Outline

8.1 Introduction

This chapter introduces the C standard library functions that facilitate string and character processing. The functions enable programs to process characters, strings, lines of text and blocks of memory. The chapter discusses the techniques used to develop editors, word processors, page-layout software, computerized typesetting systems and other kinds of text-processing software. The text manipulations performed by formatted input/output functions like `printf` and `scanf` can be implemented using the functions discussed in this chapter.

8.2 Fundamentals of Strings and Characters

Characters are the fundamental building blocks of source programs. Every program is composed of a sequence of characters that—when grouped together meaningfully—is interpreted by the computer as a series of instructions used to accomplish a task. A program may contain character constants. A character constant is an `int` value represented as a character in single quotes. The value of a character constant is the integer value of the character in the machine's character set. For example, `'z'` represents the integer value of `z`, and `'\n'` the integer value of newline (122 and 10 in ASCII, respectively).

A **string** is a series of characters treated as a single unit. A string may include letters, digits and various **special characters** such as `+`, `-`, `*`, `/` and `$`. **String literals**, or **string constants**, in C are written in double quotation marks as follows:

`"John Q. Doe"`	(a name)
`"99999 Main Street"`	(a street address)
`"Waltham, Massachusetts"`	(a city and state)
`"(201) 555-1212"`	(a telephone number)

A string in C is an array of characters ending in the **null character (`'\0'`)**. A string is accessed via a *pointer* to the first character in the string. The value of a string is the address of its first character. Thus, in C, it's appropriate to say that a **string is a pointer**—in fact, a pointer to the string's first character. In this sense, strings are like arrays, because an array is also a pointer to its first element.

A *character array* or a *variable of type* `char *` can be initialized with a string in a definition. The definitions

```
char color[] = "blue";
const char *colorPtr = "blue";
```

each initialize a variable to the string `"blue"`. The first definition creates a 5-element array `color` containing the characters `'b'`, `'l'`, `'u'`, `'e'` and `'\0'`. The second definition creates pointer variable `colorPtr` that points to the string `"blue"` somewhere in memory.

Portability Tip 8.1

When a variable of type `char *` is initialized with a string literal, some compilers may place the string in a location in memory where it cannot *be modified. If you might need to modify a string literal, it should be stored in a character array to ensure modifiability on all systems.*

The preceding array definition could also have been written

```
char color[] = { 'b', 'l', 'u', 'e', '\0' };
```

When defining a character array to contain a string, the array must be large enough to store the string *and* its terminating null character. The preceding definition automatically determines the size of the array based on the number of initializers in the initializer list.

Common Programming Error 8.1

Not allocating sufficient space in a character array to store the null character that terminates a string is an error.

Common Programming Error 8.2

Printing a "string" that does not contain a terminating null character is an error.

Error-Prevention Tip 8.1

When storing a string of characters in a character array, be sure that the array is large enough to hold the largest string that will be stored. C allows strings of any *length to be stored. If a string is longer than the character array in which it's to be stored, characters beyond the end of the array will overwrite data in memory following the array.*

A string can be stored in an array using scanf. For example, the following statement stores a string in character array word[20]:

```
scanf( "%19s", word );
```

The string entered by the user is stored in word. Variable word is an array, which is, of course, a pointer, so the & is not needed with argument word. Recall from Section 6.4 that function scanf will read characters until a space, tab, newline or end-of-file indicator is encountered. So, it's possible that, without the field width 19 in the conversion specifier %19s, the user input could exceed 19 characters and that your program might crash! For this reason, you should *always* use a field width when using *scanf* to read into a char array. The field width 19 in the preceding statement ensures that scanf reads a *maximum* of 19 characters and saves the last character for the string's terminating null character. This prevents scanf from writing characters into memory beyond the end of s. (For reading input lines of arbitrary length, there's a nonstandard—yet widely supported—function readline, usually included in stdio.h.) For a character array to be printed properly as a string, the array must contain a terminating null character.

Common Programming Error 8.3

Processing a single character as a string. A string is a pointer—probably a respectably large integer. However, a character is a small integer (ASCII values range 0–255). On many systems this causes an error, because low memory addresses are reserved for special purposes such as operating-system interrupt handlers—so "access violations" occur.

Common Programming Error 8.4

Passing a character as an argument to a function when a string is expected (and vice versa) is a compilation error.

8.3 Character-Handling Library

The character-handling library (<ctype.h>) includes several functions that perform useful tests and manipulations of character data. Each function receives an unsigned char (represented as an int) or EOF as an argument. As we discussed in Chapter 4, characters are often manipulated as integers, because a character in C is a one-byte integer. EOF normally has the value –1. Figure 8.1 summarizes the functions of the character-handling library.

Prototype	Function description
int isblank(int c);	Returns a true value if c is a *blank character* that separates words in a line of text and 0 (false) otherwise. [*Note:* This function is not available in Microsoft Visual C++.]
int isdigit(int c);	Returns a true value if c is a *digit* and 0 (false) otherwise.
int isalpha(int c);	Returns a true value if c is a *letter* and 0 otherwise.
int isalnum(int c);	Returns a true value if c is a *digit* or a *letter* and 0 otherwise.

Fig. 8.1 | Character-handling library (<ctype.h>) functions. (Part 1 of 2.)

Prototype	Function description
`int isxdigit( int c );`	Returns a true value if `c` is a *hexadecimal digit character* and 0 otherwise. (See Appendix C for a detailed explanation of binary numbers, octal numbers, decimal numbers and hexadecimal numbers.)
`int islower( int c );`	Returns a true value if `c` is a *lowercase letter* and 0 otherwise.
`int isupper( int c );`	Returns a true value if `c` is an *uppercase letter* and 0 otherwise.
`int tolower( int c );`	If `c` is an *uppercase letter*, `tolower` returns `c` as a *lowercase letter*. Otherwise, `tolower` returns the argument unchanged.
`int toupper( int c );`	If `c` is a *lowercase letter*, `toupper` returns `c` as an *uppercase letter*. Otherwise, `toupper` returns the argument unchanged.
`int isspace( int c );`	Returns a true value if `c` is a *whitespace character*—newline (`'\n'`), space (`' '`), form feed (`'\f'`), carriage return (`'\r'`), horizontal tab (`'\t'`) or vertical tab (`'\v'`)—and 0 otherwise.
`int iscntrl( int c );`	Returns a true value if `c` is a *control character* and 0 otherwise.
`int ispunct( int c );`	Returns a true value if `c` is a *printing character other than a space, a digit, or a letter* and returns 0 otherwise.
`int isprint( int c );`	Returns a true value if `c` is a *printing character including a space* and returns 0 otherwise.
`int isgraph( int c );`	Returns a true value if `c` is a *printing character other than a space* and returns 0 otherwise.

Fig. 8.1 | Character-handling library (`<ctype.h>`) functions. (Part 2 of 2.)

8.3.1 Functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`

Figure 8.2 demonstrates functions **`isdigit`**, **`isalpha`**, **`isalnum`** and **`isxdigit`**. Function `isdigit` determines whether its argument is a digit (`0`–`9`). Function `isalpha` determines whether its argument is an uppercase (`A`–`Z`) or lowercase letter (`a`–`z`). Function `isalnum` determines whether its argument is an uppercase letter, a lowercase letter or a digit. Function `isxdigit` determines whether its argument is a **hexadecimal digit** (`A`–`F`, `a`–`f`, `0`–`9`).

```
// Fig. 8.2: fig08_02.c
// Using functions isdigit, isalpha, isalnum, and isxdigit
#include <stdio.h>
#include <ctype.h>

int main( void )
{
   printf( "%s\n%s%s\n%s%s\n\n", "According to isdigit: ",
      isdigit( '8' ) ? "8 is a " : "8 is not a ", "digit",
      isdigit( '#' ) ? "# is a " : "# is not a ", "digit" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to isalpha:",
```

Fig. 8.2 | Using functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 1 of 2.)

```
      isalpha( 'A' ) ? "A is a " : "A is not a ", "letter",
      isalpha( 'b' ) ? "b is a " : "b is not a ", "letter",
      isalpha( '&' ) ? "& is a " : "& is not a ", "letter",
      isalpha( '4' ) ? "4 is a " : "4 is not a ", "letter" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n\n",
      "According to isalnum:",
      isalnum( 'A' ) ? "A is a " : "A is not a ",
      "digit or a letter",
      isalnum( '8' ) ? "8 is a " : "8 is not a ",
      "digit or a letter",
      isalnum( '#' ) ? "# is a " : "# is not a ",
      "digit or a letter" );

   printf( "%s\n%s%s\n%s%s\n%s%s\n%s%s\n%s%s\n",
      "According to isxdigit:",
      isxdigit( 'F' ) ? "F is a " : "F is not a ",
      "hexadecimal digit",
      isxdigit( 'J' ) ? "J is a " : "J is not a ",
      "hexadecimal digit",
      isxdigit( '7' ) ? "7 is a " : "7 is not a ",
      "hexadecimal digit",
      isxdigit( '$' ) ? "$ is a " : "$ is not a ",
      "hexadecimal digit",
      isxdigit( 'f' ) ? "f is a " : "f is not a ",
      "hexadecimal digit" );
} // end main
```

```
According to isdigit:
8 is a digit
# is not a digit

According to isalpha:
A is a letter
b is a letter
& is not a letter
4 is not a letter

According to isalnum:
A is a digit or a letter
8 is a digit or a letter
# is not a digit or a letter

According to isxdigit:
F is a hexadecimal digit
J is not a hexadecimal digit
7 is a hexadecimal digit
$ is not a hexadecimal digit
f is a hexadecimal digit
```

Fig. 8.2 | Using functions `isdigit`, `isalpha`, `isalnum` and `isxdigit`. (Part 2 of 2.)

Figure 8.2 uses the conditional operator (`?:`) to determine whether the string `" is a "` or the string `" is not a "` should be printed in the output for each character tested. For example, the expression