {
11:     size_t n, x;
12:
13:     puts(str1);
14:     puts(str2);
15:
16:     while (1)
17:     {
18:         puts("\n\nEnter number of characters to compare, 0 to
exit.");
19:         scanf("%d", &n);
20:
21:         if (n <= 0)
22:             break;
23:
24:         x = strncmp(str1, str2, n);
25:
26:         printf("\nComparing %d characters, strncmp() returns %d.",
n, x);
27:     }
28:     return(0);
29: }
The first string.
The second string.
Enter number of characters to compare, 0 to exit.
3
Comparing 3 characters, strncmp() returns .©]
Enter number of characters to compare, 0 to exit.
6
Comparing 6 characters, strncmp() returns -1.
Enter number of characters to compare, 0 to exit.
0
```

ANALYSIS: This program compares two strings defined on lines 6 and 7. Lines 13 and 14 print the strings to the screen so that the user can see what they are. The program executes a while loop on lines 16 through 27 so that multiple comparisons can be done. If the user asks to compare zero characters on lines 18 and 19, the program breaks on line 22; otherwise, a strncmp() executes on line 24, and the result is printed on line 26.

17-4-3. Comparing Two Strings While Ignoring Case

Unfortunately, the ANSI C library doesn't include any functions for case-insensitive string comparison. Fortunately, most C compilers provide their own "in-house" functions for this task. Symantec uses the function strcmpl(). Microsoft uses a function called _stricmp(). Borland has two functions--strcmpi() and stricmp(). You need to check your library reference manual to determine which function is appropriate for your compiler. When you use a function

that isn't case-sensitive, the strings Smith and SMITH compare as equal. Modify line 27 in Listing 17.7 to use the appropriate case-insensitive compare function for your compiler, and try the program again.

17-5. Searching Strings

The C library contains a number of functions that search strings. To put it another way, these functions determine whether one string occurs within another string and, if so, where. You can choose from six string-searching functions, all of which require the header file STRING.H.

17-5-1. The strchr() Function

The strchr() function finds the first occurrence of a specified character in a string. The prototype is

```
char *strchr(char *str, int ch);
```

The function strchr() searches str from left to right until the character ch is found or the terminating null character is found. If ch is found, a pointer to it is returned. If not, NULL is returned.

When strchr() finds the character, it returns a pointer to that character. Knowing that str is a pointer to the first character in the string, you can obtain the position of the found character by subtracting str from the pointer value returned by strchr(). Listing 17.9 illustrates this. Remember that the first character in a string is at position 0. Like many of C's string functions, strchr() is case-sensitive. For example, it would report that the character F isn't found in the string raffle.

Listing 17.9. Using strchr() to search a string for a single character.

```
1:  /* Searching for a single character with strchr(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char *loc, buf[80];
9:      int ch;
10:
11:     /* Input the string and the character. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf);
15:     printf("Enter the character to search for: ");
16:     ch = getchar();
17:
18:     /* Perform the search. */
19:
20:     loc = strchr(buf, ch);
21:
22:     if ( loc == NULL )
23:         printf("The character %c was not found.", ch);
24:     else
25:         printf("The character %c was found at position %d.\n",
26:                 ch, loc-buf);
27:     return(0);
```

```
28: }
Enter the string to be searched: How now Brown Cow?
Enter the character to search for: C
The character C was found at position 14.
```

ANALYSIS: This program uses strchr() on line 20 to search for a character within a string. strchr() returns a pointer to the location where the character is first found, or NULL if the character isn't found. Line 22 checks whether the value of loc is NULL and prints an appropriate message. As just mentioned, the position of the character within the string is determined by subtracting the string pointer from the value returned by the function.

17-5-2. The strrchr() Function

The library function strrchr() is identical to strchr(), except that it searches a string for the last occurrence of a specified character in a string. Its prototype is

```
char *strrchr(char *str, int ch);
```

The function strrchr() returns a pointer to the last occurrence of ch in str and NULL if it finds no match. To see how this function works, modify line 20 in Listing 17.9 to use strrchr() instead of strchr().

17-5-3. The strcspn() Function

The library function strcspn() searches one string for the first occurrence of any of the characters in a second string. Its prototype is

```
size_t strcspn(char *str1, char *str2);
```

The function strcspn() starts searching at the first character of str1, looking for any of the individual characters contained in str2. This is important to remember. The function doesn't look for the string str2, but only the characters it contains. If the function finds a match, it returns the offset from the beginning of str1, where the matching character is located. If it finds no match, strcspn() returns the value of strlen(str1). This indicates that the first match was the null character terminating the string. Listing 17.10 shows you how to use strcspn().

Listing 17.10. Searching for a set of characters with strcspn().

```
1:  /* Searching with strcspn(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char  buf1[80], buf2[80];
9:      size_t loc;
10:
11:     /* Input the strings. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf1);
15:     printf("Enter the string containing target characters: ");
16:     gets(buf2);
17:
18:     /* Perform the search. */
```

```
19:
20:     loc = strcspn(buf1, buf2);
21:
22:     if ( loc ==  strlen(buf1) )
23:         printf("No match was found.");
24:     else
25:         printf("The first match was found at position %d.\n",
loc);
26:     return(0);
27: }
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: Cat
The first match was found at position 14.
```

ANALYSIS: This listing is similar to Listing 17.10. Instead of searching for the first occurrence of a single character, it searches for the first occurrence of any of the characters entered in the second string. The program calls strcspn() on line 20 with buf1 and buf2. If any of the characters in buf2 are in buf1, strcspn() returns the offset from the beginning of buf1 to the location of the first occurrence. Line 22 checks the return value to determine whether it is NULL. If the value is NULL, no characters were found, and an appropriate message is displayed on line 23. If a value was found, a message is displayed stating the character's position in the string.

17-5-4. The strspn() Function

This function is related to the previous one, strcspn(), as the following paragraph explains. Its prototype is

```
size_t strspn(char *str1, char *str2);
```

The function strspn() searches str1, comparing it character by character with the characters contained in str2. It returns the position of the first character in str1 that doesn't match a character in str2. In other words, strspn() returns the length of the initial segment of str1 that consists entirely of characters found in str2. The return is 0 if no characters match. Listing 17.11 demonstrates strspn().

Listing 17.11. Searching for the first nonmatching character with strspn().

```
1: /* Searching with strspn(). */
2:
3: #include <stdio.h>
4: #include <string.h>
5:
6: main()
7: {
8:     char  buf1[80], buf2[80];
9:     size_t loc;
10:
11:     /* Input the strings. */
12:
13:     printf("Enter the string to be searched: ");
14:     gets(buf1);
15:     printf("Enter the string containing target characters: ");
16:     gets(buf2);
17:
18:     /* Perform the search. */
19:
```

```
20:     loc = strspn(buf1, buf2);
21:
22:     if ( loc ==  0 )
23:         printf("No match was found.\n");
24:     else
25:         printf("Characters match up to position %d.\n", loc-1);
26:
27: }
Enter the string to be searched: How now Brown Cow?
Enter the string containing target characters: How now what?
Characters match up to position 7.
```

ANALYSIS: This program is identical to the previous example, except that it calls strspn() instead of strcspn() on line 20. The function returns the offset into buf1, where the first character not in buf2 is found. Lines 22 through 25 evaluate the return value and print an appropriate message.

17-5-5. The strpbrk() Function

The library function strpbrk() is similar to strcspn(), searching one string for the first occurrence of any character contained in another string. It differs in that it doesn't include the terminating null characters in the search. The function prototype is

```
char *strpbrk(char *str1, char *str2);
```

The function strpbrk() returns a pointer to the first character in str1 that matches any of the characters in str2. If it doesn't find a match, the function returns NULL. As previously explained for the function strchr(), you can obtain the offset of the first match in str1 by subtracting the pointer str1 from the pointer returned by strpbrk() (if it isn't NULL, of course). For example, replace strcspn() on line 20 of Listing 17.10 with strpbrk().

17-5-6. The strstr() Function

The final, and perhaps most useful, C string-searching function is strstr(). This function searches for the first occurrence of one string within another, and it searches for the entire string, not for individual characters within the string. Its prototype is

```
char *strstr(char *str1, char *str2);
```

The function strstr() returns a pointer to the first occurrence of str2 within str1. If it finds no match, the function returns NULL. If the length of str2 is 0, the function returns str1. When strstr() finds a match, you can obtain the offset of str2 within str1 by pointer subtraction, as explained earlier for strchr(). The matching procedure that strstr() uses is case-sensitive. Listing 17.12 demonstrates how to use strstr().

Listing 17.12. Using strstr() to search for one string within another.

```
1:  /* Searching with strstr(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char *loc, buf1[80], buf2[80];
9:
10:     /* Input the strings. */
```

```
11:
12:     printf("Enter the string to be searched: ");
13:     gets(buf1);
14:     printf("Enter the target string: ");
15:     gets(buf2);
16:
17:     /* Perform the search. */
18:
19:     loc = strstr(buf1, buf2);
20:
21:     if ( loc ==  NULL )
22:         printf("No match was found.\n");
23:     else
24:         printf("%s was found at position %d.\n", buf2, loc-buf1);
25:     return(0);
26: }
Enter the string to be searched: How now brown cow?
Enter the target string: cow
Cow was found at position 14.
```

ANALYSIS: This function provides an alternative way to search a string. This time you can search for an entire string within another string. Lines 12 through 15 prompt for two strings. Line 19 uses strstr() to search for the second string, buf2, within the first string, buf1. A pointer to the first occurrence is returned, or NULL is returned if the string isn't found. Lines 21 through 24 evaluate the returned value, loc, and print an appropriate message.

DO remember that for many of the string functions, there are equivalent functions that let you specify a number of characters to manipulate. The functions that allow specification of the number of characters are usually named strn*xxx*(), where *xxx* is specific to the function.

DON'T forget that C is case-sensitive. A and a are different.

17-6. String Conversions

Many C libraries contain two functions that change the case of characters within a string. These functions aren't ANSI standard and therefore might differ or not even exist in your compiler. Because they can be quite useful, they are included here. Their prototypes, in STRING.H, are as follows for the Microsoft C compiler (if you use a different compiler, they should be similar):

```
char *strlwr(char *str);
char *strupr(char *str);
```

The function strlwr() converts all the letter characters in str from uppercase to lowercase; strupr() does the reverse, converting all the characters in str to uppercase. Nonletter characters aren't affected. Both functions return str. Note that neither function actually creates a new string but modifies the existing string in place. Listing 17.13 demonstrates these functions. Remember that to compile a program that uses non-ANSI functions, you might need to tell your compiler not to enforce the ANSI standards.

Listing 17.13. Converting the case of characters in a string with strlwr() and strupr().

```
1:  /* The character conversion functions strlwr() and strupr(). */
2:
3:  #include <stdio.h>
4:  #include <string.h>
5:
6:  main()
7:  {
8:      char buf[80];
9:
10:     while (1)
11:     {
12:         puts("Enter a line of text, a blank to exit.");
13:         gets(buf);
14:
15:         if ( strlen(buf) == 0 )
16:             break;
17:
18:         puts(strlwr(buf));
19:         puts(strupr(buf));
20:     }
21:     return(0);
22: }
Enter a line of text, a blank to exit.
Bradley L. Jones
bradley l. jones
BRADLEY L. JONES
Enter a line of text, a blank to exit.
```

ANALYSIS: This listing prompts for a string on line 12. It then checks to ensure that the string isn't blank (line 15). Line 18 prints the string after converting it to lowercase. Line 19 prints the string in all uppercase.

These functions are a part of the Symantec, Microsoft, and Borland C libraries. You need to check the Library Reference for your compiler before using these functions. If portability is a concern, you should avoid using non-ANSI functions such as these.

17-7. Miscellaneous String Functions

This section covers a few string functions that don't fall into any other category. They all require the header file STRING.H.

17-7-1. The strrev() Function

The function strrev() reverses the order of all the characters in a string. Its prototype is

```
char *strrev(char *str);
```

The order of all characters in str is reversed, with the terminating null character remaining at the end. The function returns str. After strset() and strnset() are defined in the next section, strrev() is demonstrated in Listing 17.14.

17-7-2. The strset() and strnset() Functions

Like the previous function, strrev(), strset() and strnset() aren't part of the ANSI C standard library. These functions change all characters (strset()) or a specified number of characters (strnset()) in a string to a specified character. The prototypes are

```
char *strset(char *str, int ch);
char *strnset(char *str, int ch, size_t n);
```

The function strset() changes all the characters in str to ch except the terminating null character. The function strnset() changes the first n characters of str to ch. If n >= strlen(str), strnset() changes all the characters in str. Listing 17.14 demonstrates all three functions.

Listing 17.14. A demonstration of strrev(), strnset(), and strset().

```
1:  /* Demonstrates strrev(), strset(), and strnset(). */
2:  #include <stdio.h>
3:  #include <string.h>
4:
5:  char str[] = "This is the test string.";
6:
7:  main()
8:  {
9:      printf("\nThe original string: %s", str);
10:     printf("\nCalling strrev(): %s", strrev(str));
11:     printf("\nCalling strrev() again: %s", strrev(str));
12:     printf("\nCalling strnset(): %s", strnset(str, `!', 5));
13:     printf("\nCalling strset(): %s", strset(str, `!'));
14:     return(0);
15: }
The original string: This is the test string.
Calling strrev(): .gnirts tset eht si sihT
Calling strrev() again: This is the test string.
Calling strnset(): !!!!!is the test string.
Calling strset(): !!!!!!!!!!!!!!!!!!!!!!
```

ANALYSIS: This program demonstrates the three different string functions. The demonstrations are done by printing the value of a string, str. Line 9 prints the string normally. Line 10 prints the string after it has been reversed with strrev(). Line 11 reverses it back to its original state. Line 12 uses the strnset() function to set the first five characters of str to exclamation marks. To finish the program, line 13 changes the entire string to exclamation marks.

Although these functions aren't ANSI standard, they are included in the Symantec, Microsoft, and Borland C compiler function libraries. You should check your compiler's Library Reference manual to determine whether your compiler supports these functions.

17-8. String-to-Number Conversions

Sometimes you will need to convert the string representation of a number to an actual numeric variable. For example, the string "123" can be converted to a type int variable with the value 123. Three functions can be used to convert a string to a number. They are explained in the following sections; their prototypes are in STDLIB.H.

17-8-1. The atoi() Function

The library function atoi() converts a string to an integer. The prototype is

```
int atoi(char *ptr);
```

The function atoi() converts the string pointed to by ptr to an integer. Besides digits, the string can contain leading white space and a + or -- sign. Conversion starts at the beginning of the string and proceeds until an unconvertible character (for example, a letter or punctuation mark) is encountered. The resulting integer is returned to the calling program. If it finds no convertible characters, atoi() returns 0. Table 17.2 lists some examples.

Table 17.2. String-to-number conversions with atoi().

String	**Value Returned by** atoi()
"157"	157
"-1.6"	-1
"+50x"	50
"twelve"	0
"x506"	0

The first example is straightforward. In the second example, you might be confused about why the ".6" didn't translate. Remember that this is a string-to-integer conversion. The third example is also straightforward; the function understands the plus sign and considers it a part of the number. The fourth example uses "twelve". The atoi() function can't translate words; it sees only characters. Because the string didn't start with a number, atoi() returns 0. This is true of the last example also.

17-8-2. The atol() Function

The library function atol() works exactly like atoi(), except that it returns a type long. The function prototype is

```
long atol(char *ptr);
```

The values returned by atol() would be the same as shown for atoi() in Table 17.2, except that each return value would be a type long instead of a type int.

17-8-3. The atof() Function

The function atof() converts a string to a type double. The prototype is

```
double atof(char *str);
```

The argument str points to the string to be converted. This string can contain leading white space and a + or -- character. The number can contain the digits 0 through 9, the decimal point, and the exponent indicator E or e. If there are no convertible characters, atof() returns 0. Table 17.3 lists some examples of using atof().

Table 17.3. String-to-number conversions with atof().

String	**Value Returned by** atof()
"12"	12.000000
"-0.123"	-0.123000
"123E+3"	123000.000000
"123.1e-5"	0.001231

Listing 17.15 lets you enter your own strings for conversion.

Listing 17.15. Using atof() to convert strings to type double numeric variables.

```
1:  /* Demonstration of atof(). */
2:
3:  #include <string.h>
4:  #include <stdio.h>
5:  #include <stdlib.h>
6:
7:  main()
8:  {
9:      char buf[80];
10:     double d;
11:
12:     while (1)
13:     {
14:         printf("\nEnter the string to convert (blank to exit):
");
15:         gets(buf);
16:
17:         if ( strlen(buf) == 0 )
18:             break;
19:
20:         d = atof( buf );
21:
22:         printf("The converted value is %f.", d);
23:     }
24:     return(0);
25: }
Enter the string to convert (blank to exit):     1009.12
The converted value is 1009.120000.
Enter the string to convert (blank to exit):     abc
The converted value is 0.000000.
Enter the string to convert (blank to exit):     3
The converted value is 3.000000.
```

Enter the string to convert (blank to exit):

ANALYSSIS: The while loop on lines 12 through 23 lets you keep running the program until you enter a blank line. Lines 14 and 15 prompt for the value. Line 17 checks whether a blank line was entered. If it was, the program breaks out of the while loop and ends. Line 20 calls atof(), converting the value entered (buf) to a type double, d. Line 22 prints the final result.

17-9. Character Test Functions

The header file CTYPE.H contains the prototypes for a number of functions that test characters, returning TRUE or FALSE depending on whether the character meets a certain condition. For example, is it a letter or is it a numeral? The is*xxxx*() functions are actually macros, defined in CTYPE.H. You'll learn about macros on Day 21, "Advanced Compiler Use," at which time you might want to look at the definitions in CTYPE.H to see how they work. For now, you only need to see how they're used.

The is*xxxx*() macros all have the same prototype:

```
int isxxxx(int ch);
```

In the preceding line, ch is the character being tested. The return value is TRUE (nonzero) if the condition is met or FALSE (zero) if it isn't. Table 17.4 lists the complete set of is*xxxx*() macros.

Table 17.4. The isxxxx() macros.

Macro	Action
isalnum()	Returns TRUE if ch is a letter or a digit.
isalpha()	Returns TRUE if ch is a letter.
isascii()	Returns TRUE if ch is a standard ASCII character (between 0 and 127).
iscntrl()	Returns TRUE if ch is a control character.
isdigit()	Returns TRUE if ch is a digit.
isgraph()	Returns TRUE if ch is a printing character (other than a space).
islower()	Returns TRUE if ch is a lowercase letter.
isprint()	Returns TRUE if ch is a printing character (including a space).
ispunct()	Returns TRUE if ch is a punctuation character.
isspace()	Returns TRUE if ch is a whitespace character (space, tab, vertical tab, line feed, form feed, or carriage return).
isupper()	Returns TRUE if ch is an uppercase letter.
isxdigit()	Returns TRUE if ch is a hexadecimal digit (0 through 9, a through f, A through F).

You can do many interesting things with the character-test macros. One example is the function get_int(), shown in Listing 17.16. This function inputs an integer from stdin and returns it as a type int variable. The function skips over leading white space and returns 0 if the first nonspace character isn't a numeric character.

Listing 17.16. Using the isxxxx() macros to implement a function that inputs an integer.

```
1:  /* Using character test macros to create an integer */
2:  /* input function. */
3:
4:  #include <stdio.h>
5:  #include <ctype.h>
6:
7:  int get_int(void);
8:
9:  main()
10: {
11:     int x;
12:     x =  get_int();
13:
14:     printf("You entered %d.\n", x);
15: }
16:
17: int get_int(void)
18: {
19:     int ch, i, sign = 1;
```

```
20:
21:     /* Skip over any leading white space. */
22:
23:     while ( isspace(ch = getchar()) )
24:         ;
25:
26:     /* If the first character is nonnumeric, unget */
27:     /* the character and return 0. */
28:
29:     if (ch != `-' && ch != `+' && !isdigit(ch) && ch != EOF)
30:     {
31:         ungetc(ch, stdin);
32:         return 0;
33:     }
34:
35:     /* If the first character is a minus sign, set */
36:     /* sign accordingly. */
37:
38:     if (ch == `-')
39:         sign = -1;
40:
41:     /* If the first character was a plus or minus sign, */
42:     /* get the next character. */
43:
44:     if (ch == `+' || ch == `-')
45:         ch = getchar();
46:
47:     /* Read characters until a nondigit is input. Assign */
48:     /* values, multiplied by proper power of 10, to i. */
49:
50:     for (i = 0; isdigit(ch); ch = getchar() )
51:         i = 10 * i + (ch - `0');
52:
53:     /* Make result negative if sign is negative. */
54:
55:     i *= sign;
56:
57:     /* If EOF was not encountered, a nondigit character */
58:     /* must have been read in, so unget it. */
59:
60:     if (ch != EOF)
61:         ungetc(ch, stdin);
62:
63:     /* Return the input value. */
64:
65:     return i;
66: }
-100
You entered -100.
abc3.145
You entered 0.
```

```
9 9 9
You entered 9.
2.5
You entered 2.
```

ANALYSIS: This program uses the library function ungetc() on lines 31 and 61, which you learned about on Day 14, "Working with the Screen, Printer, and Keyboard." Remember that this function "ungets," or returns, a character to the specified stream. This returned character is the first one input the next time the program reads a character from that stream. This is necessary because when the function get_int() reads a nonnumeric character from stdin, you want to put that character back in case the program needs to read it later.

In this program, main() is simple. An integer variable, x, is declared (line 11), assigned the value of the get_int() function (line 12), and printed to the screen (line 14). The get_int() function makes up the rest of the program.

The get_int() function isn't so simple. To remove leading white space that might be entered, line 23 loops with a while command. The isspace() macro tests a character, ch, obtained with the getchar() function. If ch is a space, another character is retrieved, until a nonwhitespace character is received. Line 29 checks whether the character is one that can be used. Line 29 could be read, "If the character input isn't a negative sign, a plus sign, a digit, or the end of the file(s)." If this is true, ungetc() is used on line 31 to put the character back, and the function returns to main(). If the character is usable, execution continues.

Lines 38 through 45 handle the sign of the number. Line 38 checks to see whether the character entered was a negative sign. If it was, a variable (sign) is set to -1. sign is used to make the final number either positive or negative (line 55). Because positive numbers are the default, after you have taken care of the negative sign, you are almost ready to continue. If a sign was entered, the program needs to get another character. Lines 44 and 45 take care of this.

The heart of the function is the for loop on lines 50 and 51, which continues to get characters as long as the characters retrieved are digits. Line 51 might be a little confusing at first. This line takes the individual character entered and turns it into a number. Subtracting the character `0' from your number changes a character number to a real number. (Remember the ASCII values.) When the correct numerical value is obtained, the numbers are multiplied by the proper power of 10. The for loop continues until a nondigit number is entered. At that point, line 55 applies the sign to the number, making it complete.

Before returning, the program needs to do a little cleanup. If the last number wasn't the end of file, it needs to be put back in case it's needed elsewhere. Line 61 does this before line 65 returns.

DON'T use non-ANSI functions if you plan to port your application to other platforms.

DO take advantage of the string functions that are available.

DON'T confuse characters with numbers. It's easy to forget that the character "1" isn't the same thing as the number 1.

17-10. Summary

This chapter showed various ways you can manipulate strings. Using C standard library functions (and possibly compiler-specific functions as well), you can copy, concatenate, compare, and search strings. These are all necessary tasks in most programming projects. The standard library also contains functions for converting the case of characters in strings and for converting strings to numbers. Finally, C provides a variety of character-test functions or, more accurately, macros that perform a variety of tests on individual characters. By using these macros to test characters, you can create your own custom input functions.

Q&A

Q How do I know whether a function is ANSI-compatible?
A Most compilers have a Library Function Reference manual or section. This manual or section of a manual lists all the compiler's library functions and how to use them. Usually the manual includes information on the compatibility of the function. Sometimes the descriptions state not only whether the function is ANSI-compatible, but also whether it is compatible with DOS, UNIX, Windows, C++, or OS/2. (Most compilers tell you only what is relevant to their compiler.)
Q Are all of the available string functions presented in this chapter?
A No. However, the string functions presented in this chapter should cover virtually all your needs. Consult your compiler's Library Reference to see what other functions are available.
Q Does strcat() ignore trailing spaces when doing a concatenation?
A No. strcat() looks at a space as just another character.
Q Can I convert numbers to strings?
A Yes. You can write a function similar to the one in Listing 17.16, or you can check your Library Reference for available functions. Some available functions include itoa(), ltoa(), and ultoa(). sprint(f) can also be used.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned.

Quiz

1. What is the length of a string, and how can it be determined?
2. Before copying a string, what must you be sure to do?
3. What does the term *concatenate* mean?
4. When comparing strings, what is meant by "One string is greater than another string?"
5. What is the difference between strcmp() and strncmp()?
6. What is the difference between strcmp() and strcmpi()?
7. What values does isascii() test for?
8. Using Table 17.4, which macros would return TRUE for var?

```
int var = 1;
```

9. Using Table 17.4, which macros would return TRUE for x?

```
char x = 65;
```

10. What are the character-test functions used for?

Exercises

1. What values do the test functions return?
2. What would the atoi() function return if passed the following values?
a. "65"
b. "81.23"
c. "-34.2"
d. "ten"
e. "+12hundred"
f. "negative100"
3. What would the atof() function return if passed the following?
a. "65"
b. "81.23"
c. "-34.2"
d. "ten"
e. "+12hundred"
f. "1e+3"
4. BUG BUSTER: Is anything wrong with the following?

```
char *string1, string2;
```

```
string1 = "Hello World";
strcpy( string2, string1);
printf( "%s %s", string1, string2 );
```

Because of the many possible solutions, answers aren't provided for the following exercises.

5. Write a program that prompts for the user's last name, first name, and middle name individually. Then store the name in a new string as first initial, period, space, middle initial, period, space, last name. For example, if Bradley, Lee, and Jones are entered, store B. L. Jones. Display the new name to the screen.

6. Write a program to prove your answers to quiz questions 8 and 9.

7. The function strstr() finds the first occurrence of one string within another, and it is case-sensitive. Write a function that performs the same task without case-sensitivity.

8. Write a function that determines the number of times one string occurs within another.

9. Write a program that searches a text file for occurrences of a user-specified target string and then reports the line numbers where the target is found. For example, if you search one of your C source code files for the string "printf(", the program should list all the lines where the printf() function is called by the program.

10. Listing 17.16 demonstrates a function that inputs an integer from stdin. Write a function get_float() that inputs a floating-point value from stdin.