CHAPTER 10

Essential Input and Output Operations

In this chapter you're going to look in more detail at input from the keyboard, output to the screen, and output to a printer. The good news is that everything in this chapter is fairly easy, although there may be moments when you feel it's all becoming a bit of a memory test. Treat this as a breather from the last two chapters. After all, you don't have to memorize everything you see here; you can always come back to it when you need it.

Like most modern programming languages, the C language has no input or output capability within the language. All operations of this kind are provided by functions from standard libraries. You've been using many of these functions to provide input from the keyboard and output to the screen in all the preceding chapters.

This chapter will put all the pieces together into some semblance of order and round it out with the aspects I haven't explained so far. I'll also add a bit about printing because it's usually a fairly essential facility for a program. You don't have a program demonstrating a problem solution with this chapter for the simple reason that I don't really cover anything that requires any practice on a substantial example (it's that easy).

In this chapter you'll learn the following:

- How to read data from the keyboard
- How to format data for output on the screen
- How to deal with character output
- How to output data to a printer

Input and Output Streams

Up to now you've primarily used `scanf()` for keyboard input and `printf()` for output to the screen. Actually, there has been nothing in particular about the way you've used these functions to specify where the input came from or where the output went. The information that `scanf()` received could have come from anywhere, as long as it was a suitable stream of characters. Similarly, the output from `printf()` could have been going anywhere that could accept a stream of characters. This is no accident: the standard input/output functions in C have been designed to be device-independent, so that the transfer of data to or from a specific device isn't a concern of the programmer. The C library functions and the operating system make sure that operations with a specific device are executed correctly.

Each input source and output destination in C is called a **stream**. An **input stream** is a source of data that can be read into your program, and an **output stream** is a destination for data that originates in your program. A stream is independent of the physical piece of equipment involved, such as the display or the keyboard. Each device that a program uses will usually have one or more streams associated with it, depending on whether it's simply an input device such as a keyboard, or an output device

such as a printer, or a device that can have both input and output operations, such as a disk drive. This is illustrated in Figure 10-1.

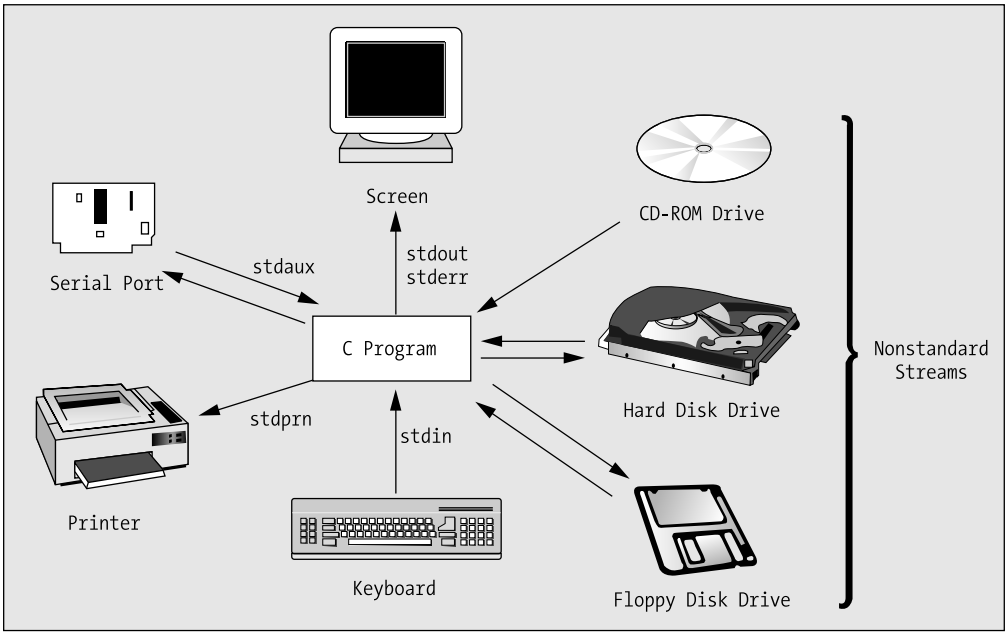

Figure 10-1. *Standard and nonstandard streams*

A disk drive can have multiple input and output streams because it can contain multiple files. The correspondence is between a stream and a file, not between a stream and the device. A stream can be associated with a specific file on the disk. The stream that you associate with a particular file could be an input stream, so you could only read from the file; it could be an output stream, in which case you could only write to the file; or the stream might allow input and output so reading and writing the file would both be possible. Obviously, if the stream that is associated with a file is an input stream, the file must have been written at some time so it contained some data. You could also associate a stream with a file on a CD-ROM drive. Because this device is typically read-only, the stream would, of necessity, be an input stream.

There are two further kinds of streams: **character streams**, which are also referred to as **text streams**, and **binary streams**. The main difference between these is that data transferred to or from character streams is treated as a sequence of characters and may be modified by the library routine concerned, according to a format specification. Data that's transferred to or from binary streams is just a sequence of bytes that isn't modified in any way. I discuss binary streams in Chapter 12 when I cover reading and writing disk files.

Standard Streams

C has three predefined **standard streams** that are automatically available in any program, provided, of course, that you've included the `<stdio.h>` header file, which contains their definitions, into your program. These standard streams are `stdin`, `stdout`, and `stderr`. Two other streams that are available

with some systems are identified by the names `stdprn` and `stdaux`, but these are not part of the C language standard so your compiler may not support them.

No initialization or preparation is necessary to use these streams. You just have to apply the appropriate library function that sends data to them. They are each preassigned to a specific physical device, as shown in Table 10-1.

Table 10-1. *Standard Streams*

Stream	Device
`stdin`	Keyboard
`stdout`	Display screen
`stderr`	Display screen
`stdprn`	Printer
`stdaux`	Serial port

In this chapter I concentrate on how you can use the standard input stream, `stdin`, the standard output stream, `stdout`, and the printer stream, `stdprn`.

The `stderr` stream is simply the stream to which error messages from the C library are sent, and you can direct your own error messages to `stderr` if you wish. The main difference between `stdout` and `stderr` is that output to `stdout` is buffered in memory so the data that you write to it won't necessarily be transferred immediately to the device, whereas `stderr` is unbuffered so any data you write to it is always transferred immediately to the device. With a buffered stream your program transfers data to or from a buffer area in memory, and the actual data transfer to or from the physical device can occur asynchronously. This makes the input and output operations much more efficient. The advantage of using an unbuffered stream for error messages is that you can be sure that they will actually be displayed but the output operations will be inefficient; a buffered stream is efficient but may not get flushed when a program fails for some reason, so the output may never be seen. I won't discuss this further, other than to say `stderr` points to the display screen and can't be redirected to another device. Output to the stream `stdaux` is directed to the serial port and is outside the scope of this book for reasons of space rather than complexity.

Both `stdin` and `stdout` can be reassigned to files, instead of the default of keyboard and screen, by using operating system commands. This offers you a lot of flexibility. If you want to run your program several times with the same data, during testing for example, you could prepare the data as a text file and redirect `stdin` to the file. This enables you to rerun the program with the same data without having to re-enter it each time. By redirecting the output from your program to a file, you can easily retain it for future reference, and you could use a text editor to access it or search it.

Input from the Keyboard

There are two forms of input from the keyboard on `stdin` that you've already seen in previous chapters: **formatted input**, which is provided primarily by the `scanf()` function and **unformatted input**, in which you receive the raw character data from a function such as `getchar()`. There's rather more to both of these possibilities, so let's look at them in detail.

Formatted Keyboard Input

As you know, the function scanf() reads characters from the stream stdin and converts them to one or more values according to the format specifiers in a format control string. The prototype of the scanf() function is as follows:

```
int scanf(char *format, ... );
```

The format control string parameter is of type char *, a pointer to a character string as shown here. However, this usually appears as an explicit argument in the function call, such as

```
scanf("%lf", &variable);
```

But there's nothing to prevent you from writing this:

```
char str[] = "%lf";
scanf(str, &variable);
```

The scanf() function makes use of the facility of handling a variable number of arguments that you learned about in Chapter 9. The format control string is basically a coded description of how scanf() should convert the incoming character stream to the values required. Following the format control string, you can have one or more optional arguments, each of which is an address in which a corresponding converted input value is to be stored. As you've seen, this implies that each of these arguments must be a pointer or a variable name prefixed by & to define the address of the variable rather than its value.

The scanf() function reads from stdin until it comes to the end of the format control string, or until an error condition stops the input process. This sort of error is the result of input that doesn't correspond to what is expected with the current format specifier, as you'll see. Something that I haven't previously noted is that scanf() returns a value that is the count of the number of input values read. This provides a way for you to detect when an error occurs by comparing the value returned by scanf() with the number of input values you are expecting.

The wscanf() function provides exactly the same capabilities as scanf() except that the first argument to the function, which is the format control string, must be a wide character string of type wchar_t *.

Thus you could use wscanf() to read a floating-point value from the keyboard like this:

```
wscanf(L"%lf", &variable);
```

The first argument is a wide character string constant and in all other respects the function works like scanf(). If you omit the L for the wide character string literal, you will get an error message from the compiler because your argument does not match the type of the first parameter.

Of course, you could also write this:

```
wchar_t wstr[] = L"%lf";
wscanf(wstr, &variable);
```

Input Format Control Strings

The format control string that you use with scanf() or wscanf() isn't precisely the same as that used with printf(). For one thing, putting one or more whitespace characters—blank ' ', tab '\t', or newline '\n'—in the format control string causes scanf() to read and ignore whitespace characters up to the next nonwhitespace character in the input. A single whitespace character in the format control string causes any number of consecutive whitespace characters to be ignored. You can therefore include as many whitespace characters as you wish in the format string to make it more readable. Note that whitespace characters are ignored by scanf() by default except when you are reading data using %c, %[], or %n specifications (see Table 10-2).

Any nonwhitespace character other than % will cause scanf() to read but not store successive occurrences of the character. If you want scanf() to ignore commas separating values in the input for instance, just precede each format specifier by a comma. There are other differences too, as you'll see when I discuss formatted output in the section "Output to the Screen" a bit later in this chapter.

The most general form of a format specifier is shown in Figure 10-2.

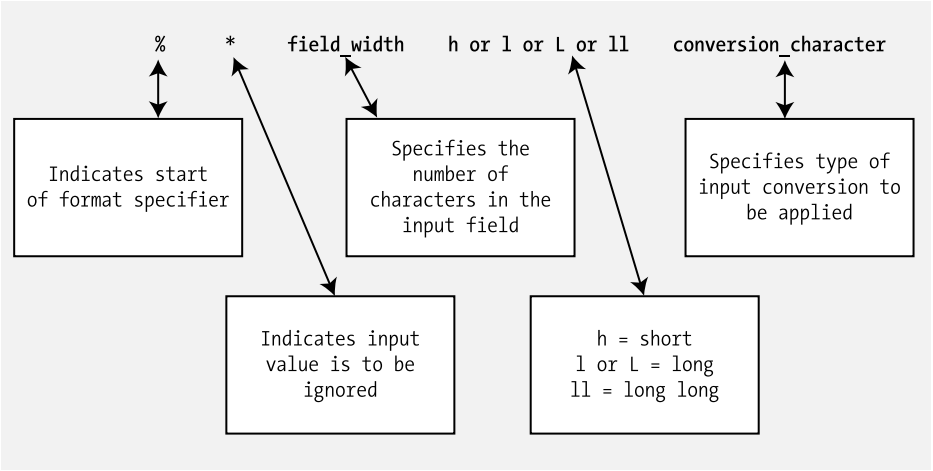

Figure 10-2. *The general form of an output specifier*

Let's take a look at what the various parts of this general form mean:

- The % simply indicates the start of the format specifier. It must always be present.
- The next * is optional. If you include it, it indicates that the next input value is to be ignored. This isn't normally used with input from the keyboard. It does become useful, however, when stdin has been reassigned to a file and you don't want to process all the values that appear within the file in your program.
- The field width is optional. It's an integer specifying the number of characters that scanf() should assume makes up the current value being input. This allows you to input a sequence of values without spaces between them. This is also often quite useful when reading files.
- The next character is also optional, and it can be h, L, l (the lowercase letter L), or ll (two lowercase Ls). If it's h, it can only be included with an integer conversion specifier (d, I, o, u, or x) and indicates that the input is to be converted as short. If it's l, it indicates long when preceding an int conversion specifier, and double when preceding a float conversion specifier. Prefixing the c specification with l specifies a wide character conversion so the input is read as wchar_t. The prefix L applied to e, E, f, g, or G specifies the value is of type long double. The ll prefix applies to integer conversions and specifies that the input is to be stored as type long long.
- The conversion character specifies the type of conversion to be carried out on the input stream and therefore must be included. The possible characters and their meanings are shown in Table 10-2.

Table 10-2. *Conversion Characters and Their Meanings*

Conversion Character	Meaning
`d`	Convert input to `int`.
`i`	Convert input to `int`. If preceded by `0`, then assume octal digits input. If preceded by `0x` or `0X`, then assume hexadecimal digits input.
`o`	Convert input to `int` and assume all digits are octal.
`u`	Convert input to `unsigned int`.
`x`	Convert to `int` and assume all digits are hexadecimal.
`c`	Read the next character as `char` (including whitespace). If you want to ignore whitespace when reading a single character, just precede the format specification by a whitespace character.
`s`	Input a string of successive nonwhitespace characters, starting with the next nonwhitespace character.
`e`, `f`, or `g`	Convert input to type `float`. A decimal point and an exponent in the input are optional.
`n`	No input is read but the number characters that have been read from the input source up to this point are stored in the corresponding argument, which should be of type `int*`.

You can also read a string that consists of specific characters by placing all the possible characters between square brackets in the specification. For example, the specification `%[0123456789.-]` will read a numerical value as a string, so if the input is –1.25 it will be read as `"-1.25"`. To read a string consisting of the lowercase letters *a* to *z*, you could use the specification `%[abcdefghijklmnopqrstuvwxyz]`. This will read any sequence of the characters that appear between the square brackets as a string, and the first character in the input that isn't in the set between the square brackets marks the end of the input. Although it isn't required by the standard, many C library implementations support the form `%[a-z]` to read a string consisting of any lowercase letters.

A specification using square brackets is also very useful for reading strings that are delimited by characters other than whitespace. In this case, you can specify the characters that are *not* in the string by using the `^` character as the first character in the set. Thus, the specification `%[^,]` will include everything in the string except a comma, so this form will enable you to read a series of strings separated by commas.

Table 10-3 shows a few examples of applying the various options.

Table 10-3. *Examples of Options in Conversion Specifications*

Specification	Description
`%lf`	Read the next value as type `double`
`%*d`	Read the next integer value but don't store it
`%lc`	Read the next character as type `wchar_t`
`%\nc`	Read the next character as type `char` ignoring whitespace characters
`%10lld`	Reads the next ten characters as an integer value of type `long long`

Table 10-3. *Examples of Options in Conversion Specifications*

Specification	Description
`%5d`	Read the next five characters as an integer
`%Lf`	Read the next value as a floating-point value of type `long double`
`%hu`	Read the next value as type `unsigned short`

Let's exercise some of these format control strings with practical examples.

TRY IT OUT: EXERCISING FORMATTED INPUT

To start, let's read a variety of data and then output the result:

```
/* Program 10.1        Exercising formatted input */
#include <stdio.h>

const size_t SIZE = 20;      /* Max characters in a word */

int main(void)
{
  int value_count = 0;       /* Count of input values read */
  float fp1 = 0.0;           /* Floating-point value read  */
  int i = 0;                 /* First integer read         */
  int j = 0;                 /* Second integer read        */
  char word1[SIZE] = " ";    /* First string read          */
  char word2[SIZE] = " ";    /* Second string read         */
  int byte_count = 0;        /* Count of input bytes read  */

  value_count = scanf("%f %d %d %[abcdefghijklmnopqrstuvwxyz] %*1d %s%n",
                              &fp1, &i , &j, word1, word2, &byte_count);
  printf("\nCount of bytes read = %d\n", byte_count);
  printf("\nCount of values read = %d\n", k);
  printf("\nfp1 = %f   i = %d   j = %d", fp1, i, j);
  printf("\nword1 = %s   word2 = %s\n", word1, word2);
  return 0;
}
```

Here's an example of the output from this program:

```
-2.35 15 25 ready2go

Count of bytes read = 20

Count of values read = 5

fp1 = -2.350000   i = 15   j = 25
word1 = ready   word2 = go
```

How It Works

The first three input values are read in a straightforward way. The fourth input value is read using the specifier "%[abcdefghijklmnopqrstuvwxyz]", which will read a sequence of lowercase letters as a string. This reads the string "ready" because the character that follows, '2', isn't in the set between the square brackets. The '2' in the input is read using the specifier "%*1d". The * in the specification causes the input to be read but not stored, and the field width is one character. The word "go" is read and stored in word2 using the "%s" specifier. The %n specifier does not extract data from the input stream; it just stores the number of bytes that the scanf() function has read from the input stream up to that point so that value is stored in byte_count.

The value of value_count holds the count of the number of values processed that is returned by the scanf() function. As you can see, the value reflects the number of values stored and doesn't include the value read by the "%*1d" specification.

It isn't essential that you enter all the data on a single line. If you key in the first two values and press Enter, the scanf() function will wait for you to enter the next value on the next line.

Now let's change the program a little bit by altering one statement. Replace the input statement with

```
value_count = scanf(
              "%4f %d %d %*d %[abcdefghijklmnopqrstuvwxyz] %*1d %[^o]%n",
              &fp1, &i , &j, word1, word2, &byte_count);
```

Now run the program with exactly the same input line as before. You should get the following output:

```
-2.35 15 25 ready2go

Count of bytes read = 19

Count of values read = 5

fp1 = -2.300000   i = 5   j = 15
word1 = ready   word2 = g
```

Because you specified a field width of 4 for the floating-point value, the first four characters are taken as defining the value of the first input variable. The following integer value to be input is read as 5, the last digit following the value read as –2.3. The integer that's stored in j is 15, and the value 25 is read and ignored by the "%*d" specification. The last string is read now as just "g" because the specification "%[^o]" accepts any character in the string except the letter 'o'. Because the letter 'o' is not read as part of the input the byte count is now 19.

Let's try another variation. Change the input statement to this:

```
value_count = scanf(
              "%4f %4d %d %*d %[abcdefghijklmnopqrstuvwxyz] %*1d %[^o]%n",
              &fp1, &i , &j, word1, word2, &byte_count);
```

With exactly the same input line as before, you should now get the following output:

```
-2.35 15 25 ready2go

Count of bytes read = 19

Count of values read = 5

fp1 = -2.300000   i = 5   j = 15
word1 = ready   word2 = g
```

So what can you conclude from this case? The first floating-point value has clearly been defined by the first four characters of input. The next two values result in the integers 5 and 15. This shows that in spite of the fact that you specified a field width of 4 for the second integer, it appears to have been overridden. This is a consequence of the blank following digit 5, which terminates the input scanning for the value being read. So whatever value you put as a field width, the scanning of the input line for a given value stops as soon as you meet the first blank. You could change the specifiers for the integer values to `%12d` and the result would still be the same for the given input.

You can demonstrate one further aspect of numerical input processing by running the last version of the previous example with a slightly different input line:

```
-2.3A 15 25 ready2go

Count of bytes read = 0

Count of values read = 1

fp1 = -2.300000   i = 0   j = 0
word1 =     word2 =
```

The count of the number of input values is 1, corresponding to a value for the variable `fp1` being read. The count of the number of bytes read is zero, which is clearly incorrect, the reason being that we never got to store a value in `byte_count`. The `A` in the input stream is invalid in numerical input, and so the whole process stops dead. No values for variables `i` and `j`, `word1` and `word2`, are processed, and no value is stored for the byte count. This demonstrates how unforgiving `scanf()` really is. A single invalid character in the input stream will stop your program in its tracks. If you want to be able to recover from the situation in which invalid input is entered, you can use the return value from `scanf()` as a measure of whether all the necessary input has been processed correctly and include some code to retrieve the situation when necessary.

The simplest approach would perhaps be to print an irritable message and then demand the whole input be repeated. But beware of errors in your code getting you into a permanent loop in this circumstance. You'll need to think through all of the possible ways that things might go wrong if you're going to produce a robust program.

You could also read the input using `wscanf()`. The only difference is that you must specify the format string to be a wide character string:

```
value_count = wscanf(L"%f %d %d %[abcdefghijklmnopqrstuvwxyz] %*1d %s%n",
                              &fp1, &i , &j, word1, word2, &byte_count);
```

For this statement to compile, you need an `#include` directive for the `<wchar.h>` header file. The `L` prefix specifies in the first argument the string constant to be a wide character string, so it will occupy twice as much memory as a regular string. You would probably only use `wscanf()` if you were using wide character strings in your program, and in this case you would also be storing the strings read as wide character strings. Here's a fragment that illustrates how you could read strings as wide character strings:

```
wchar_t wword1[SIZE] = L" ";
wchar_t wword2[SIZE] = L" ";
value_count = wscanf(L"%l[abcdefghijklmnopqrstuvwxyz] %*1d %ls%n",
                              wword1, wword2, &byte_count);
printf("\nwword1 = %ls   wword2 = %ls\n", wword1, wword2);
```

Here you have two arrays that store elements of type `wchar_t` to hold the strings. The type specifiers in the format string for the two strings have `l` (lowercase L) as the prefix so the input is read as wide character strings. If you enter **ready2go**, then `ready` and `go` will be stored in `wword1` and `wword2` as wide character strings and the `2` between them will be discarded, as in the example. To output the strings, the `printf()` function uses `%ls` as the format specification, because the function needs to know they are wide character strings. If you were to use `%s`, the output would be incorrect. You could try it to see what you get.

Characters in the Input Format String

You can include a sequence of one or more characters that isn't a format conversion specifier within the input format string. If you do this, you're indicating that you expect the same characters to appear in the input and that the `scanf()` function should read them but not store them. These have to be matched exactly, character for character, by the data in the input stream. Any variation will terminate the input scanning process in `scanf()`.

TRY IT OUT: CHARACTERS IN THE INPUT FORMAT STRING

You can illustrate the effect of including characters in the input format string with the following example:

```
/* Program 10.2 Characters in the format control string */
#include <stdio.h>

int main(void)
{
  int i = 0;
  int j = 0;
  int value_count = 0;
  float fp1 = 0.0;

  printf("Input:\n");
  value_count = scanf("fp1 = %f i = %d %d", &fp1, &i , &j);

  printf("\nOutput:\n");
  printf("\nCount of values read = %d", value_count);
  printf("\nfp1 = %f\ti = %d\tj = %d\n", fp1, i, j);
  return 0;
}
```

Here's an example of the output:

```
Input:
fp1 = 3.14159 i = 7 8

Output:

Count of values read = 3
fp1 = 3.141590  i = 7   j = 8
```

How It Works

It doesn't matter whether the blanks before and after the = are included in the input—they're whitespace characters and are therefore ignored. The important thing is to include the same characters that appear in the format control string in the correct sequence and at the correct place in the input. Try an input line in which this isn't the case:

```
Input:
fp1 = 3.14159 i = 7 j = 8

Output:

Count of values read = 2
fp1 = 3.141590  i = 7   j = 0
```

Now only two values are read. This is because the character `j` in the input stops processing immediately, and no value is received by the variable `j`. The input processing of characters by `scanf()` is also case sensitive. If you input `Fp1=` instead of `fp1=`, no values will be processed at all, because the mismatch with the capital `F` will stop scanning before any values are entered.

Variations on Floating-Point Input

When you're reading floating-point values formatted using `scanf()`, you not only have a choice of specification that you use, but also you can enter the values in a variety of forms. You can see this with a simple example.

TRY IT OUT: FLOATING-POINT INPUT

With this example you can try various forms of specifier and various ways in which you can enter the input values.

```
/* Program 10.3 Floating-Point Input */
#include <stdio.h>

int main(void)
{
  float fp1 = 0.0f;
  float fp2 = 0.0f;
  float fp3 = 0.0f;
  int value_count = 0;

  printf("Input:\n");
  value_count = scanf("%f %f %f", &fp1, &fp2, &fp3);

  printf("\nOutput:\n");
  printf("Return value = %d", value_count);
  printf("\nfp1 = %f  fp2 = %f  fp3 = %f\n", fp1, fp2, fp3);
  return 0;
}
```

Here's an example of output from this program with the same input value written three different ways:

```
Input:
3.14.314E1.0314e+02

Output:
Return value = 3
fp1 = 3.140000  fp2 = 3.140000  fp3 = 3.140000
```

How It Works

This example demonstrates three different ways of entering the same value. The first way is a straightforward decimal value, the second has an exponent value defined by the `E1` that indicates that the value is to be multiplied by 10, and the third has an exponent value of `e+02` and therefore is to be multiplied by 100. As you can see, when you're reading a floating-point value with the `"%f"` specification, you have the option of whether to include an exponent. If you do include an exponent, you can define it beginning with either an `e` or an `E`. You also have the option to include a sign for the exponent value, `+` or `-`, and, of course, the value can be signed too. There are countless variations possible here.

You could try changing the `scanf()` statement to the following:

```
value_count = scanf("%e %g %f", &fp1, &fp2, &fp3);
```

Here's the output with this statement in the program:

```
Input:
3.14.314E1.0314e+02

Output:
Return value = 3
fp1 = 3.140000  fp2 = 3.140000  fp3 = 3.140000
```

Clearly all three format specifications work equally well with the various input forms. The variation between these is only when you use them for output with the `printf()` function.

I recommend that you experiment with the various possibilities here. In particular, try experimenting with floating-point numbers and the field-width specifiers for reading integers.

Reading Hexadecimal and Octal Values

As you saw earlier, you can read hexadecimal values from the input stream using the format specifier `%x`. For octal values you use `%o`. These are very straightforward, but let's see them working in an example.

TRY IT OUT: READING HEXADECIMAL AND OCTAL VALUES

Try the following example:

```
/* Program 10.4 Reading hexadecimal and octal values */
#include <stdio.h>

int main(void)
{
  int i = 0;
```

```
int j = 0;
  int k = 0;
  int n = 0;

  printf("Input:\n");
  n = scanf(" %d %x %o", &i , &j, &k );

  printf("\nOutput:\n");
  printf("%d values read.", n);
  printf("\ni = %d   j = %d   k = %d\n", i, j, k );
  return 0;
}
```

Here's some sample output:

```
Input:
12 12 12

Output:
3 values read.
i = 12   j = 18   k = 10
```

How It Works

You read the three values entered as 12. The first is read with a decimal format specifier `%d`, the second with a hexadecimal format specifier `%x`, and the third with an octal format specifier `%o`. The output shows that 12 in hexadecimal is 18 in decimal notation, whereas 12 in octal is 10 in decimal notation.

Hexadecimal data entry can be useful when you want to enter bit patterns (sequences of 1s and 0s), as they're easier to specify in hexadecimal than in decimal. Each hexadecimal digit corresponds to 4 bits, so you can specify a 16-bit word as four hexadecimal digits. Octal is hardly ever used, and it appears here mainly for historical reasons.

Note the following example of output:

```
Input:
18 18 18

Output:
3 values read.
i = 18   j = 24   k = 1
```

Here, the first two values are read correctly as 18, as a hexadecimal value is indeed 24 in decimal notation. However, the third value is read as 1. This is because 8 isn't a legal octal digit. Octal digits are 0 to 7.

You can enter hexadecimal values using `A` to `F`, or `a` to `f`, or even a mixture if you're so inclined. Here's another example of output:

```
Input:
12 aA 17

Output:
3 values read.
i = 12   j = 170   k = 15
```

The value aA is 10 × 16 + 10, which is 170 as a decimal value. The octal value 17 is 1 × 8 + 7, which is 15 as a decimal value.

There's no difference between using "%x" and "%X" with scanf(), but they'll have a different effect when you use them with printf() for output. You can demonstrate this by changing the last printf() statement to the following:

```
printf("\ni = %x   j = %X   k = %d\n", i, j, k );
```

This now outputs the first two values in hexadecimal notation. You can get the following output with the input shown:

```
Input:
26 AE 77

Output:
3 values read.
i = 1a   j = AE   k = 63
```

So "%x" produces hexadecimal output using hexadecimal digits a to e, and "%X" produces output using hexadecimal digits A to E.

Reading Characters Using scanf()

You tried reading strings in the first example, but there are more possibilities. You know that there are three format specifiers for reading one or more single-byte characters. You can read a single character and store it as type char using the format specifier %c and as type wchar_t using %lc. For a string of characters, you use either the specifier %s or the specifier %[], or if you are storing the input as wide characters, %ls or %l[], where the prefix to the conversion specification is lowercase *L*. In this case, the string is stored as a null-terminated string with '\0' as the last character. With %[] or %l[] format specification, the string to be read must include only the characters that appear between the square brackets, or if the first character between the square brackets is ^, the string must contain only characters that are *not* among those following the ^ characters. Thus, %[aeiou] will read a string that consists only of vowels. The first character that isn't a vowel will signal the end of the string. The specification %[^aeiou] reads a string that contains any character that isn't a vowel. The first vowel will signal the end of the string.

Note that one interesting aspect of the %[] specification is it enables you to read a string containing spaces, something that the %s specification can't do. You just need to include a space as one of the characters between the square brackets.

TRY IT OUT: READING CHARACTERS AND STRINGS

You can see these character-reading capabilities in operation with the following example:

```
/* Program 10.5 Reading characters with scanf() */
#include <stdio.h>

int main(void)
{
  char initial = ' ';
  char name[80] = { 0 };
```

```
 char age[4] = { 0 };
  printf("Enter your first initial: ");
  scanf("%c", &initial );
  printf("Enter your first name: " );
  scanf("%s", name );

  if(initial  != name[0])
    printf("\n%s,you got your initial wrong.", name);
  else
    printf("\nHi, %s. Your initial is correct. Well done!", name );
  printf("\nEnter your full name and your age separated by a comma:\n" );
  scanf("%[^,] , %[0123456789]", name, age );
  printf("\nYour name is %s and you are %s years old\n", name, age );
  return 0;
}
```

Here's some output from this program:

```
Enter your first initial: I
Enter your first name: Ivor

Hi, Ivor. Your initial is correct. Well done!
Enter your full name and your age separated by a comma:
Ivor Horton    ,  99

Your name is
Ivor Horton     and you are 99 years old
```

How It Works

This program first expects you to enter your first initial and then your first name. It checks that the first letter of your name is the same as the initial you entered. This works in a straightforward way, as you can see from the output.

Next, you're asked to enter your full name followed by your age, separated by a comma. The read operation is carried out by the following statement:

```
scanf("%[^,] , %[0123456789]", name, age );
```

I deliberately spaced out the input data so you could see that the first input specification, `%[^,]`, reads any character as part of the string that isn't a comma, including spaces. Hence the extra spaces following the name in the last line of output. You then have a comma in the control string that will cause `scanf()` to read the comma (or several commas in succession) in the input and not store it. The input for `age` is read as a string with the specifier `%[0123456789]`. This will read any sequence of consecutive digits as a string.

Note that the comma in the input string is essential for the input to be read properly. If you leave it out, `scanf()` will attempt to read the comma as part of the input for age. Because a comma is evidently not a digit, this will stop input for `age` so it will just consist of an empty string.

If you try entering a space and then your initial as the first input, the program will treat the blank as the value for `initial` and the single character you entered as your `name`. With the way the control string is defined, the first character that you enter when using the `%c` specifier is taken to be the character, whatever it is. If you don't want a space to be accepted as the initial, you can fix this by writing the input statement as follows:

```
scanf(" %c", &initial );
```

Now the first character in the control string is a space, so `scanf()` will read and ignore any number of spaces and read the first character that isn't a space into `initial`.

Pitfalls with scanf()

There are two very common mistakes people make when using `scanf()` that you should keep in mind:

- Don't forget that the arguments *must* be pointers. Perhaps the most common error is to forget the ampersand (`&`) when specifying single variables as arguments to `scanf()`, particularly because you don't need it with `printf()`. Of course, the `&` isn't necessary if the argument is an array name or a pointer variable.
- When reading a string, remember to ensure that there's enough space for the string to be read in, *plus* the terminating `'\0'`. If you don't do this, you'll overwrite something in memory, possibly even some of your program code.

String Input from the Keyboard

As you've seen, the `gets()` function in `<stdio.h>` will read a complete line of text as a string. The prototype of the function is as follows:

```
char *gets(char *str);
```

This function reads successive characters into the memory pointed to by `str` until you press the Enter key. It appends the terminating null, `'\0'`, in place of the newline character that is read when you press the Enter key. The return value is identical to the argument, which is the address where the string has been stored. The following example provides a reminder of how it works.

TRY IT OUT: READING A STRING WITH GETS()

Here's a simple program using `gets()`:

```
/* Program 10.6 Reading a string with gets() */
#include <stdio.h>

int main(void)
{
  char initial[2] = {0};
  char name[80] = {0};

  printf("Enter your first initial:  ");
  gets(initial);
  printf("Enter your name:  " );
  gets(name);
  if(initial[0] != name[0])
    printf("\n%s,you got your initial wrong.\n", name);
  else
    printf("\nHi, %s. Your initial is correct. Well done!\n", name);
  return 0;
}
```

Here's some output from this program:

```
Enter your first initial:  M
Enter your name:  Mephistopheles

Hi, Mephistopheles. Your initial is correct. Well done!
```

How It Works

You read the initial and the name as strings using `gets()`. The function is very easy to use because there's no format specification involved. Because `gets()` will read characters until you press Enter, you can now enter your full name if you wish.

Of course, the downside to using `gets()` is that you have no control over how many characters are stored. This implies that you must be sure to create an array to receive the data with sufficient space for the maximum length string that might be entered. Where you want to be sure that your array length will not be exceeded, you have the option of using the `fgets()` function. You could replace the statements that manage the input with the following:

```
printf("Enter your first initial: ");
fgets(initial, sizeof(initial), stdin);     /* Read 1 character max       */
fflush(stdin);                              /* Flush the newline          */

printf("Enter your name: " );
fgets(name, sizeof(name), stdin);           /* Read max name-1 characters */
size_t length = strlen(name);
name[length-1] = name[length];              /* Overwrite the newline      */
```

The `fgets()` function will read up to one less than the number of characters specified by the second argument and append the terminating `'\0'`. This ensures that the length of the array you pass as the first argument is not exceeded. You need to remember that the `fgets()` function stores a newline character in the input string corresponding to the Enter key being pressed, whereas the `gets()` function does not. With the first read operation, the newline will be left in the input buffer so the call to `fflush()` will flush `stdin` and remove it. Without this, the newline would be read as the input for `name`. The last character before the null in `name` will be the newline character, so copying the terminating null one position back will overwrite it.

For string input, using `gets()` or `fgets()` is usually the preferred approach unless you want to control the content of the string, in which case you can use `%[]`. The `%[]` specification is more convenient to use when the nonstandard `%[a-z]` form is supported, but remember, because this is nonstandard, your code is no longer as portable as it is if you use the standard form for reading a string of lowercase letters, `%[abcdefghijklmnopqrstuvwxyz]`.

Unformatted Input from the Keyboard

The `getchar()`function reads one character at a time from `stdin`. The `getchar()` function is defined in `<stdio.h>`, and its general syntax is as follows:

```
int getchar(void);
```

The `getchar()` function requires no arguments, and it returns the character read from the input stream. Note that this character is returned as `int`, and the character is displayed on the screen as it is entered from the keyboard.

With many implementations of C, the nonstandard header file `<conio.h>` is often included. This provides additional functions for character input and output. One of the most useful of these is `getch()`, which reads a character from the keyboard without displaying it on the screen. This is

particularly useful when you need to prevent others from being able to see what's being keyed in—for example, when a password is being entered.

The standard header <stdio.h> also declares the ungetc() function that enables you to put a character that you have just read back into an input stream. The function requires two arguments: the first is the character to be pushed back onto the stream, and the second is the identifier for the stream, which would be stdin for the standard input stream. The ungetc() returns a value of type int that corresponds to the character pushed back onto the stream, or a special character, EOF (end-of-file), if the operation fails.

In principle you can push a succession of characters back into an input stream but only one character is guaranteed. As I noted, a failure to push a character back onto a stream will be indicated by EOF being returned by the function, so you should check for this if you are attempting to return several characters to a stream.

The ungetc() function is useful when you are reading input character by character and don't know how many characters make up a data unit. You might be reading an integer value, for example, but don't know how many digits there are. In this situation the ungetc() function makes it possible for you to read a succession of characters using getchar(), and when you find you have read a character that is not a digit, you can return it to the stream. Here's a function that ignores spaces and tabs from the standard input stream using the getchar() and ungetc() functions:

```
void eatspaces(void)
{
  char ch = 0;
  while(isspace(ch = getchar()));  /* Read as long as there are spaces */
  ungetc(ch, stdin);               /* Put back the nonspace character */
}
```

The isspace() function that is declared in the <ctype.h> header file returns true when the argument is a space character. The while loop continues to read characters as long as they are spaces or tabs, storing each character in ch. The first nonspace character that is read will end the loop, and the character will be left in ch. The call to ungetc() returns the nonblank character back to the stream for future processing.

Let's try out the getchar() and ungetc() functions in a working example.

TRY IT OUT: READING AND UNREADING CHARACTERS

This example will assume the input from the keyboard consists of some arbitrary sequence of integers and names:

```
/* Program 10.7 Reading and unreading characters */
#include <stdio.h>
#include <ctype.h>
#include <stdbool.h>
#include <string.h>

const size_t LENGTH = 50;

/* Function prototypes */
void eatspaces(void);
bool getinteger(int *n);
char *getname(char *name, size_t length);
bool isnewline(void);

int main(void)
```

```
{
  int number;
  char name[LENGTH];
  printf("Enter a sequence of integers and alphabetic names:\n");
  while(!isnewline())
    if(getinteger(&number))
      printf("\nInteger value:%8d", number);
    else if(strlen(getname(name, LENGTH)) > 0)
      printf("\nName: %s", name);
    else
    {
      printf("\nInvalid input.");
      return 1;
    }
  return 0;
}

/* Function to check for newline */
bool isnewline(void)
{
  char ch = 0;
  if((ch = getchar()) == '\n')
    return true;

  ungetc(ch, stdin);
  return false;
}

/* Function to ignore spaces from standard input */
void eatspaces(void)
{
  char ch = 0;
  while(isspace(ch = getchar()));
  ungetc(ch, stdin);
}

/* Function to read an integer from standard input */
bool getinteger(int *n)
{
  eatspaces();
  int value = 0;
  int sign = 1;
  char ch = 0;

  /* Check first character */
  if((ch=getchar()) == '-')            /* should be minus */
    sign = -1;
  else if(isdigit(ch))                 /* ...or a digit   */
   value = 10*value + (ch - '0');
  else  if(ch != '+')                  /* ...or plus      */
  {
    ungetc(ch, stdin);
    return false;                      /* Not an integer  */
  }
```

```
  /* Find more digits */
  while(isdigit(ch = getchar()))
    value = 10*value + (ch - '0');

  /* Push back first nondigit character */
  ungetc(ch,stdin);
  *n = value*sign;
  return true;
}

/* Function to read an alphabetic name from input */
char *getname(char *name, size_t length)
{

  eatspaces();                          /* Remove leading spaces */
  size_t count = 0;
  char ch = 0;
  while(isalpha(ch=getchar()))          /* As long as there are letters */
  {
    name[count++] = ch;                 /* store them in name           */
    if(count == length-1)
      break;
  }

  name[count] = '\0';                   /* Append string terminator     */
  if(count < length-1)
    ungetc(ch, stdin);                  /* Return nonletter to stream  */
  return name;
}
```

Here's an example of output from the program:

```
Enter a sequence of integers and alphabetic names:
12              Jack Jim  234 Jo Janet 99  88

Integer value:      12
Name: Jack
Name: Jim
Integer value:     234
Name: Jo
Name: Janet
Integer value:      99
Integer value:      88
```

Here's another sample:

```
Enter a sequence of integers and alphabetic names:
Jim     Jo Will Bert

Name: Jim
Name: Jo
Name: Will
Name: Bert
```

How It Works

There are four functions using the `getchar()` and `ungetc()` functions to read from `stdin`. You saw the `eatspaces()` function in the previous section. The `isnewline()` function just reads a character from the keyboard and returns `true` if it is a newline character. This function is used to control when input ends in `main()`.

The `getinteger()` function reads an integer of arbitrary length from the keyboard that is optionally preceded by a sign. The first step is to remove leading spaces by calling the `eatspaces()` function. After checking for a sign or the first digit, the function continues to read digits from the keyboard in a loop:

```
while(isdigit(ch = getchar()))
  value = 10*value + (ch - '0');
```

The digits are read from left to right, so the latest digit is the low-order digit in the number. The digit value is obtained by subtracting the code value for the 0 digit from the code value for the current digit. This works because the code values for digits are in ascending sequence. To insert the digit, you multiply the current accumulated value by 10 and add the new digit value. Of course, storing the result as type `int` is a constraint. You could implement the function to store the value as type `long long` to accommodate a wider range of values. You could also include code to check for how large the number is getting and to output an error message if it cannot be stored as type `int`.

The first character read that is not a digit ends the loop, and this character is returned to the stream so that it can be read again.

The `getname()` function reads an alphabetic name from the keyboard. The arguments are an array in which the name is to be stored, and the length of the array so the function can ensure the capacity is not exceeded. The function returns the address of the first byte of the string as a convenience to the calling program. The process is, in principle, the same as the `getinteger()` function. The function continues to read characters in a loop as long as they are alphabetic characters:

```
while(isalpha(ch=getchar()))         /* As long as there are letters */
{
  name[count++] = ch;                /* store them in name           */
  if(count == length-1)
    break;
}
```

The `count` variable tracks the number of characters stored in the `name` array, and when only one element is still free, the loop ends. After the loop, the code appends a `'\0'` to terminate the string. Of course, the last character read will have been alphabetic and therefore stored in the array if the value of `count` reaches `length-1`. You therefore only restore the last character back to the stream by calling `ungetc()` when this is not the case.

The `main()` function reads an arbitrary sequence of names and integers in a loop:

```
while(!isnewline())
  if(getinteger(&number))
    printf("\nInteger value:%8d", number);
  else if(strlen(getname(name, LENGTH)) > 0)
    printf("\nName: %s", name);
  else
  {
    printf("\nInvalid input.");
    return 1;
  }
```

The loop continues as long as the current character is not a newline signaling the end of the current line. The program expects to read either an integer or a name on each loop iteration. The loop first tries to read an integer by calling the `getinteger()` function. This function returns `false` if an integer is not found, in which case the `getname()` function is called to read a name. If no name is found, the input is neither a name nor an integer, so the program ends after outputting a message.

Output to the Screen

Writing data to the command line on the screen is much easier than reading input from the keyboard. You know what data you're writing, whereas with input you have all the vagaries of possible incorrect entry of the data. The primary function for formatted output to the `stdout` stream is `printf()`.

Fortunately—or unfortunately, depending how you view the chore of getting familiar with this stuff—`printf()` provides myriad possible variations for the output you can obtain, much more than the scope of the format specifiers associated with `scanf()`.

Formatted Output to the Screen Using printf()

The `printf()` function is defined in the header file `<stdio.h>`, and its general form is the following:

```
int printf(char *format, ...);
```

The first parameter is the format control string. The argument for this parameter is usually passed to the function as an explicit string constant, as you've seen in all the examples, but it can be a pointer to a string that you specify elsewhere. The optional arguments to the function are the values to be output in sequence, and they must correspond in number and type with the format conversion specifiers that appear in the string that is passed as the first argument. Of course, as you've also seen in earlier examples, if the output is simply the text that appears in the control string, there are no additional arguments after the first. But where there are argument values to be output, there must be *at least* as many arguments as there are format specifiers. If not, the results are unpredictable. If there are *more* arguments than specifiers, the excess is ignored. This is because the function uses the format string as the determinant of how many arguments follow and what type they have.

■ **Note** The fact that the format string alone determines how the data is interpreted is the reason why you get the wrong result with a `%d` specifier combined with a `long long` argument.

The `<stdio.h>` header also declares the `wprintf()` function. Analogous to the situation you saw with `scanf()`, the `wprintf()` function works in exactly the same way as `printf()` except that it expects

the first argument to be a wide character string. The format specifications are exactly the same for both functions.

The format conversion specifiers for `printf()` and `wprintf()` are a little more complicated than those you use for input with `scanf()` and `wscanf()`. The general form of an output format specifier is shown in Figure 10-3.

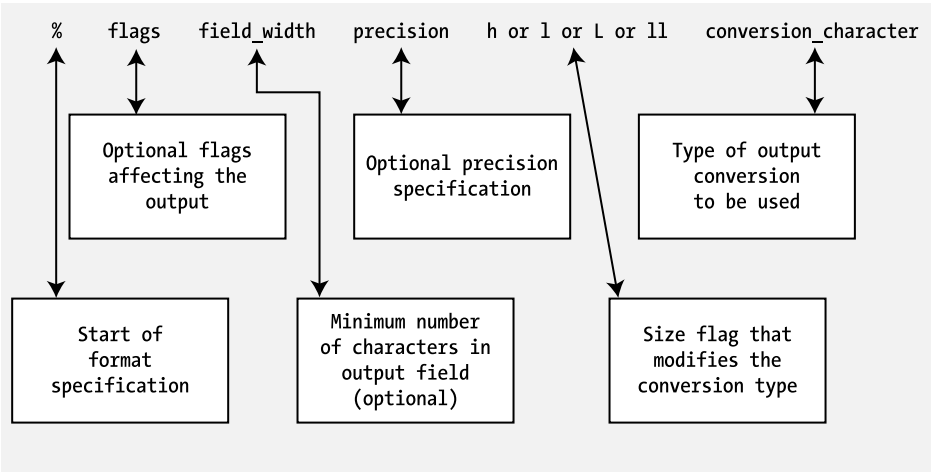

Figure 10-3. *Format specifications for the `printf()` and `wprintf()` functions*

You've seen most of the details before, but let's take a quick pass through the elements of this general format specifier:

- The `%` sign indicates the start of the specifier, as it does for output.
- The optional flag characters are `+`, `-`, `#`, and space. These affect the output, as shown in Table 10-4.

Table 10-4. *Effects of the Optional Flag Characters in an Output Specification*

Character	Use
`+`	Ensures that, for signed output values, there's always a sign preceding the output value—either a plus or a minus sign. By default, only negative values have a sign.
`-`	Specifies that the output value is left-justified in the output field and padded with blanks on the right. The default positioning of the output is right-justified.
`0`	Prefixes a `field_width` value to specify that the value should be padded with zeros to fill out the field width to the left.
`#`	Specifies that `0` is to precede an octal output value, `0x`; or `0X` is to precede a hexadecimal output value; or a floating-point output value will contain a decimal point. For the `g` or `G` floating-point conversion characters, trailing zeros will also be omitted.
space	Specifies that positive or zero output values are preceded by a space rather than a plus sign.

- The optional `field_width` specifies the minimum number of characters for the output value. If the value requires more characters, the field is simply expanded. If it requires less than the minimum specified, it is padded with blanks, unless the field width is specified with a leading zero, as in `09`, for example, where it would be filled on the left with zeros.
- The precision specifier is also optional and is generally used with floating-point output values. A specifier of `.n` indicates that `n` decimal places are to be output. If the value to be output has more than `n` significant digits, it's rounded or truncated.
- You prefix the appropriate type conversion character with the `h`, `l` (lowercase letter *L*), `ll`, or `L` modifier to specify that the output conversion is being applied to `short`, `long`, `long long`, or `long double` values, respectively. The `l` modifier applied to the `c` type specification specifies that the character is to be stored as type `wchar_t`.
- The conversion character that you use defines how the output is to be converted for a particular type of value. Conversion characters are defined in Table 10-5.

Table 10-5. *Conversion Characters in an Output Specification*

Conversion Character	Output Produced
Applicable to integers	
`d`	Signed decimal integer value
`o`	Unsigned octal integer value
`u`	Unsigned decimal integer value
`x`	Unsigned hexadecimal integer value with lowercase hexadecimal digits `a`, `b`, `c`, `d`, `e`, `f`
`X`	As `x` but with uppercase hexadecimal digits `A`, `B`, `C`, `D`, `E`, `F`
Applicable to floating-point	
`f`	Signed decimal value
`e`	Signed decimal value with exponent
`E`	As `e` but with `E` for exponent instead of `e`
`g`	As `e` or `f` depending on size of value and precision
`G`	As `g` but with `E` for exponent values
Applicable to characters	
`c`	Single character
`s`	All characters until `'\0'` is reached or `precision` characters have been output

Believe it or not, this set of output options includes only the most important ones. If you consult the documentation accompanying your compiler, you'll find a few more.

Escape Sequences

You can include whitespace characters in the format control string for `printf()` and `wprintf()`. The characters that are referred to as whitespace are the newline, carriage return, and form-feed characters; blank (a space); and tab. Some of these are represented by escape sequences that begin with `\`. Table 10-6 shows the most common escape sequences.

You use the escape sequence `\\` in format control strings when you want to output the backslash character `\`. If this weren't the case, it would be impossible to output a backslash, because it would always be assumed that a backslash was the start of an escape sequence. To write a `%` character to `stdout`, use `%%`. You can't use `%` by itself as this would be interpreted as the start of a format specification.

Table 10-6. *Common Escape Sequences*

Escape Sequence	Description
\a	Bell sound (a beep on your computer not used much these days)
\b	Backspace
\f	Form-feed or page eject
\n	Newline
\r	Carriage return (for printers) or move to the beginning of the current line for output to the screen
\t	Horizontal tab

■ **Note** Of course, you can use escape sequences within any string, not just in the context of the format string for the `printf()` function.

Integer Output

Let's take a look at some of the variations that you haven't made much use of so far. Those with field width and precision specifiers are probably the most interesting.

TRY IT OUT: OUTPUTTING INTEGERS

Let's try a sample of integer output formats first:

```
/* Program 10.8 Integer output variations */
#include <stdio.h>

int main(void)
{
  int i = 15;
  int j = 345;
  int k = 4567;
  long li = 56789L;
  long lj = 678912L;
  long lk = 23456789L;

  printf("\ni = %d   j = %d    k = %d   i = %6.3d   j = %6.3d   k = %6.3d\n",
                                                    i ,j, k, i, j, k);
  printf("\ni = %-d   j = %+d   k = %-d   i = %-6.3d   j = %-6.3d   k ="
                                       " %-6.3d\n",i ,j, k, i, j, k);
  printf("\nli = %d   lj = %d   lk = %d\n", li, lj, lk);
  printf("\nli = %ld   lj = %ld   lk = %ld\n", li, lj, lk);
  return 0;
}
```

When you execute this example, you should see something like this:

```
i = 15   j = 345    k = 4567   i =    015   j =    345   k =   4567

i = 15   j = +345   k = 4567   i = 015      j = 345      k = 4567

li = -8747   lj = 23552   lk = -5099

li = 56789   lj = 678912   lk = 23456789
```

How It Works

This example illustrates a miscellany of options for integer output. You can see the effects of the - flag by comparing the first two lines produced by these statements:

```
printf("\ni = %d   j = %d    k = %d   i = %6.3d   j = %6.3d   k = %6.3d\n",
                                                    i ,j, k, i, j, k);
printf("\ni = %-d   j = %+d   k = %-d   i = %-6.3d   j = %-6.3d   k ="
                                      " %-6.3d\n",i ,j, k, i, j, k);
```

The - flag causes the output to be left-justified. The effect of the field width specifier is also apparent from the spacing of the last three outputs in each group of six. Note that the default width provides just enough output positions to accommodate the number of digits to be output, so the - flag has no effect.

You get a leading plus in the output of `j` on the second line because of the flag modifier. You can use more than one flag modifier if you want. With the second output of the value of `i`, you have a leading 0 inserted due to the minimum precision being specified as 3. You could also have obtained leading zeroes by preceding the minimum width value with a 0 in the format specification.

The third output line is produced by the following statement:

```
printf("\nli = %d   lj = %d   lk = %d\n", li, lj, lk);
```

Here, you can see that failure to insert the `l` (lowercase letter *L*) modifier when outputting integers of type `long` results in apparent garbage, because the output value is assumed to be a 2-byte integer. Of course, if your system implements type `int` as a 4-byte integer, the values will be correct here. The problem arises only if `long` and `int` are differentiated. You should get a warning from your compiler if there is a mismatch. The same problem can arise when you use the wrong type conversion when outputting values of type `long long`.

You get the correct values from this statement:

```
printf("\nli = %ld   lj = %ld   lk = %ld\n", li, lj, lk);
```

It's unwise to specify inadequate values for the width and the precision of the values to be displayed. Weird and wonderful results may be produced if you do. Try experimenting with this example to see just how much variation you can get.

TRY IT OUT: VARIATIONS ON A SINGLE INTEGER

You'll try one more integer example to run the gamut of possibilities with a single integer value:

```
/* Program 10.9 Variations on a single integer */
#include <stdio.h>

int main(void)
{
  int k = 678;

  printf("%%d   %%o   %%x   %%X");  /* Display format as heading */
  printf("\n%d  %o  %x  %X", k, k, k, k );      /* Display values */

  /* Display format as heading then display the values */
  printf("\n\n%%8d       %%-8d      %%+8d      %%O8d      %%-+8d");
  printf("\n%8d  %-8d  %-+8d  %O8d  %-+8d\n", k, k, k, k, k );
  return 0;
}
```

How It Works

This program may look a little confusing at first because the first of each pair of `printf()` statements displays the format used to output the number appearing immediately below. The `%%` specifier simply outputs the `%` character.

When you execute this example, you should get something like this:

```
%d   %o   %x   %X
678  1246  2a6  2A6

%8d       %-8d      %+8d      %O8d      %-+8d
678       678       +678      00000678  +678
```

The first row of output values is produced by this statement:

```
printf("\n%d  %o  %x  %X", k, k, k, k );      /* Display values */
```

The outputs are decimal, octal, and two varieties of hexadecimal for the value 678, with the default width specification. The corresponding format appears above each value in the output.

The next row of output values is produced by this:

```
printf("\n%8d  %-8d  %-+8d  %O8d  %-+8d\n", k, k, k, k, k );
```

This statement includes a variety of flag settings with a width specification of `8`. The first is the default right-justification in the field. The second is left-justified because of the `-` flag. The third has a sign because of the `+` flag. The fourth has leading zeroes because the width is specified as `O8` instead of `8`, and it also has a sign because of the `+` flag. The last output value uses a specifier with all the trimmings, `%-+8d`, so the output is left-justified in the field and also has a leading sign.

Tip When you're outputting multiple rows of values on the screen, using a width specification—possibly with tabs—will enable you to line them up in columns.

Outputting Floating-Point Values

If plowing through the integer options hasn't started you nodding off, then take a quick look at the floating-point output options through working examples.

TRY IT OUT: OUTPUTTING FLOATING-POINT VALUES

Look at the following example:

```
/* Program 10.10 Outputting floating-point values */
 #include <stdio.h>

int main(void)
{
  float fp1 = 345.678f;
  float fp2 = 1.234E6f;
  double fp3 = 234567898.0;
  double fp4 = 11.22334455e-6;

  printf("\n%f  %+f  %-10.4f  %6.4f\n", fp1, fp2, fp1, fp2);
  printf("\n%e  %+E\n", fp1, fp2);
  printf("\n%f  %g  %#+f  %8.4f  %10.4g\n", fp3,fp3, fp3, fp3, fp4);
  return 0;
}
```

How It Works

With my compiler, I get this output:

```
345.678009      +1234000.000000 345.6780        1234000.0000

3.456780e+002   +1.234000E+006

234567898.000000        2.34568e+008    +234567898.000000       234567898.0000

1.122e-005
```

It's possible that you may not get exactly the same output, but it should be close. Most of the output is a straightforward demonstration of the effects of the format conversion specifiers that I've discussed, but a few points are noteworthy.

The value of the first output for `fp1` differs slightly from the value that you assigned to the variable. This is typical of the kind of small difference that can creep in when floating-point numbers are converted from decimal to binary. With fractional decimal values, there isn't always an exact equivalent in binary floating-point.

In the output generated by the statement

```
printf("\n%f  %+f  %-10.4f  %6.4f\n", fp1, fp2, fp1, fp2);
```

the second output value for `fp1` shows how the number of decimal places after the point can be constrained. The output in this case is left-justified in the field. The second output of `fp2` has a field width specified that is too small for the number of decimal places required and is therefore overridden.

The second `printf()` statement is as follows:

```
printf("\n%e  %+E\n", fp1, fp2);
```

This outputs the same values in floating-point format with an exponent. Whether you get an uppercase `E` or a lowercase `e` for the exponent indicator depends on how you write the format specifier.

In the last line, you can see how the `g`-specified output of `fp3` has been rounded up compared to the `f` specified output.

Note There are a huge number of possible variations for the output obtainable with `printf()`. It would be very educational for you to play around with the options, trying various ways of outputting the same information.

Character Output

Now that you've looked at the various possibilities for outputting numbers, let's have a look at outputting characters. There are basically four flavors of output specifications you can use with `printf()` and `wprintf()` for character data: `%c` and `%s` for single characters and strings, and `%lc` and `%ls` for single wide characters and wide character strings, respectively. You have seen how to use `%s` and `%ls`, so let's just try out single character output in an example.

TRY IT OUT: OUTPUTTING CHARACTER DATA

This example outputs all the printable characters, then all the lower and uppercase letters:

```
/* Program 10.11 Outputting character data */
#include <stdio.h>
#include <limits.h>
#include <wchar.h>
#include <ctype.h>
#include <wctype.h>

int main(void)
{
  int count = 0;
  char ch = 0;
  printf("\nThe printable characters are the following:\n");

  /* Iterate over all values of type char */
  for(int code = 0 ; code <= CHAR_MAX ; code++)
  {
    ch = (char)code;
    if(isprint(ch))
    {
      if(count++ % 32 == 0)
        printf("\n");
      printf("%c", ch);
    }
  }
```

```
  /* Use wprintf() to output wide characters */
  count = 0;
  wchar_t wch = 0;
  wprintf(
      L"\n\nThe alphabetic characters and their codes are the following:\n");

  /* Iterate over the lowercase wide character letters */
  for(wchar_t wch = L'a' ; wch <= L'z' ; wch++)
  {
    if(count++ % 3 == 0)
      wprintf(L"\n");

    wprintf(L" %lc    %#x    %lc    %#x", wch, (long)wch, towupper(wch),
                                                    (long)towupper(wch));
  }
  return 0;
}
```

The output from this program is the following:

```
The printable characters are the following:

 !"#$%&'()*+,-./0123456789:;<=>?
@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_
`abcdefghijklmnopqrstuvwxyz{|}~

The alphabetic characters and their codes are the following:

 a    0x61    A    0x41 b    0x62    B    0x42 c    0x63    C    0x43
 d    0x64    D    0x44 e    0x65    E    0x45 f    0x66    F    0x46
 g    0x67    G    0x47 h    0x68    H    0x48 i    0x69    I    0x49
 j    0x6a    J    0x4a k    0x6b    K    0x4b l    0x6c    L    0x4c
 m    0x6d    M    0x4d n    0x6e    N    0x4e o    0x6f    O    0x4f
 p    0x70    P    0x50 q    0x71    Q    0x51 r    0x72    R    0x52
 s    0x73    S    0x53 t    0x74    T    0x54 u    0x75    U    0x55
 v    0x76    V    0x56 w    0x77    W    0x57 x    0x78    X    0x58
 y    0x79    Y    0x59 z    0x7a    Z    0x5a
```

How It Works

The first block of output for the printable characters is generated in the `for` loop:

```
  for(int code = 0 ; code <= CHAR_MAX ; code++)
  {
    ch = (char)code;
    if(isprint(ch))
    {
      if(count++ % 32 == 0)
        printf("\n");
      printf("%c", ch);
    }
  }
```

First, note the type of the loop control variable, code. You might be tempted to use type char here but this would be a serious mistake, as the loop would run indefinitely. The reason for this is that the condition for ending the loop is checked after the value of code has been incremented. On the last good iteration, code has the value CHAR_MAX, which is the maximum value that can be stored as type char. If code is type char, when 1 is added to CHAR_MAX the result would be 0, so the loop would continue instead of ending as it should.

Within the loop you cast the value of code to type char and store the result in ch. The explicit cast is not required for the code to compile, as the compiler will insert the conversion, but it does indicate that it is intentional. You then use the isprint() function that is declared in the <ctype.h> header to test for a printable character. When isprint() returns true, you output the character using the %c format specification. You also arrange to output a newline character each time 32 characters have been output so that you don't have output just spilling arbitrarily from one line to the next.

The second loop uses the wprintf() function to output alphabetic wide characters and their character code values:

```
for(wchar_t wch = L'a' ; wch <= L'z' ; wch++)
{
  if(count++ % 3 == 0)
    wprintf(L"\n");

  wprintf(L" %lc    %#x    %lc    %#x", wch, (long)wch, towupper(wch),
                                                    (long)towupper(wch));
}
```

This time you can use a loop control variable wch of type wchar_t that iterates over the code values from L'a' to L'z'. You use essentially the same trick as you used in the previous loop to output three success groups of output on each line. The wprintf() outputs the character using the %lc specification and the code value as %ld. To output the code value you cast the value stored in wch to type long and use the %ld format specification to display it. You use the towupper() function that is declared in the <wctype.h> header to get the uppercase equivalent of wch.

Don't forget, the only difference between the wprintf() and printf() functions is that the former requires the first argument to be a wide character string. All of the last output operation could be done just as well with printf(); you would just need to remove the L prefix from the format string that is the first argument.

Other Output Functions

In addition to the string output capabilities of printf() and wprintf(), you have the puts() function that is also declared in <stdio.h>, and which complements the gets() function. The name of the function derives from its purpose: **put** string. The general form of puts() is as follows:

```
int puts(const char *string);
```

The puts() function accepts a pointer to a string as an argument and writes the string to the standard output stream, stdout. The string must be terminated by '\0'. The parameter to puts() is const so the function does not modify the string you pass to it. The function returns a negative integer value if any errors occur on output, and a nonnegative value otherwise. The puts() function is very useful for outputting single-line messages, for example:

```
puts("Is there no end to input and output?");
```

This will output the string that is passed as the argument and then move the cursor to the next line. The function `printf()` requires an explicit `'\n'` to be included at the end of the string to do the same thing.

■ **Note** The function `puts()` will process embedded `'\n'` characters in the string you pass as the argument to generate output on multiple lines.

Unformatted Output to the Screen

Also included in `<stdio.h>`, and complementing the function `getchar()`, is the function `putchar()`. This has the following general form:

```
int putchar(int c);
```

The `putchar()` function outputs a single character, `c`, to `stdout` and returns the character that was displayed. This allows you to output a message one character at a time, which can make your programs a bit bigger, but gives you control over whether or not you output particular characters. For example, you could simply write the following to output a string:

```
char string[] = "Beware the Jabberwock, \nmy son!";
puts(string);
```

Alternatively, you could write:

```
char string[] = " Beware the Jabberwock, \nmy son!";
int i = 0;
while( string[i] != '\0')
  if(string[i] != '\n')
    putchar(string[i++]);
```

The first fragment outputs the string over two lines, like this:

```
Beware the Jabberwock,
my son!
```

The second fragment skips newline characters in the string so the output will be the following:

```
Beware the Jabberwock, my son!
```

Your use of `putchar()` need not be as simple as this. With `putchar()` you could choose to output a selected sequence of characters from the middle of a string bounded by a given delimiter, or you could selectively transform the occurrences of certain characters within a string before output, converting tabs to spaces, for example.

Formatted Output to an Array

You can write formatted data to an array of elements of type `char` in memory using the `sprintf()` function that is declared in the `<stdio.h>` header file. This function has the prototype:

```
int sprintf(char *str, const char *format, . . .);
```

The function writes the data specified by the third and subsequent arguments formatted according to the format string second argument. This works identically to `printf()` except that the data is written to the string specified by the first argument to the function. The integer returned is a count of the number of characters written to `str`, excluding the terminating null character.

Here's a fragment illustrating how this works:

```
char result[20];                /* Output from sprintf */
int count = 4;
int nchars = sprintf(result, "A dog has %d legs.", count);
```

The effect of this is to write the value of `count` to `result` using the format string that is the second argument. The effect is that `result` will contain the string `"A dog has 4 legs."`. The variable `nchars` will contain the value 17, which is the same as the return value you would get from `strlen(result)` after the `sprintf()` call has executed. The `sprintf()` function will return a negative integer if an encoding error occurs during the operation.

One use for the `sprintf()` function is to create format strings programmatically. You'll see a simple example of this in Program12.8 in Chapter 12.

Formatted Input from an Array

The `sscanf()` function complements the `sprintf()` function because it enables you to read data under the control of a format string from an array of elements of type char. The prototype looks like this:

```
int sscanf(const char *str, const char *format, . . .);
```

Data will be read from `str` into variables that are specified by the third and subsequent arguments according to the format string `format`. The function returns a count of the number of items actually read, or `EOF` if a failure occurs before any data values are read and stored. The end of the string is recognized as the end-of-file condition, so reaching the end of the `str` string before any values are converted will cause `EOF` to be returned.

Here's a simple illustration of how the `sscanf()` function works:

```
char *source = "Fred 94";
char name[10];
int age = 0;
int items = sscanf(source, " %s %d", name, age);
```

The result of executing this fragment is that `name` will contain the string `"Fred"` and `age` will contain the value 94. The variable `items` will contain the value 2 because two items are read from `source`.

One use for the `sscanf()` function is to try various ways of reading the same data. You can always read an input line into an array as a string. You can then use `sscanf()` to reread the same input line from the array with different format strings as many times as you like.

Sending Output to the Printer

Writing to a printer is not part of standard C, but some C libraries do support it, so it's worth mentioning. To write output to the default printer, you can use a more generalized form of the `printf()` function called `fprintf()`. This function is designed to send formatted output to *any* stream, and more often to files on disk, but I'll stick to printing for now. For the purpose of printing, the general form for using `fprintf()` is as follows:

```
fprintf(stdprn, format_string, argument1, argument2, ..., argumentn);
```

The function will return a value of type `int` that is a count of the number of values that were sent to `stdprn`. With the exception of the first argument and the extra `f` in the function name, `fprintf()` looks exactly like `printf()`. And so it is. If you don't have `stdprn` defined with your compiler and library, you'll need to consult your documentation to see how to handle printing, but many C libraries do define `stdprn`. You can use the same format string with the same set of specifiers to output data to your printer in exactly the same way that you display results with `printf()`. However, there are a couple of minor variations you need to be aware of. I'll illustrate these in the next example.

TRY IT OUT: PRINTING ON A PRINTER

This program shows how you can get programs to output to a printer:

```
/* Program 10.12 Printing on a printer - where else? */
#include <stdio.h>
int main(void)
{
  fprintf(stdprn, "The barber shaves all those who do not"
                                        " shave themselves.");
  fprintf(stdprn, "\n\rQuestion: Who shaves the barber?\n\r");
  fprintf(stdprn, "\n\rAnswer: She doesn't need to shave.\f");
  return 0;
}
```

How It Works

The only oddities here are the new escape sequences `\r` and `\f`. The sequence `\n\r` is equivalent to newline/carriage return on a printer, and the `\f` is form-feed character, which produces a page eject on printers where this is necessary.

Summary

Although I chose the various specifications for formatting that you've seen in this chapter with the idea of them being as meaningful as possible, there are a lot of them. The only way you're going to become comfortable with them is through practice, and ideally this practice needs to take place in a real-world context. Understanding the various codes is one thing, but they'll probably become really familiar to you only once you've used them a few times in real programs. In the meantime, when you need a quick reminder you can always look them up in the summary Appendix D.

Exercises

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

Exercise 10-1. Write a program that will read, store, and output the following five types of strings on separate lines, when one of each type of string is entered on a single line without spaces between the lines:

- *Type 1*: A sequence of lowercase letters followed by a digit (e.g., `number1`)
- *Type 2*: Two words that both begin with a capital letter and have a hyphen between them (e.g., `Seven-Up`)
- *Type 3*: A decimal value (e.g., `7.35`)
- *Type 4*: A sequence of upper- and lowercase letters and spaces (e.g., `Oliver Hardy`)
- *Type 5*: A sequence of any characters except spaces and digits (e.g., `floating-point`)

The following is a sample of input that should be read as five separate strings:

```
babylon5John-Boy3.14159Stan Laurel'Winner!'
```

Exercise 10-2. Write a program that will read the numerical values in the following line of input, and output the total:

```
  $3.50  ,  $4.75 , $9.95 , $2.50
```

Exercise 10-3. Define a function that will output an array of values of type `double` that is passed as an argument along with the number of elements in the array. The prototype of the function will be the following:

```
void show(double array[], int array_size, int field_width);
```

The values should be output 5 to a line, each with 2 decimal places after the decimal point and in a field width of 12. Use the function in a program that will output the values from 1.5 to 4.5 in steps of 0.3 (i.e., 1.5, 1.8, 2.1, and so on, up to 4.5).

Exercise 10-4. Define a function using the `getchar()` function that will read a string from `stdin` terminated by a character that is passed as the second argument to the function. Thus the prototype will be the following:

```
char *getString(char *buffer, char end_char);
```

The return value is the pointer that is passed as the first argument. Write a program to demonstrate the use of the function to read and output five strings that are from the keyboard, each terminated by a colon.