Chapter 10 Characters and Strings

10-1. The char Data Type

C uses the char data type to hold characters. You saw on Day 3, "Storing Data: Variables and Constants," that char is one of C's numeric integer data types. If char is a numeric type, how can it be used to hold characters?

The answer lies in how C stores characters. Your computer's memory stores all data in numeric form. There is no direct way to store characters. However, a numeric code exists for each character. This is called the *ASCII code* or the *ASCII character set*. (*ASCII* stands for American Standard Code for Information Interchange.) The code assigns values between 0 and 255 for upper- and lowercase letters, numeric digits, punctuation marks, and other symbols. The ASCII character set is listed in Appendix A.

For example, 97 is the ASCII code for the letter a. When you store the character a in a type char variable, you're really storing the value 97. Because the allowable numeric range for type char matches the standard ASCII character set, char is ideally suited for storing characters.

At this point, you might be a bit puzzled. If C stores characters as numbers, how does your program know whether a given type char variable is a character or a number? As you'll learn later, declaring a variable as type char is not enough; you must do something else with the variable:

- If a char variable is used somewhere in a C program where a character is expected, it is interpreted as a character.
- If a char variable is used somewhere in a C program where a number is expected, it is interpreted as a number.

This gives you some understanding of how C uses a numeric data type to store character data. Now you can go on to the details.

10-2. Using Character Variables

Like other variables, you must declare chars before using them, and you can initialize them at the time of declaration. Here are some examples:

```
char a, b, c;         /* Declare three uninitialized char variables */
char code = `x';      /* Declare the char variable named code */
                      /* and store the character x there */
code = `!';           /* Store ! in the variable named code */
```

To create literal character constants, you enclose a single character in single quotation marks. The compiler automatically translates literal character constants into the corresponding ASCII codes, and the numeric code value is assigned to the variable.

You can create symbolic character constants by using either the #define directive or the const keyword:

```
#define EX `x'
char code = EX;      /* Sets code equal to `x' */
const char A = `Z';
```

Now that you know how to declare and initialize character variables, it's time for a demonstration. Listing 10.1 illustrates the numeric nature of character storage using the printf() function you learned on Day 7, "Fundamentals of Input and Output." The function printf() can be used to print both characters and numbers. The format string %c instructs printf() to print a character, whereas %d instructs it to print a decimal integer. Listing 10.1 initializes two type char variables and prints each one, first as a character, and then as a number.

Listing 10.1. The numeric nature of type char variables.

```
1:  /* Demonstrates the numeric nature of char variables */
2:
3:  #include <stdio.h>
4:
5:  /* Declare and initialize two char variables */
6:
7:  char c1 = `a';
8:  char c2 = 90;
9:
10: main()
11: {
12:     /* Print variable c1 as a character, then as a number */
13:
14:     printf("\nAs a character, variable c1 is %c", c1);
15:     printf("\nAs a number, variable c1 is %d", c1);
16:
17:     /* Do the same for variable c2 */
18:
19:     printf("\nAs a character, variable c2 is %c", c2);
20:     printf("\nAs a number, variable c2 is %d\n", c2);
21:
22:     return 0;
23: }
As a character, variable c1 is a
As a number, variable c1 is 97
As a character, variable c2 is Z
As a number, variable c2 is 90
```

ANALYSIS: You learned on Day 3 that the allowable range for a variable of type char goes only to 127, whereas the ASCII codes go to 255. The ASCII codes are actually divided into two parts. The standard ASCII codes go only to 127; this range includes all letters, numbers, punctuation marks, and other keyboard symbols. The codes from 128 to 255 are the extended ASCII codes and represent special characters such as foreign letters and graphics symbols (see Appendix A for a complete list). Thus, for standard text data, you can use type char variables; if you want to print the extended ASCII characters, you must use an unsigned char.

Listing 10.2 prints some of the extended ASCII characters.

Listing 10.2. Printing extended ASCII characters.

```
1:  /* Demonstrates printing extended ASCII characters */
2:
3:  #include <stdio.h>
4:
5:  unsigned char x;    /* Must be unsigned for extended ASCII */
6:
7:  main()
8:  {
9:      /* Print extended ASCII characters 180 through 203 */
10:
11:     for (x = 180; x < 204; x++)
```

```
12:     {
13:         printf("ASCII code %d is character %c\n", x, x);
14:     }
15:
16:         return 0;
17: }
ASCII code 180 is character ¥
ASCII code 181 is character µ
ASCII code 182 is character [partialdiff]
ASCII code 183 is character [Sigma]
ASCII code 184 is character [Pi]
ASCII code 185 is character [pi]
ASCII code 186 is character [integral]
ASCII code 187 is character ª
ASCII code 188 is character º
ASCII code 189 is character [Omega]
ASCII code 190 is character æ
ASCII code 191 is character ø
ASCII code 192 is character ¿
ASCII code 193 is character ¡
ASCII code 194 is character ¬
ASCII code 195 is character [radical]
ASCII code 196 is character [florin]
ASCII code 197 is character ~
ASCII code 198 is character [Delta]
ASCII code 199 is character «
ASCII code 200 is character »
ASCII code 201 is character ...
ASCII code 202 is character g
ASCII code 203 is character À
```

ANALYSIS: Looking at this program, you see that line 5 declares an unsigned character variable, x. This gives a range of 0 to 255. As with other numeric data types, you must not initialize a char variable to a value outside the allowed range, or you might get unexpected results. In line 11, x is not initialized outside the range; instead, it is initialized to 180. In the for statement, x is incremented by 1 until it reaches 204. Each time x is incremented, line 13 prints the value of x and the character value of x. Remember that %c prints the character, or ASCII, value of x.

DO use %c to print the character value of a number.

DON'T use double quotations when initializing a character variable.

DO use single quotations when initializing a variable.

DON'T try to put extended ASCII character values into a signed char variable.

DO look at the ASCII chart in Appendix A to see the interesting characters that can be printed.

NOTE: Some computer systems might use a different character set; however, most use the same values for 0 to 127.

10-3. Using Strings

Variables of type char can hold only a single character, so they have limited usefulness. You also need a way to store *strings,* which are sequences of characters. A person's name and address are examples of strings. Although there is no special data type for strings, C handles this type of information with arrays of characters.

10-3-1. Arrays of Characters

To hold a string of six characters, for example, you need to declare an array of type char with seven elements. Arrays of type char are declared like arrays of other data types. For example, the statement

```
char string[10];
```

declares a 10-element array of type char. This array could be used to hold a string of nine or fewer characters.

"But wait," you might be saying. "It's a 10-element array, so why can it hold only nine characters? In C, a string is defined as a sequence of characters ending with the null character, a special character represented by \0. Although it's represented by two characters (backslash and zero), the null character is interpreted as a single character and has the ASCII value of 0. It's one of C's escape sequences, covered on Day 7.

When a C program stores the string Alabama, for example, it stores the seven characters A, l, a, b, a, m, and a, followed by the null character \0, for a total of eight characters. Thus, a character array can hold a string of characters numbering one less than the total number of elements in the array.

A type char variable is one byte in size, so the number of bytes in an array of type char variables is the same as the number of elements in the array.

10-3-2. Initializing Character Arrays

Like other C data types, character arrays can be initialized when they are declared. Character arrays can be assigned values element by element, as shown here:

```
char string[10] = { `A', `l', `a', `b', `a', `m', `a', `\0' };
```

It's more convenient, however, to use a *literal string,* which is a sequence of characters enclosed in double quotes:

```
char string[10] = "Alabama";
```

When you use a literal string in your program, the compiler automatically adds the terminating null character at the end of the string. If you don't specify the number of subscripts when you declare an array, the compiler calculates the size of the array for you. Thus, the following line creates and initializes an eight-element array:

```
char string[] = "Alabama";
```

Remember that strings require a terminating null character. The C functions that manipulate strings (covered on Day 17, "Manipulating Strings") determine string length by looking for the null character. These functions have no other way of recognizing the end of the string. If the null character is missing, your program thinks that the string extends until the next null character in memory. Pesky program bugs can result from this sort of error.

10-4. Strings and Pointers

You've seen that strings are stored in arrays of type char, with the end of the string (which might not occupy the entire array) marked by the null character. Because the end of the string is marked, all you need in order to define a given string is something that points to its beginning. (Is *points* the right word? Indeed it is!)

With that hint, you might be leaping ahead of the game. From Day 9, "Understanding Pointers," you know that the name of an array is a pointer to the first element of the array. Therefore, for a string that's stored in an array, you need only the array name in order to access it. In fact, using the array's name is C's standard method of accessing strings.

To be more precise, using the array's name to access strings is the method the C library functions expect. The C standard library includes a number of functions that manipulate strings. (These functions are covered on Day 17.) To pass a string to one of these functions, you pass the array name. The same is true of the string display functions printf() and puts(), discussed later in this chapter.

You might have noticed that I mentioned "strings stored in an array" a moment ago. Does this imply that some strings aren't stored in arrays? Indeed it does, and the next section explains why.

10-5. Strings Without Arrays

From the preceding section, you know that a string is defined by the character array's name and a null character. The array's name is a type char pointer to the beginning of the string. The null marks the string's end. The actual space occupied by the string in an array is incidental. In fact, the only purpose the array serves is to provide allocated space for the string.

What if you could find some memory storage space without allocating an array? You could then store a string with its terminating null character there instead. A pointer to the first character could serve to specify the string's beginning just as if the string were in an allocated array. How do you go about finding memory storage space? There are two methods: One allocates space for a literal string when the program is compiled, and the other uses the malloc() function to allocate space while the program is executing, a process known as *dynamic allocation*.

10-5-1. Allocating String Space at Compilation

The start of a string, as mentioned earlier, is indicated by a pointer to a variable of type char. You might recall how to declare such a pointer:

```
char *message;
```

This statement declares a pointer to a variable of type char named message. It doesn't point to anything now, but what if you changed the pointer declaration to read

```
char *message = "Great Caesar\'s Ghost!";
```

When this statement executes, the string Great Caesar's Ghost! (with a terminating null character) is stored somewhere in memory, and the pointer message is initialized to point to the first character of the string. Don't worry where in memory the string is stored; it's handled automatically by the compiler. Once defined, message is a pointer to the string and can be used as such.

The preceding declaration/initialization is equivalent to the following, and the two notations *message and message[] also are equivalent; they both mean "a pointer to."

```
char message[] = "Great Caesar\'s Ghost!";
```

This method of allocating space for string storage is fine when you know what you need while writing the program. What if the program has varying string storage needs, depending on user input or other factors that are unknown when you're writing the program? You use the malloc() function, which lets you allocate storage space "on-the-fly."

10-5-2. The malloc() Function

The malloc() function is one of C's *memory allocation* functions. When you call malloc(), you pass it the number of bytes of memory needed. malloc() finds and reserves a block of memory of the required size and returns the address of the first byte in the block. You don't need to worry about where the memory is found; it's handled automatically.

The malloc() function returns an address, and its return type is a pointer to type void. Why void? A pointer to type void is compatible with all data types. Because the memory allocated by malloc() can be used to store any of C's data types, the void return type is appropriate.

The malloc() Function

```
#include <stdlib.h>
void *malloc(size_t size);
```

malloc() allocates a block of memory that is the number of bytes stated in *size*. By allocating memory as needed with malloc() instead of all at once when a program starts, you can use a computer's memory more efficiently. When using malloc(), you need to include the STDLIB.H header file. Some compilers have other header files that can be included; for portability, however, it's best to include STDLIB.H.

malloc() returns a pointer to the allocated block of memory. If malloc() was unable to allocate the required amount of memory, it returns null. Whenever you try to allocate memory, you should always check the return value, even if the amount of memory to be allocated is small.

Example 1

```
#include <stdlib.h>
#include <stdio.h>
main()
{
   /* allocate memory for a 100-character string */
   char *str;
   if (( str = (char *) malloc(100)) == NULL)
   {
      printf( "Not enough memory to allocate buffer\n");
      exit(1);
   }
   printf( "String was allocated!"\n );
   return 0;
}
```

Example 2

```
/* allocate memory for an array of 50 integers */
int *numbers;
numbers = (int *) malloc(50 * sizeof(int));
```

Example 3

```
/* allocate memory for an array of 10 float values */
float *numbers;
numbers = (float *) malloc(10 * sizeof(float));
```

10-5-3. Using the malloc() Function

You can use malloc() to allocate memory to store a single type char. First, declare a pointer to type char:

```
char *ptr;
```

Next, call malloc() and pass the size of the desired memory block. Because a type char usually occupies one byte, you need a block of one byte. The value returned by malloc() is assigned to the pointer:

```
ptr = malloc(1);
```

This statement allocates a memory block of one byte and assigns its address to ptr. Unlike variables that are declared in the program, this byte of memory has no name. Only the pointer can reference the variable. For example, to store the character `x' there, you would write

```
*ptr = `x';
```

Allocating storage for a string with malloc() is almost identical to using malloc() to allocate space for a single variable of type char. The main difference is that you need to know the amount of space to allocate--the maximum number of characters in the string. This maximum depends on the needs of your program. For this example, say you want to allocate space for a string of 99 characters, plus one for the terminating null character, for a total of 100. First you declare a pointer to type char, and then you call malloc():

```
char *ptr;
ptr = malloc(100);
```

Now ptr points to a reserved block of 100 bytes that can be used for string storage and manipulation. You can use ptr just as though your program had explicitly allocated that space with the following array declaration:

```
char ptr[100];
```

Using malloc() lets your program allocate storage space as needed in response to demand. Of course, available space is not unlimited; it depends on the amount of memory installed in your computer and on the program's other storage requirements. If not enough memory is available, malloc() returns 0 (null). Your program should test the return value of malloc() so that you'll know the memory requested was allocated successfully. You should always test malloc()'s return value against the symbolic constant NULL, which is defined in STDLIB.H. Listing 10.3 illustrates the use of malloc(). Any program using malloc() must #include the header file STDLIB.H.

Listing 10.3. Using the malloc() function to allocate storage space for string data.

```
1:  /* Demonstrates the use of malloc() to allocate storage */
2:  /* space for string data. */
3:
4:  #include <stdio.h>
5:  #include <stdlib.h>
6:
7:  char count, *ptr, *p;
8:
9:  main()
10: {
```

```
11:    /* Allocate a block of 35 bytes. Test for success. */
12:    /* The exit() library function terminates the program. */
13:
14:    ptr = malloc(35 * sizeof(char));
15:
16:    if (ptr == NULL)
17:    {
18:        puts("Memory allocation error.");
19:        exit(1);
20:    }
21:
22:    /* Fill the string with values 65 through 90, */
23:    /* which are the ASCII codes for A-Z. */
24:
25:    /* p is a pointer used to step through the string. */
26:    /* You want ptr to remain pointed at the start */
27:    /* of the string. */
28:
29:    p = ptr;
30:
31:    for (count = 65; count < 91 ; count++)
32:        *p++ = count;
33:
34:    /* Add the terminating null character. */
35:
36:    *p = `\0';
37:
38:    /* Display the string on the screen. */
39:
40:    puts(ptr);
41:
42:    return 0;
43: }
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

ANALYSIS: This program uses malloc() in a simple way. Although the program seems long, it's filled with comments. Lines 1, 2, 11, 12, 22 through 27, 34, and 38 are all comments that detail everything the program does. Line 5 includes the STDLIB.H header file needed for malloc(), and line 4 includes the STDIO.H header file for the puts() functions. Line 7 declares two pointers and a character variable used later in the listing. None of these variables is initialized, so they shouldn't be used--yet!

The malloc() function is called in line 14 with a parameter of 35 multiplied by *the size of* a char. Could you have just used 35? Yes, but you're assuming that everyone running this program will be using a computer that stores char type variables as one byte in size. Remember from Day 3 that different compilers can use different-size variables. Using the sizeof operator is an easy way to create portable code.

Never assume that malloc() gets the memory you tell it to get. In fact, you aren't *telling* it to get memory--you're *asking* it. Line 16 shows the easiest way to check whether malloc() provided the memory. If the memory was allocated, ptr points to it; otherwise, ptr is null. If the program failed to get the memory, lines 18 and 19 display an error message and gracefully exit the program.

Line 29 initializes the other pointer declared in line 7, p. It is assigned the same address value as ptr. A for loop uses this new pointer to place values into the allocated memory. Looking at line 31, you see that count is initialized to 65 and incremented by 1 until it reaches 91. For each loop of the for statement, the value of count is assigned to

the address pointed to by p. Notice that each time count is incremented, the address pointed to by p is also incremented. This means that each value is placed one after the other in memory.

You should have noticed that numbers are being assigned to count, which is a type char variable. Remember the discussion of ASCII characters and their numeric equivalents? The number 65 is equivalent to A, 66 equals B, 67 equals C, and so on. The for loop ends after the alphabet is assigned to the memory locations pointed to. Line 36 caps off the character values pointed to by putting a null at the final address pointed to by p. By appending the null, you can now use these values as a string. Remember that ptr still points to the first value, A, so if you use it as a string, it prints every character until it reaches the null. Line 40 uses puts() to prove this point and to show the results of what has been done.

DON'T allocate more memory than you need. Not everyone has a lot of memory, so you should try to use it sparingly.

DON'T try to assign a new string to a character array that was previously allocated only enough memory to hold a smaller string. For example, in this declaration:

```
char a_string[] = "NO";
```

a_string points to "NO". If you try to assign "YES" to this array, you could have serious problems. The array initially could hold only three characters--'N', `O', and a null. "YES" is four characters--'Y', `E', `S', and a null. You have no idea what the fourth character, null, overwrites.

10-6. Displaying Strings and Characters

If your program uses string data, it probably needs to display the data on the screen at some time. String display is usually done with either the puts() function or the printf() function.

10-6-1. The puts() Function

You've seen the puts() library function in some of the programs in this book. The puts() function puts a string on-screen--hence its name. A pointer to the string to be displayed is the only argument puts() takes. Because a literal string evaluates as a pointer to a string, puts() can be used to display literal strings as well as string variables. The puts() function automatically inserts a newline character at the end of each string it displays, so each subsequent string displayed with puts() is on its own line.

Listing 10.4 illustrates the use of puts().

Listing 10.4. Using the puts() function to display text on-screen.

```
1:  /* Demonstrates displaying strings with puts(). */
2:
3:  #include <stdio.h>
4:
5:  char *message1 = "C";
6:  char *message2 = "is the";
7:  char *message3 = "best";
8:  char *message4 = "programming";
9:  char *message5 = "language!!";
10:
11: main()
```

```
12: {
13:    puts(message1);
14:    puts(message2);
15:    puts(message3);
16:    puts(message4);
17:    puts(message5);
18:
19:    return 0;
20: }
C
is the
best
programming
language!!
```

ANALYSIS: This is a fairly simple listing to follow. Because puts() is a standard output function, the STDIO.H header file needs to be included, as done on line 3. Lines 5 through 9 declare and initialize five different message variables. Each of these variables is a character pointer, or string variable. Each of these variables is a character pointer, or string variable. Lines 13 through 17 use the puts() function to print each string.

10-6-2. The printf() Function

You can also display strings using the printf() library function. Recall from Day 7 that printf() uses a format string and conversion specifiers to shape its output. To display a string, use the conversion specifier %s.

When printf() encounters a %s in its format string, the function matches the %s with the corresponding argument in its argument list. For a string, this argument must be a pointer to the string that you want displayed. The printf() function displays the string on-screen, stopping when it reaches the string's terminating null character. For example:

```
char *str = "A message to display";
printf("%s", str);
```

You can also display multiple strings and mix them with literal text and/or numeric variables:

```
char *bank = "First Federal";
char *name = "John Doe";
int balance = 1000;
printf("The balance at %s for %s is %d.", bank, name, balance);
```

The resulting output is

```
The balance at First Federal for John Doe is 1000.
```

For now, this information should be sufficient for you to be able to display string data in your programs. Complete details on using printf() are given on Day 14, "Working with the Screen, Printer, and Keyboard."

10-7. Reading Strings from the Keyboard

In addition to displaying strings, programs often need to accept inputted string data from the user via the keyboard. The C library has two functions that can be used for this purpose--gets() and scanf(). Before you can read in a string from the keyboard, however, you must have somewhere to put it. You can create space for string storage using either of the methods discussed earlier--an array declaration or the malloc() function.

10-7-1. Inputting Strings Using the gets() Function

The gets() function gets a string from the keyboard. When gets() is called, it reads all characters typed at the keyboard up to the first newline character (which you generate by pressing Enter). This function discards the newline, adds a null character, and gives the string to the calling program. The string is stored at the location indicated by a pointer to type char passed to gets(). A program that uses gets() must #include the file STDIO.H. Listing 10.5 presents an example.

Listing 10.5. Using gets() to input string data from the keyboard.

```
1:  /* Demonstrates using the gets() library function. */
2:
3:  #include <stdio.h>
4:
5:  /* Allocate a character array to hold input. */
6:
7:  char input[81];
8:
9:  main()
10: {
11:     puts("Enter some text, then press Enter: ");
12:     gets(input);
13:     printf("You entered: %s\n", input);
14:
15:     return 0;
16: }
Enter some text, then press Enter:
This is a test
You entered: This is a test
```

ANALYSIS: [endd] In this example, the argument to gets() is the expression input, which is the name of a type char array and therefore a pointer to the first array element. The array is declared with 81 elements in line 7. Because the maximum line length possible on most computer screens is 80 characters, this array size provides space for the longest possible input line (plus the null character that gets() adds at the end).

The gets() function has a return value, which was ignored in the previous example. gets() returns a pointer to type char with the address where the input string is stored. Yes, this is the same value that is passed to gets(), but having the value returned to the program in this way lets your program test for a blank line. Listing 10.6 shows how to do this.

Listing 10.6. Using the gets() return value to test for the input of a blank line.

```
1:  /* Demonstrates using the gets() return value. */
2:
3:  #include <stdio.h>
4:
5:  /* Declare a character array to hold input, and a pointer. */
6:
7:  char input[81], *ptr;
8:
9:  main()
10: {
11:     /* Display instructions. */
```

```
12:
13:     puts("Enter text a line at a time, then press Enter.");
14:     puts("Enter a blank line when done.");
15:
16:     /* Loop as long as input is not a blank line. */
17:
18:     while ( *(ptr = gets(input)) != NULL)
19:         printf("You entered %s\n", input);
20:
21:     puts("Thank you and good-bye\n");
22:
23:     return 0;
24: }
Enter text a line at a time, then press Enter.
Enter a blank line when done.
First string
You entered First string
Two
You entered Two
Bradley L. Jones
You entered Bradley L. Jones
Thank you and good-bye
```

ANALYSIS: Now you can see how the program works. If you enter a blank line (that is, if you simply press Enter) in response to line 18, the string (which contains 0 characters) is still stored with a null character at the end. Because the string has a length of 0, the null character is stored in the first position. This is the position pointed to by the return value of gets(), so if you test that position and find a null character, you know that a blank line was entered.

Listing 10.6 performs this test in the while statement in line 18. This statement is a bit complicated, so look carefully at the details in order. 10.1 illustrates the components of this statement.

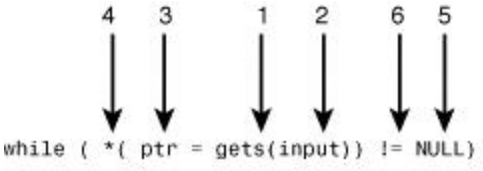

Figure 10-1. The components of a while statement that tests for the input of a blank line.

1. The gets() function accepts input from the keyboard until it reaches a newline character.
2. The input string, minus the newline and with a trailing null character, is stored in the memory location pointed to by input.
3. The address of the string (the same value as input) is returned to the pointer ptr.
4. An *assignment statement* is an expression that evaluates to the value of the variable on the left side of the assignment operator. Therefore, the entire expression ptr = gets(input) evaluates to the value of ptr. By enclosing this expression in parentheses and preceding it with the indirection operator (*), you obtain the value stored at the pointed-to address. This is, of course, the first character of the input string.
5. NULL is a symbolic constant defined in the header file STDIO.H. It has the value of the null character (0).
6. If the first character of the input string isn't the null character (if a blank line hasn't been entered), the comparison operator returns true, and the while loop executes. If the first character is the null character (if a blank line has been entered), the comparison operator returns false, and the while loop terminates.

When you use gets() or any other function that stores data using a pointer, be sure that the pointer points to allocated space. It's easy to make a mistake such as this:

```
char *ptr;
gets(ptr);
```

The pointer ptr has been declared but not initialized. It points somewhere, but you don't know where. The gets() function doesn't know this, so it simply goes ahead and stores the input string at the address contained in ptr. The string might overwrite something important, such as program code or the operating system. The compiler doesn't catch these kinds of mistakes, so you, the programmer, must be vigilant.

The gets() Function

```
#include <stdio.h>
char *gets(char *str);
```

The gets() function gets a string, str, from the standard input device, usually the keyboard. The string consists of any characters entered until a newline character is read. At that point, a null is appended to the end of the string.

Then the gets() function returns a pointer to the string just read. If there is a problem getting the string, gets() returns null.

Example

```
/* gets() example */
#include <stdio.h>
char line[256];
void main()
{
   printf( "Enter a string:\n");
   gets( line );
   printf( "\nYou entered the following string:\n" );
   printf( "%s\n", line );
}
```

10-7-2. Inputting Strings Using the scanf() Function

You saw on Day 7 that the scanf() library function accepts numeric data input from the keyboard. This function can also input strings. Remember that scanf() uses a *format string* that tells it how to read the input. To read a string, include the specifier %s in scanf()'s format string. Like gets(), scanf() is passed a pointer to the string's storage location.

How does scanf() decide where the string begins and ends? The beginning is the first nonwhitespace character encountered. The end can be specified in one of two ways. If you use %s in the format string, the string runs up to (but not including) the next whitespace character (space, tab, or newline). If you use %ns (where n is an integer constant that specifies field width), scanf() inputs the next n characters or up to the next whitespace character, whichever comes first.

You can read in multiple strings with scanf() by including more than one %s in the format string. For each %s in the format string, scanf() uses the preceding rules to find the requested number of strings in the input. For example:

```
scanf("%s%s%s", s1, s2, s3);
```

If in response to this statement you enter January February March, January is assigned to the string s1, February is assigned to s2, and March to s3.

What about using the field-width specifier? If you execute the statement

```
scanf("%3s%3s%3s", s1, s2, s3);
```

and in response you enter September, Sep is assigned to s1, tem is assigned to s2, and ber is assigned to s3.

What if you enter fewer or more strings than the scanf() function expects? If you enter fewer strings, scanf() continues to look for the missing strings, and the program doesn't continue until they're entered. For example, if in response to the statement

```
scanf("%s%s%s", s1, s2, s3);
```

you enter January February, the program sits and waits for the third string specified in the scanf() format string. If you enter more strings than requested, the unmatched strings remain pending (waiting in the keyboard buffer) and are read by any subsequent scanf() or other input statements. For example, if in response to the statements

```
scanf("%s%s", s1, s2);
scanf("%s", s3);
```

you enter January February March, the result is that January is assigned to the string s1, February is assigned to s2, and March is assigned to s3.

The scanf() function has a return value, an integer value equaling the number of items successfully inputted. The return value is often ignored. When you're reading text only, the gets() function is usually preferable to scanf(). It's best to use the scanf() function when you're reading in a combination of text and numeric data. This is illustrated by Listing 10.7. Remember from Day 7 that you must use the address-of operator (&) when inputting numeric variables with scanf().

Listing 10.7. Inputting numeric and text data with scanf().

```
char state[10]="Minneapolis";  /* Wrong! String longer than array. */
char state2[10]="MN";          /* OK, but wastes space because */
```

ANALYSIS: Remember that scanf() requires the addresses of variables for parameters. In Listing 10.7, lname and fname are pointers (that is, addresses), so they don't need the address-of operator (&). In contrast, id_num is a regular variable name, so it requires the & when passed to scanf() on line 17.

Some programmers feel that data entry with scanf() is prone to errors. They prefer to input all data, numeric and string, using gets(), and then have the program separate the numbers and convert them to numeric variables. Such techniques are beyond the scope of this book, but they would make a good programming exercise. For that task, you need the string manipulation functions covered on Day 17.

10-8. Summary

This chapter covered C's char data type. One use of type char variables is to store individual characters. You saw that characters are actually stored as numbers: The ASCII code assigns a numerical code to each character. Therefore, you can use type char to store small integer values as well. Both signed and unsigned char types are available.

A string is a sequence of characters terminated by the null character. Strings can be used for text data. C stores strings in arrays of type char. To store a string of length *n*, you need an array of type char with *n*+1 elements.

You can use memory allocation functions such as malloc() to make your programs more dynamic. By using malloc(), you can allocate the right amount of memory for your program. Without such functions, you would have to guess at the amount of memory storage the program needs. Your estimate would probably be high, so you would allocate more memory than needed.

Q&A

Q What is the difference between a string and an array of characters?
A A string is defined as a sequence of characters ending with the null character. An array is a sequence of characters. A string, therefore, is a null-terminated array of characters.

If you define an array of type char, the actual storage space allocated for the array is the specified size, not the size minus 1. You're limited to that size; you can't store a larger string. Here's an example:

```
char state[10]="Minneapolis";  /* Wrong! String longer than array. */
char state2[10]="MN";          /* OK, but wastes space because */

                               /* string is shorter than array. */
```

If, on the other hand, you define a pointer to type char, these restrictions don't apply. The variable is a storage space only for the pointer. The actual strings are stored elsewhere in memory (but you don't need to worry about where in mem-ory). There's no length restriction or wasted space. The actual string is stored else-where. A pointer can point to a string of any length.

Q Why shouldn't I just declare big arrays to hold values instead of using a memory allocation function such as malloc()?

A Although it might seem easier to declare large arrays, this isn't an effective use of memory. When you're writing small programs, such as those in this chapter, it might seem trivial to use a function such as malloc() instead of arrays, but as your programs get bigger, you'll want to be able to allocate memory only as needed. When you're done with memory, you can put it back by *freeing* it. When you free memory, some other variable or array in a different part of the program can use it. (Day 20, "Working with Memory," covers freeing allocated memory.)

Q Do all computers support the extended ASCII character set?

A No. Most PCs support the extended ASCII set. Some older PCs don't, but the number of older PCs lacking this support is diminishing. Most programmers use the line and block characters of the extended set.

Q What happens if I put a string into a character array that is bigger than the array?

A This can cause a hard-to-find error. You can do this in C, but anything stored in the memory directly after the character array is overwritten. This could be an area of memory not used, some other data, or some vital system information. Your results will depend on what you overwrite. Often, nothing happens for a while. You don't want to do this.

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 range of numeric values in the ASCII character set?

2. When the C compiler encounters a single character enclosed in single quotation marks, how is it interpreted?

3. What is C's definition of a string?

4. What is a literal string?

5. To store a string of *n* characters, you need a character array of *n*+1 elements. Why is the extra element needed?

6. When the C compiler encounters a literal string, how is it interpreted?

7. Using the ASCII chart in Appendix A, state the numeric value stored for each of the following:

- **a.** a
- **b.** A
- **c.** 9
- **d.** a space
- **e.** OE
- **f.** F

8. Using the ASCII chart in Appendix A, translate the following numeric values to their equivalent characters:

- **a.** 73
- **b.** 32
- **c.** 99
- **d.** 97
- **e.** 110
- **f.** 0

g. 2

9. How many bytes of storage are allocated for each of the following variables? (Assume that a character is one byte.)

a. char *str1 = { "String 1" };
b. char str2[] = { "String 2" };
c. char string3;
d. char str4[20] = { "This is String 4" };
e. char str5[20];

10. Using the following declaration:

```
char *string = "A string!";
```

What are the values of the following?

a. string[0]
b. *string
c. string[9]
d. string[33]
e. *string+8
f. string

Exercises

1. Write a line of code that declares a type char variable named letter, and initialize it to the character $.
2. Write a line of code that declares an array of type char, and initialize it to the string "Pointers are fun!". Make the array just large enough to hold the string.
3. Write a line of code that allocates storage for the string "Pointers are fun!", as in exercise 2, but without using an array.
4. Write code that allocates space for an 80-character string and then inputs a string from the keyboard and stores it in the allocated space.
5. Write a function that copies one array of characters into another. (Hint: Do this just like the programs you wrote on Day 9.)
6. Write a function that accepts two strings. Count the number of characters in each, and return a pointer to the longer string.
7. ON YOUR OWN: Write a function that accepts two strings. Use the malloc() function to allocate enough memory to hold the two strings after they have been concatenated (linked). Return a pointer to this new string.
For example, if I pass "Hello " and "World!", the function returns a pointer to "Hello World!". Having the concatenated value be the third string is easiest. (You might be able to use your answers from exercises 5 and 6.)
8. BUG BUSTER: Is anything wrong with the following?

```
char a_string[10] = "This is a string";
```

9. BUG BUSTER: Is anything wrong with the following?

```
char *quote[100] = { "Smile, Friday is almost here!" };
```

10. BUG BUSTER: Is anything wrong with the following?

```
char *string1;
char *string2 = "Second";
string1 = string2;
```

11. BUG BUSTER: Is anything wrong with the following?

```
char string1[];
char string2[] = "Second";
string1 = string2;
```

12. ON YOUR OWN: Using the ASCII chart, write a program that prints a box on-screen using the double-line characters.