Chapter 4

Data Input and Output

We have already seen that the C language is accompanied by a collection of library functions, which includes a number of input/output functions. In this chapter we will make use of six of these functions: `getchar`, `putchar`, `scanf`, `printf`, `gets` and `puts`. These six functions permit the transfer of information between the computer and the standard input/output devices (e.g., a keyboard and a TV monitor). The first two functions, `getchar` and `putchar`, allow single characters to be transferred into and out of the computer; `scanf` and `printf` are the most complicated, but they permit the transfer of single characters, numerical values and strings; `gets` and `puts` facilitate the input and output of strings. Once we have learned how to use these functions, we will be able to write a number of complete, though simple, C programs.

4.1 PRELIMINARIES

An input/output function can be accessed from anywhere within a program simply by writing the function name, followed by a list of arguments enclosed in parentheses. The arguments represent data items that are sent to the function. Some input/output functions do not require arguments, though the empty parentheses must still appear.

The names of those functions that return data items may appear within expressions, as though each function reference were an ordinary variable (e.g., `c = getchar();`), or they may be referenced as separate statements (e.g., `scanf( . . . );`). Some functions do not return any data items. Such functions are referenced as though they were separate statements (e.g., `putchar( . . . );`).

Most versions of C include a collection of header files that provide necessary information (e.g., symbolic constants) in support of the various library functions. Each file generally contains information in support of a group of related library functions. These files are entered into the program via an `#include` statement at the beginning of the program. As a rule, the header file required by the standard input/output library functions is called `stdio.h` (see Sec. 8.6 for more information about the contents of these header files).

EXAMPLE 4.1 Here is an outline of a typical C program that makes use of several input/output routines from the standard C library.

```
/* sample setup illustrating the use of input/output library functions */

#include <stdio.h>

main()
{
        char c,d;                       /* declarations */
        float x,y;
        int i,j,k;

        c = getchar();                  /* character input */
        scanf("%f", &x);                /* floating-point input */
        scanf("%d %d", &i, &j);         /* integer input */
        . . .                           /* action statements */
        putchar(d);                     /* character output */
        printf("%3d %7.4f", k, y);      /* numerical output */
}
```

The program begins with the preprocessor statement `#include <stdio.h>`. This statement causes the contents of the header file `stdio.h` to be included within the program. The header file supplies required information to the library functions `scanf` and `printf`. (The syntax of the `#include` statement may vary from one version of C to another; some versions of the language use quotes instead of angle-brackets, e.g., `#include "stdio.h"`.)

Following the preprocessor statement is the program heading `main()` and some variable declarations. Several input/output statements are shown in the skeletal outline that follows the declarations. In particular, the assignment statement `c = getchar();` causes a single character to be entered from the keyboard and assigned to the character variable `c`. The first reference to `scanf` causes a floating-point value to be entered from the keyboard and assigned to the floating-point variable `x`, whereas the second reference to `scanf` causes two decimal integer quantities to be entered from the keyboard and assigned to the integer variables `i` and `j`, respectively.

The output statements behave in a similar manner. Thus, the reference to `putchar` causes the value of the character variable `d` to be displayed. Similarly, the reference to `printf` causes the values of the integer variable `k` and the floating-point variable `y` to be displayed.

The details of each input/output statement will be discussed in subsequent sections of this chapter. For now, you should consider only a general overview of the input/output statements appearing in this typical C program.

4.2 SINGLE CHARACTER INPUT — THE `getchar` FUNCTION

Single characters can be entered into the computer using the C library function `getchar`. We have already encountered the use of this function in Chaps. 1 and 2, and in Example 4.1. Let us now examine it more thoroughly.

The `getchar` function is a part of the standard C I/O library. It returns a single character from a standard input device (typically a keyboard). The function does not require any arguments, though a pair of empty parentheses must follow the word `getchar`.

In general terms, a function reference would be written as

```
character variable = getchar();
```

where *`character variable`* refers to some previously declared character variable.

EXAMPLE 4.2 A C program contains the following statements.

```
char c;
. . . . .
c = getchar();
```

The first statement declares that `c` is a character-type variable. The second statement causes a single character to be entered from the standard input device (usually a keyboard) and then assigned to `c`.

If an *end-of-file* condition is encountered when reading a character with the `getchar` function, the value of the symbolic constant `EOF` will automatically be returned. (This value will be assigned within the `stdio.h` file. Typically, `EOF` will be assigned the value -1, though this may vary from one compiler to another.) The detection of `EOF` in this manner offers a convenient way to detect an end of file, whenever and wherever it may occur. Appropriate corrective action can then be taken. Both the detection of the `EOF` condition and the corrective action can be carried out using the `if - else` statement described in Chap. 6.

The `getchar` function can also be used to read multicharacter strings, by reading one character at a time within a multipass loop. We will see one illustration of this in Example 4.4 below. Additional examples will be presented in later chapters of this book.

4.3 SINGLE CHARACTER OUTPUT — THE `putchar` FUNCTION

Single characters can be displayed (i.e, written out of the computer) using the C library function `putchar`. This function is complementary to the character input function `getchar`, which we discussed in the last section. We have already seen illustrations of the use of these two functions in Chaps. 1 and 2, and in Example 4.1. We now examine the use of `putchar` in more detail.

The `putchar` function, like `getchar`, is a part of the standard C I/O library. It transmits a single character to a standard output device (typically a TV monitor). The character being transmitted will normally be represented as a character-type variable. It must be expressed as an argument to the function, enclosed in parentheses, following the word `putchar`.

In general, a function reference would be written as

`putchar(`*`character variable`*`)`

where *`character variable`* refers to some previously declared character variable.

EXAMPLE 4.3 A C program contains the following statements.

```
char c;
. . . . .
putchar(c);
```

The first statement declares that `c` is a character-type variable. The second statement causes the current value of `c` to be transmitted to the standard output device (e.g., a TV monitor) where it will be displayed. (Compare with Example 4.2, which illustrates the use of the `getchar` function.)

The `putchar` function can be used to output a string constant by storing the string within a one-dimensional, character-type array, as explained in Chap. 2. Each character can then be written separately within a loop. The most convenient way to do this is to utilize a `for` statement, as illustrated in the following example. (The `for` statement is discussed in detail in Chap. 6.)

EXAMPLE 4.4 Lowercase to Uppercase Text Conversion Here is a complete program that reads a line of lowercase text, stores it within a one-dimensional, character-type array, and then displays it in uppercase.

```
/* read in a line of lowercase text and display it in uppercase */

#include <stdio.h>
#include <ctype.h>

main()
{
    char letter[80];
    int count, tag;

    /* enter the text */

    for (count = 0; (letter[count] = getchar()) != '\n'; ++count)
        ;

    /* tag the character count */

    tag = count;

    /* display the line in uppercase */

    for (count = 0; count < tag; ++count)
        putchar(toupper(letter[count]));
}
```

Notice the declaration

```
char letter[80];
```

This declares `letter` to be an 80-element, character-type array whose elements will represent the individual characters within the line of text.

Now consider the statement

```
for (count = 0; (letter[count] = getchar()) != '\n'; ++count)
    ;
```

This statement creates a loop that causes the individual characters to be read into the computer and assigned to the array elements. The loop begins with a value of `count` equal to zero. A character is then read into the computer from the standard input device, and assigned to `letter[0]` (the first element in `letter`). The value of `count` is then incremented, and the process is repeated for the next array element. This looping action continues as long as a *newline* character (i.e., `'\n'`) is not encountered. The *newline* character will signify the end of the line, and will therefore terminate the process.

Once all of the characters have been entered, the value of `count` corresponding to the last character is assigned to `tag`. Another `for` loop is then initiated, in which the uppercase equivalents of the original characters are displayed on the standard output device. Characters that were originally uppercase, digits, punctuation characters, etc., will be displayed in their original form. Thus, if the message

```
Now is the time for all good men to come to the aid of their country!
```

is entered as input, the corresponding output will be

```
NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUNTRY!
```

Note that `tag` will be assigned the value 69 after all of the characters have been entered, since the 69th character will be the newline character following the exclamation point.

Chapter 6 contains more detailed information on the use of the `for` statement to control a character array. For now, you should seek only a general understanding of what is happening.

4.4 ENTERING INPUT DATA — THE `scanf` FUNCTION

Input data can be entered into the computer from a standard input device by means of the C library function `scanf`. This function can be used to enter any combination of numerical values, single characters and strings. The function returns the number of data items that have been entered successfully.

In general terms, the `scanf` function is written as

```
scanf(control string, arg1, arg2, . . . , argn)
```

where *control string* refers to a string containing certain required formatting information, and *arg1*, *arg2*, . . . , *argn* are arguments that represent the individual input data items. (Actually, the arguments represent *pointers* that indicate the *addresses* of the data items within the computer's memory. More about this later, in Chap. 10.)

The control string consists of individual groups of characters, with one character group for each input data item. Each character group must begin with a percent sign (%). In its simplest form, a single character group will consist of the percent sign, followed by a *conversion character* which indicates the type of the corresponding data item.

Within the control string, multiple character groups can be contiguous, or they can be separated by whitespace characters (i.e., blank spaces, tabs or newline characters). If whitespace characters are used to separate multiple character groups in the control string, then all consecutive whitespace characters in the input data will be read but ignored. The use of blank spaces as character-group separators is very common.

The more frequently used conversion characters are listed in Table 4-1.

Table 4-1 Commonly Used Conversion Characters for Data Input

Conversion Character	*Meaning*
c	data item is a single character
d	data item is a decimal integer
e	data item is a floating-point value
f	data item is a floating-point value
g	data item is a floating-point value
h	data item is a short integer
i	data item is a decimal, hexadecimal or octal integer
o	data item is an octal integer
s	data item is a string followed by a whitespace character (the null character `\0` will automatically be added at the end)
u	data item is an unsigned decimal integer
x	data item is a hexadecimal integer
[. . .]	data item is a string which may include whitespace characters (see explanation below)

The arguments are written as variables or arrays, whose types match the corresponding character groups in the control string. *Each variable name must be preceded by an ampersand* (&). (The arguments are actually pointers that indicate where the data items are stored in the computer's memory, as explained in Chap. 10.) However, array names should *not* begin with an ampersand.

EXAMPLE 4.5 Here is a typical application of a `scanf` function.

```
#include <stdio.h>

main()

{
    char item[20];
    int partno;
    float cost;

    . . . . .

    scanf("%s %d %f", item, &partno, &cost);

    . . . . .
}
```

Within the `scanf` function, the control string is `"%s %d %f"`. It contains three character groups. The first character group, `%s`, indicates that the first argument (`item`) represents a string. The second character group, `%d`, indicates that the second argument (`&partno`) represents a decimal integer value, and the third character group, `%f`, indicates that the third argument (`&cost`) represents a floating-point value.

Notice that the numerical variables `partno` and `cost` are preceded by ampersands within the `scanf` function. An ampersand does not precede `item`, however, since `item` is an array name.

Notice also that the `scanf` function could have been written

```
scanf("%s%d%f", item, &partno, &cost);
```

with no whitespace characters in the control string. This is also valid, though the input data could be interpreted differently when using c-type conversions (more about this later in this chapter).

The actual data items are numeric values, single characters or strings, or some combination thereof. They are entered from a standard input device (typically a keyboard). The data items must correspond to the arguments in the scanf function in number, in type and in order. Numeric data items are written in the same form as numeric constants (see Sec. 2.4), though octal values need not be preceded by a 0, and hexadecimal values need not be preceded by 0x or 0X. Floating-point values must include either a decimal point or an exponent (or both).

If two or more data items are entered, they must be separated by whitespace characters. (A possible exception to this rule occurs with c-type conversions, as described in Sec. 4.5) The data items may continue onto two or more lines, since the newline character is considered to be a whitespace character and can therefore separate consecutive data items.

Moreover, if the control string begins by reading a character-type data item, it is generally a good idea to precede the first conversion character with a blank space. This causes the scanf function to ignore any extraneous characters that may have been entered earlier (for example, by pressing the Enter key after entering a previous line of data).

EXAMPLE 4.6 Consider once again the skeletal outline of a C program shown in Example 4.5; i.e.,

```
#include <stdio.h>

main()

{
    char item[20];
    int partno;
    float cost;

    . . . . .

    scanf(" %s %d %f", item, &partno, &cost);

    . . . . .
}
```

Notice the blank space that precedes %s. This prevents any previously entered extraneous characters from being assigned to item.

The following data items could be entered from the standard input device when the program is executed.

```
fastener 12345 0.05
```

Thus, the characters that make up the string fastener would be assigned to the first eight elements of the array item; the integer value 12345 would be assigned to partno, and the floating-point value 0.05 would be assigned to cost.

Note that the individual data items are entered on one line, separated by blank spaces. The data items could also be entered on separate lines, however, since newline characters are also whitespace characters. Therefore, the data items could also be entered in any of the following ways:

```
fastener            fastener             fastener   12345
12345               12345    0.05        0.05
0.05
```

Note that the s-type conversion character applies to a string that is terminated by a whitespace character. Therefore, a string that *includes* whitespace characters cannot be entered in this manner. There are ways, however, to work with strings that include whitespace characters. One way is to use the `getchar` function within a loop, as illustrated in Example 4.4. It is also possible to use the `scanf` function to enter such strings. To do so, the s-type conversion character within the control string is replaced by a sequence of characters enclosed in square brackets, designated as `[. . .]`. Whitespace characters may be included within the brackets, thus accommodating strings that contain such characters.

When the program is executed, successive characters will continue to be read from the standard input device as long as each input character matches one of the characters enclosed within the brackets. The order of the characters within the square brackets need not correspond to the order of the characters being entered. Input characters may be repeated. The string will terminate, however, once an input character is encountered that does not match any of the characters within the brackets. A null character (`\0`) will then automatically be added to the end of the string.

EXAMPLE 4.7 This example illustrates the use of the `scanf` function to enter a string consisting of uppercase letters and blank spaces. The string will be of undetermined length, but it will be limited to 79 characters (actually, 80 characters including the null character that is added at the end). Notice the blank space that precedes the % sign.

```
#include <stdio.h>

main()

{
    char line[80];

    . . . . .

    scanf(" %[ ABCDEFGHIJKLMNOPQRSTUVWXYZ]", line);

    . . . . .

}
```

If the string

```
NEW YORK CITY
```

is entered from the standard input device when the program is executed, the entire string will be assigned to the array `line` since the string is comprised entirely of uppercase letters and blank spaces. If the string were written as

```
New York City
```

however, then only the single letter `N` would be assigned to `line`, since the first lowercase letter (in this case, `e`) would be interpreted as the first character beyond the string. It would, of course, be possible to include both uppercase and lowercase characters within the brackets, but this becomes cumbersome.

A variation of this feature which is often more useful is to precede the characters within the square brackets by a *circumflex* (i.e., ^). This causes the subsequent characters within the brackets to be interpreted in the opposite manner. Thus, when the program is executed, successive characters will continue to be read from the standard input device as long as each input character *does not* match one of the characters enclosed within the brackets.

If the characters within the brackets are simply the circumflex followed by a newline character, then the string entered from the standard input device can contain any ASCII characters except the newline character (line feed). Thus, the user may enter whatever he or she wishes and then press the `Enter` key. The `Enter` key will issue the newline character, thus signifying the end of the string.

EXAMPLE 4.8 Suppose a C program contains the following statements.

```
#include <stdio.h>

main()
{
    char line[80];

    . . . . .

    scanf(" %[^\n]", line);

    . . . . .
}
```

Notice the blank space preceding `%[^\n]`, to ignore any unwanted characters that may have been entered previously.

When the `scanf` function is executed, a string of undetermined length (but not more than 79 characters) will be entered from the standard input device and assigned to `line`. There will be no restrictions on the characters that comprise the string, except that they all fit on one line. For example, the string

```
The PITTSBURGH STEELERS is one of America's favorite football teams!
```

could be entered from the keyboard and assigned to `line`.

4.5 MORE ABOUT THE scanf FUNCTION

This section contains some additional details about the `scanf` function. Beginning C programmers may wish to skip over this material for the time being.

The consecutive nonwhitespace characters that define a data item collectively define a *field*. It is possible to limit the number of such characters by specifying a maximum *field width* for that data item. To do so, an unsigned integer indicating the field width is placed within the control string, between the percent sign (%) and the conversion character.

The data item may contain fewer characters than the specified field width. However, the number of characters in the actual data item cannot exceed the specified field width. Any characters that extend beyond the specified field width will not be read. Such leftover characters may be incorrectly interpreted as the components of the next data item.

EXAMPLE 4.9 The skeletal structure of a C program is shown below.

```
#include <stdio.h>

main()
{
    int a, b, c;

    . . . . .

    scanf("%3d %3d %3d", &a, &b, &c);

    . . . . .
}
```

When the program is executed, three integer quantities will be entered from the standard input device (the keyboard). Suppose the input data items are entered as

```
1 2 3
```

Then the following assignments will result:

```
a = 1,   b = 2,   c = 3
```

If the data had been entered as

```
123 456 789
```

Then the assignments would be

```
a = 123,      b = 456,   c = 789
```

Now suppose that the data had been entered as

```
123456789
```

Then the assignments would be

```
a = 123,      b = 456,   c = 789
```

as before, since the first three digits would be assigned to a, the next three digits to b, and the last three digits to c.

Finally, suppose that the data had been entered as

```
1234 5678 9
```

The resulting assignments would now be

```
a = 123,      b = 4,   c = 567
```

The remaining two digits (8 and 9) would be ignored, unless they were read by a subsequent scanf statement.

EXAMPLE 4.10 Consider a C program that contains the following statements.

```
#include <stdio.h>

main()

{
    int i;
    float x;
    char c;

    . . . . .

    scanf("%3d %5f %c", &i, &x, &c);

    . . . . .
}
```

If the data items are entered as

```
10 256.875 T
```

when the program is executed, then 10 will be assigned to i, 256.8 will be assigned to x and the character 7 will be assigned to c. The remaining two input characters (5 and T) will be ignored.

Most versions of C allow certain conversion characters within the control string to be preceded by a single-letter *prefix*, which indicates the length of the corresponding argument. For example, an l (lowercase L) is used to indicate either a signed or unsigned long integer argument, or a double-precision argument. Similarly, an h is used to indicate a signed or unsigned short integer. Also, some versions of of C permit the use of an uppercase L to indicate a long double.

EXAMPLE 4.11 Suppose the following statements are included in a C program.

```
#include <stdio.h>

main()

{
    short ix,iy;
    long lx,ly;
    double dx,dy;

    . . . . .

    scanf("%hd %ld %lf", &ix, &lx, &dx);

    . . . . .

    scanf("%3ho %7lx %15le", &iy, &ly, &dy);

    . . . . .
}
```

The control string in the first scanf function indicates that the first data item will be assigned to a short decimal integer variable, the second will be assigned to a long decimal integer variable, and the third will be assigned to a double-precision variable. The control string in the second scanf function indicates that the first data item will have a maximum field width of 3 characters and it will be assigned to a short octal integer variable, the second data item will have a maximum field width of 7 characters and it will be assigned to a long hexadecimal integer variable, and the third data item will have a maximum field width of 15 characters and it will be assigned to a double-precision variable.

Some versions of C permit the use of uppercase conversion characters to indicate long integers (signed or unsigned). This feature may be available in addition to the prefix "l", or it may replace the use of the prefix.

EXAMPLE 4.12 Consider once again the skeletal outline of the C program given in Example 4.11. With some versions of C, it may be possible to write the scanf functions somewhat differently, as follows.

```
#include <stdio.h>

main()

{
    short ix,iy;
    long lx,ly;
    double dx,dy;

    . . . . .

    scanf("%hd %D %f", &ix, &lx, &dx);

    . . . . .

    scanf("%3ho %7X %15e", &iy, &ly, &dy);

    . . . . .
}
```

Notice the use of uppercase conversion characters (in the `scanf` functions) to indicate long integers. The interpretation of the `scanf` functions will be the same as in the previous example.

In most versions of C it is possible to skip over a data item, without assigning it to the designated variable or array. To do so, the `%` sign within the appropriate control group is followed by an asterisk (`*`). This feature is referred to as *assignment suppression.*

EXAMPLE 4.13 Here is a variation of the `scanf` features shown in Example 4.6.

```
#include <stdio.h>

main()

{
    char item[20];
    int partno;
    float cost;

    . . . . .

    scanf(" %s %*d %f", item, &partno, &cost);

    . . . . .
}
```

Notice the asterisk in the second character group.

If the corresponding data items are

```
fastener 12345 0.05
```

then `fastener` will be assigned to `item` and `0.05` will be assigned to `cost`. However `12345` will not be assigned to `partno` because of the asterisk, which is interpreted as an assignment suppression character.

Note that the integer quantity `12345` will be read into the computer along with the other data items, even though it is not assigned to its corresponding variable.

If the control string contains multiple character groups without interspersed whitespace characters, then some care must be taken with c-type conversion. In such cases a whitespace character within the input data will be interpreted as a data item. To skip over such whitespace characters and read the next nonwhitespace character, the conversion group `%1s` should be used.

EXAMPLE 4.14 Consider the following skeletal outline of a C program.

```
#include <stdio.h>

main()

{
    char c1,c2,c3;

    . . . . .

    scanf(" %c%c%c", &c1, &c2, &c3);

    . . . . .
}
```

If the input data consisted of

```
a b c
```

(with blank spaces between the letters), then the following assignments would result:

```
c1 = a,          c2 = <blank space>,          c3 = b
```

If the scanf function were written as

```
scanf(" %c%1s%1s", &c1, &c2, &c3)
```

however, then the same input data would result in the following assignments:

```
c1 = a,          c2 = b,          c3 = c
```

as intended.

Note that there are some other ways around this problem. We could have written the scanf function as

```
scanf(" %c %c %c", &c1, &c2, &c3);
```

with blank spaces separating the %c terms, or we could have used the original scanf function but written the input data as consecutive characters without blanks; i.e., abc.

Unrecognized characters within the control string are expected to be matched by the same characters in the input data. Such input characters will be read into the computer, but not assigned to an identifier. Execution of the scanf function will terminate if a match is not found.

EXAMPLE 4.15 Consider the following skeletal outline.

```
#include <stdio.h>

main()

{
    int i;
    float x;

    . . . . .

    scanf("%d a %f", &i, &x);

    . . . . .
}
```

If the input data consist of

```
1 a 2.0
```

then the decimal integer 1 will be read in and assigned to i, the character a will be read in but subsequently ignored, and the floating-point value 2.0 will be read in and assigned to x.

On the other hand, if the input were entered simply as

```
1 2.0
```

then the scanf function would stop executing once the expected character (a) is not found. Therefore, i would be assigned the value 1 but x would automatically represent the value 0.

You should understand that there is some variation in the features supported by the `scanf` function from one version of C to another. The features described above are quite common and are available in virtually all versions of the language. However, there may be slight differences in their implementation. Moreover, additional features may be available in some versions of the language.

4.6 WRITING OUTPUT DATA — THE `printf` FUNCTION

Output data can be written from the computer onto a standard output device using the library function `printf`. This function can be used to output any combination of numerical values, single characters and strings. It is similar to the input function `scanf`, except that its purpose is to display data rather than to enter it into the computer. That is, the `printf` function moves data from the computer's memory to the standard output device, whereas the `scanf` function enters data from the standard input device and stores it in the computer's memory.

In general terms, the `printf` function is written as

```
printf(control string, arg1, arg2, . . . , argn)
```

where *`control string`* refers to a string that contains formatting information, and *`arg1, arg2, . . . , argn`* are arguments that represent the individual output data items. The arguments can be written as constants, single variable or array names, or more complex expressions. Function references may also be included. In contrast to the `scanf` function discussed in the last section, the arguments in a `printf` function do *not* represent memory addresses and therefore are *not* preceded by ampersands.

The control string consists of individual groups of characters, with one character group for each output data item. Each character group must begin with a percent sign (%). In its simplest form, an individual character group will consist of the percent sign, followed by a *conversion character* indicating the type of the corresponding data item.

Multiple character groups can be contiguous, or they can be separated by other characters, including whitespace characters. These "other" characters are simply transferred directly to the output device, where they are displayed. The use of blank spaces as character-group separators is particularly common.

Several of the more frequently used conversion characters are listed in Table 4-2.

Table 4-2 Commonly Used Conversion Characters for Data Output

Conversion Character	*Meaning*
c	Data item is displayed as a single character
d	Data item is displayed as a signed decimal integer
e	Data item is displayed as a floating-point value with an exponent
f	Data item is displayed as a floating-point value without an exponent
g	Data item is displayed as a floating-point value using either e-type or f-type conversion, depending on value. Trailing zeros and trailing decimal point will not be displayed.
i	Data item is displayed as a signed decimal integer
o	Data item is displayed as an octal integer, without a leading zero
s	Data item is displayed as a string
u	Data item is displayed as an unsigned decimal integer
x	Data item is displayed as a hexadecimal integer, without the leading 0x

Note that some of these characters are interpreted differently than with the `scanf` funtion (see Table 4-1).

EXAMPLE 4.16 Here is a simple program that makes use of the `printf` function.

```
#include <stdio.h>
#include <math.h>

main()   /* print several floating-point numbers */
{
    float i = 2.0, j = 3.0;
    printf("%f %f %f %f", i, j, i+j, sqrt(i+j));
}
```

Notice that the first two arguments within the `printf` function are single variables, the third argument is an arithmetic expression, and the last argument is a function reference that has a numeric expression as an argument.

Executing the program produces the following output:

```
2.000000 3.000000 5.000000 2.236068
```

EXAMPLE 4.17 The following skeletal outline indicates how several different types of data can be displayed using the `printf` function.

```
#include <stdio.h>

main()
{
    char item[20];
    int partno;
    float cost;

    . . . . .

    printf("%s %d %f", item, partno, cost);

    . . . . .

}
```

Within the `printf` function, the control string is `"%s %d %f"`. It contains three character groups. The first character group, `%s`, indicates that the first argument (`item`) represents a string. The second character group, `%d`, indicates that the second argument (`partno`) represents a decimal integer value, and the third character group, `%f`, indicates that the third argument (`cost`) represents a floating-point value.

Notice that the arguments are not preceded by ampersands. This differs from the `scanf` function, which requires ampersands for all arguments other than array names (see Example 4.5).

Now suppose that `name`, `partno` and `cost` have been assigned the values `fastener`, `12345` and `0.05`, respectively, within the program. When the `printf` statement is executed, the following output will be generated.

```
fastener 12345 0.050000
```

The single space between data items is generated by the blank spaces that appear within the control string in the `printf` statement.

Suppose the `printf` statement had been written as

```
printf("%s%d%f", item, partno, cost);
```

This `printf` statement is syntactically valid, though it causes the output items to run together; i.e.,

```
fastener123450.050000
```

The `f`-type conversion and the `e`-type conversion are both used to output floating-point values. However, the latter causes an exponent to be included in the output, whereas the former does not.

EXAMPLE 4.18 The following program generates the same floating-point output in two different forms.

```
#include <stdio.h>

main()    /* display floating-point output 2 different ways */
{
    double x = 5000.0, y = 0.0025;

    printf("%f %f %f %f\n\n", x, y, x*y, x/y);
    printf("%e %e %e %e", x, y, x*y, x/y);
}
```

Both printf statements have the same arguments. However, the first printf statement makes use of f-type conversion, whereas the second printf statement uses e-type conversion. Also, notice the repeated newline character in the first printf statement. This causes the output to be double-spaced, as shown below.

When the program is executed, the following output is generated.

```
5000.000000 0.002500 12.500000 2000000.000000

5.000000e+03 2.500000e-03 1.250000e+01 2.000000e+06
```

The first line of output shows the quantities represented by x, y, x*y and x/y in standard floating-point format, without exponents. The second line of output shows these same quantities in a form resembling scientific notation, with exponents.

Notice that six decimal places are shown for each value. The number of decimal places can be altered, however, by specifying the *precision* as a part of each character group within the control string (more about this in Sec. 4.7).

The printf function interprets s-type conversion differently than the scanf function. In the printf function, s-type conversion is used to output a string that is terminated by the null character (\0). Whitespace characters may be included within the string.

EXAMPLE 4.19 Reading and Writing a Line of Text Here is a short C program that will read in a line of text and then write it back out, just as it was entered. The program illustrates the syntactic differences in reading and writing a string that contains a variety of characters, including whitespace characters.

```
#include <stdio.h>

main()        /* read and write a line of text */
{
    char line[80];

    scanf(" %[^\n]", line);
    printf("%s", line);
}
```

Notice the difference in the control strings within the scanf function and the printf function.

Now suppose that the following string is entered from the standard input device when the program is executed.

```
The PITTSBURGH STEELERS is one of America's favorite football teams!
```

This string contains lowercase characters, uppercase characters, punctuation characters and whitespace characters. The entire string can be entered with the single scanf function, as long as it is terminated by a newline character (by pressing the Enter key). The printf function will then cause the entire string to be displayed on the standard output device, just as it had been entered. Thus, the message

```
The PITTSBURGH STEELERS is one of America's favorite football teams!
```

would be generated by the computer.

A *minimum* field width can be specified by preceding the conversion character by an unsigned integer. If the number of characters in the corresponding data item is less than the specified field width, then the data item will be preceded by enough leading blanks to fill the specified field. If the number of characters in the data item exceeds the specified field width, however, then additional space will be allocated to the data item, so that the entire data item will be displayed. This is just the opposite of the field width indicator in the `scanf` function, which specifies a *maximum* field width.

EXAMPLE 4.20 The following C program illustrates the use of the minimum field width feature.

```
#include <stdio.h>

main()       /* minimum field width specifications */

{
    int i = 12345;
    float x = 345.678;

    printf("%3d %5d %8d\n\n", i, i, i);
    printf("%3f %10f %13f\n\n", x, x, x);
    printf("%3e %13e %16e", x, x, x);
}
```

Notice the double newline characters in the first two `printf` statements. They will cause the lines of output to be double spaced, as shown below.

When the program is executed, the following output is generated.

```
12345 12345    12345

345.678000 345.678000    345.678000

3.456780e+02  3.456780e+02     3.456780e+02
```

The first line of output displays a decimal integer using three different minimum field widths (three characters, five characters and eight characters). The entire integer value is displayed within each field, even if the field width is too small (as with the first field in this example).

The second value in the first line is preceded by one blank space. This is generated by the blank space separating the first two character groups within the control string.

The third value is preceded by four blank spaces. One blank space comes from the blank space separating the last two character groups within the control field. The other three blank spaces fill the minimum field width, which exceeds the number of characters in the output value (the minimum field width is eight, but only five characters are displayed).

A similar situation is seen in the next two lines, where the floating-point value is displayed using `f`-type conversion (in line 2) and e-type conversion (line 3).

EXAMPLE 4.21 Here is a variation of the program presented in Example 4.20, which makes use of g-type conversion.

```
#include <stdio.h>

main()       /* minimum field width specifications */

{
    int i = 12345;
    float x = 345.678;

    printf("%3d %5d %8d\n\n", i, i, i);
    printf("%3g %10g %13g\n\n", x, x, x);
    printf("%3g %13g %16g", x, x, x);
}
```

Execution of this program causes the following output to be displayed.

```
12345 12345     12345

345.678     345.678         345.678

345.678         345.678             345.678
```

The floating-point values are displayed with an `f`-type conversion, since this results in a shorter display. The minimum field widths conform to the specifications within the control string.

4.7 MORE ABOUT THE `printf` FUNCTION

This section contains additional details about the `printf` function. Beginning C programmers may wish to skip over this material for the time being.

We have already learned how to specify a minimum field width in a `printf` function. It is also possible to specify the maximum number of decimal places for a floating-point value, or the maximum number of characters for a string. This specification is known as *precision*. The precision is an unsigned integer that is always preceded by a decimal point. If a minimum field width is specified in addition to the precision (as is usually the case), then the precision specification follows the field width specification. Both of these integer specifications precede the conversion character.

A floating-point number will be *rounded* if it must be shortened to conform to a precision specification.

EXAMPLE 4.22 Here is a program that illustrates the use of the precision feature with floating-point numbers.

```
#include <stdio.h>

main()   /* display a floating-point number with several different precisions */

{
    float x = 123.456;

    printf("%7f %7.3f %7.1f\n\n", x, x, x);
    printf("%12e %12.5e %12.3e", x, x, x);
}
```

When this program is executed, the following output is generated.

```
123.456000 123.456   123.5

1.234560e+02  1.23456e+02     1.235e+02
```

The first line is produced by `f`-type conversion. Notice the rounding that occurs in the third number because of the precision specification (one decimal place). Also, notice the leading blanks that are added to fill the specified minimum field width (seven characters).

The second line, produced by `e`-type conversion, has similar characteristics. Again, we see that the third number is rounded to conform to the specified precision (three decimal places). Also, note the leading blanks that are added to fill the specified minimum field width (12 characters).

A minimum field width specification need not necessarily accompany the precision specification. It is possible to specify the precision without the minimum field width, though the precision must still be preceded by a decimal point.

EXAMPLE 4.23 Now let us rewrite the program shown in the last example without any minimum field width specifications, but with precision specifications.

```
#include <stdio.h>

main()   /* display a floating-point number with several different precisions */

{
    float x = 123.456;

    printf("%f %.3f %.1f\n\n", x, x, x);
    printf("%e %.5e %.3e", x, x, x);
}
```

Execution of this program produces the following output.

```
123.456000 123.456 123.5

1.234560e+02 1.23456e+02 1.235e+02
```

Notice that the third number in each line does not have multiple leading blanks, since there is no minimum field width that must be satisfied. In all other respects, however, this output is the same as the output generated in the last example.

Minimum field width and precision specifications can be applied to character data as well as numerical data. When applied to a string, the minimum field width is interpreted in the same manner as with a numerical quantity; i.e., leading blanks will be added if the string is shorter than the specified field width, and additional space will be allocated if the string is longer than the specified field width. Hence, the field width specification will not prevent the entire string from being displayed.

However, the precision specification will determine the maximum number of characters that can be displayed. If the precision specification is less than the total number of characters in the string, the excess right-most characters will not be displayed. This will occur even if the minimum field width is larger than the entire string, resulting in the addition of leading blanks to the truncated string.

EXAMPLE 4.24 The following program outline illustrates the use of field width and precision specifications in conjunction with string output.

```
#include <stdio.h>

main()

{
    char line[12];

    . . . . .

    printf("%10s %15s %15.5s %.5s", line, line, line, line);
}
```

Now suppose that the string `hexadecimal` is assigned to the character array `line`. When the program is executed, the following output will be generated.

```
hexadecimal     hexadecimal           hexad hexad
```

The first string is shown in its entirety, even though this string consists of 11 characters but the field width specification is only 10 characters. Thus, the first string overrides the minimum field width specification. The second string is padded with four leading blanks to fill out the 15-character minimum; hence, the second string is *right justified* within its field. The third string consists of only five nonblank characters because of the five-character precision specification; however, 10 leading blanks are added to fill out the minimum field width specification, which is 15 characters. The last string also consists of five nonblank characters. Leading blanks are not added, however, because there is no minimum field width specification.

Most versions of C permit the use of prefixes within the control string to indicate the length of the corresponding argument. The allowable prefixes are the same as the prefixes used with the `scanf` function. Thus, an `l` (lowercase) indicates a signed or unsigned integer argument, or a double-precision argument; an `h` indicates a signed or unsigned short integer. Some versions of C permit an `L` (uppercase) to indicate a long double.

EXAMPLE 4.25 Suppose the following statements are included in a C program.

```
#include <stdio.h>

main ()

{
    short a, b;
    long c, d;

    . . . . .

    printf("%5hd %6hx %8lo %lu", a, b, c, d);

    . . . . .
}
```

The control string indicates that the first data item will be a short decimal integer, the second will be a short hexadecimal integer, the third will be a long octal integer, and the fourth will be a long unsigned (decimal) integer. Note that the first three fields have minimum field width specifications, but the fourth does not.

Some versions of C allow the conversion characters `X`, `E` and `G` to be written in uppercase. These uppercase conversion characters cause any letters within the output data to be displayed in uppercase. (Note that this use of uppercase conversion characters is distinctly different than with the `scanf` function.)

EXAMPLE 4.26 The following program illustrates the use of uppercase conversion characters in the `printf` function.

```
#include <stdio.h>

main()      /* use of uppercase conversion characters */

{
    int a = 0x80ec;
    float b = 0.3e-12;

    printf("%4x %10.2e\n\n", a, b);
    printf("%4X %10.2E", a, b);
}
```

Notice that the first `printf` statement contains lowercase conversion characters, whereas the second `printf` statement contains uppercase conversion characters.

When the program is executed, the following output is generated.

```
80ec  3.00e-13

80EC  3.00E-13
```

The first quantity on each line is a hexadecimal number. Note that the letters `ec` (which are a part of the hexadecimal number) are shown in lowercase on the first line, and in uppercase on the second line.

The second quantity on each line is a decimal floating-point number which includes an exponent. Notice that the letter `e`, which indicates the exponent, is shown in lowercase on the first line and uppercase on the second.

You are again reminded that the use of uppercase conversion characters is not supported by all compilers.

In addition to the field width, the precision and the conversion character, each character group within the control string can include a *flag*, which affects the appearance of the output. The flag must be placed immediately after the percent sign (%). Some compilers allow two or more flags to appear consecutively, within the same character group. The more commonly used flags are listed in Table 4-3.

Table 4-3 Commonly Used Flags

Flag	*Meaning*
-	Data item is left justified within the field (blank spaces required to fill the minimum field width will be added *after* the data item rather than *before* the data item).
+	A sign (either + or -) will precede each signed numerical data item. Without this flag, only negative data items are preceded by a sign.
0	Causes leading zeros to appear instead of leading blanks. Applies only to data items that are right justified within a field whose minimum size is larger than the data item. (*Note:* Some compilers consider the zero flag to be a part of the field width specification rather than an actual flag. This assures that the 0 is processed last, if multiple flags are present.)
' ' (*blank space*)	A blank space will precede each positive signed numerical data item. This flag is overridden by the + flag if both are present.
# (*with* o- *and* x-*type conversion*)	Causes octal and hexadecimal data items to be preceded by 0 and 0x, respectively.
# (*with* e-, f- *and* g-*type conversion*)	Causes a decimal point to be present in all floating-point numbers, even if the data item is a whole number. Also prevents the truncation of trailing zeros in g-type conversion.

EXAMPLE 4.27 Here is a simple C program that illustrates the use of flags with integer and floating-point quantities.

```
#include <stdio.h>

main()    /* use of flags with integer and  floating-point numbers */
{
    int i = 123;
    float x = 12.0, y = -3.3;

    printf(":%6d %7.0f %10.1e:\n\n", i, x, y);
    printf(":%-6d %-7.0f %-10.1e:\n\n", i, x, y);
    printf(":%+6d %+7.0f %+10.1e:\n\n", i, x, y);
    printf(":%-+6d %-+7.0f %-+10.1e:\n\n", i, x, y);
    printf(":%7.0f %#7.0f %7g %#7g:", x, x, y, y);
}
```

When the program is executed, the following output is produced. (The colons indicate the beginning of the first field and the end of the last field in each line.)

```
:   123      12   -3.3e+00:

:123    12       -3.3e+00  :

:  +123     +12   -3.3e+00:

:+123   +12      -3.3e+00  :

:     12     12.    -3.3 -3.30000:
```

The first line illustrates how integer and floating-point numbers appear without any flags. Each number is right justified within its respective field. The second line shows the same numbers, using the same conversions, with a - flag included within each character group. Note that the numbers are now left justified within their respective fields. The third line shows the effect of using a + flag. The numbers are now right justified, as in the first line, but each number (whether positive or negative) is preceded by an appropriate sign.

The fourth line shows the effect of combining a - and a + flag. The numbers are now left justified and preceded by an appropriate sign. Finally, the last line shows two floating-point numbers, each displayed first without and then with the # flag. Note that the effect of the flag is to include a decimal point in the number `12.` (which is printed with `f`-type conversion), and to include the trailing zeros in the number `-3.300000` (printed with `g`-type conversion).

EXAMPLE 4.28 Now consider the following program, which displays decimal, octal and hexadecimal numbers.

```
#include <stdio.h>

main()   /* use of flags with unsigned decimal,    octal and hexadecimal numbers */
{
    int i = 1234, j = 01777, k = 0xa08c;

    printf(":%8u %8o %8x:\n\n", i, j, k);
    printf(":%-8u %-8o %-8x:\n\n", i, j, k);
    printf(":%#8u %#8o %#8X:\n\n", i, j, k);
    printf(":%08u %08o %08X:\n\n", i, j, k);
}
```

Execution of this program results in the following output. (The colons indicate the beginning of the first field and the end of the last field in each line.)

```
:    1234     1777     a08c:

:1234     1777     a08c     :

:    1234    01777   0XA08C:

:00001234 00001777 0000A08C:
```

The first line illustrates the display of unsigned integer, octal and hexadecimal output without any flags. Note that the numbers are right justified within their respective fields. The second line shows what happens when you include a - flag within each character group. Now the numbers are left justified within their respective fields.

In the third line we see what happens when the # flag is used. This flag causes the octal number `1777` to be preceded by a `0` (appearing as `01777`), and the hexadecimal number to be preceded by `0X` (i.e., `0XA08C`). Notice that the unsigned decimal integer `1234` is unaffected by this flag. Also, notice that the hexadecimal number now contains uppercase characters, since the conversion character was written in uppercase (`X`).

The last line illustrates the use of the `0` flag. This flag causes the fields to be filled with leading `0`s rather than leading blanks. We again see uppercase hexadecimal characters, in response to the uppercase conversion character (`X`).

EXAMPLE 4.29 The following program outline illustrates the use of flags with string output.

```
#include <stdio.h>

main()
{
    char line[12];

    . . . . .

    printf(":%15s %15.5s %.5s:\n\n", line, line, line);
    printf(":%-15s %-15.5s %-.5s:", line, line, line);
}
```

Now suppose that the string `lower-case` is assigned to the character array `line`. The following output will be generated when the program is executed.

```
:     lower-case          lower lower:

:lower-case     lower          lower:
```

The first line illustrates how strings are displayed when flags are not present, as explained in Example 4.24. The second line shows the same strings, left justified, in response to the – flag in each character group.

Unrecognized characters within the control string will be displayed just as they appear. This feature allows us to include labels and messages with the output data items, if we wish.

EXAMPLE 4.30 The following program illustrates how printed output can be labeled.

```
#include <stdio.h>

main()     /* labeling of floating-point numbers */

{
    float a = 2.2, b = -6.2, x1 = .005, x2 = -12.88;

    printf("$%4.2f  %7.1f%%\n\n", a, b);
    printf("x1=%7.3f   x2=%7.3f", x1, x2);
}
```

This program causes the value of `a` (`2.2`) to be preceded by a dollar sign (`$`), and the value of `b` (`-6.2`) to be followed by a percent sign (`%`). Note the two consecutive percent signs in the first `printf` statement. The first percent sign indicates the start of a character group, whereas the second percent sign is interpreted as a label.

The second `printf` statement causes the value of `x1` to be preceded by the label `x1=`, and the value of `x2` to be preceded by the label `x2=`. Three blank spaces will separate these two labeled data items.

The actual output is shown below.

```
$2.20     -6.2%

x1=  0.005   x2=-12.880
```

Remember that there is some variation in the features supported by the `printf` function in different versions of C. The features described in this section are very common, though there may be differences in the way these features are implemented. Additional features are also available in many versions of the language.

4.8 THE `gets` AND `puts` FUNCTIONS

C contains a number of other library functions that permit some form of data transfer into or out of the computer. We will encounter several such functions in Chap. 12, where we discuss data files. Before leaving this chapter, however, we mention the `gets` and `puts` functions, which facilitate the transfer of strings between the computer and the standard input/output devices.

Each of these functions accepts a single argument. The argument must be a data item that represents a string. (e.g., a character array). The string may include whitespace characters. In the case of `gets`, the string will be entered from the keyboard, and will terminate with a newline character (i.e., the string will end when the user presses the `Enter` key).

The `gets` and `puts` functions offer simple alternatives to the use of `scanf` and `printf` for reading and displaying strings, as illustrated in the following example.

EXAMPLE 4.31 Reading and Writing a Line of Text Here is another version of the simple program originally presented in Example 4.19, that reads a line of text into the computer and then writes it back out in its original form.

```
#include <stdio.h>

main()        /* read and write a line of text */

{
    char line[80];

    gets(line);
    puts(line);
}
```

This program utilizes `gets` and `puts`, rather than `scanf` and `printf`, to transfer the line of text into and out of the computer. Note that the syntax is simpler in the present program (compare carefully with the program shown in Example 4.19). On the other hand, the `scanf` and `printf` functions in the earlier program can be expanded to include additional data items, whereas the present program cannot.

When this program is executed, it will behave in exactly the same manner as the program shown in Example 4.19.

4.9 INTERACTIVE (CONVERSATIONAL) PROGRAMMING

Many modern computer programs are designed to create an interactive dialog between the computer and the person using the program (the "user"). These dialogs usually involve some form of question-answer interaction, where the computer asks the questions and the user provides the answers, or vice versa. The computer and the user thus appear to be carrying on some limited form of conversation.

In C, such dialogs can be created by alternate use of the `scanf` and `printf` functions. The actual programming is straightforward, though sometimes confusing to beginners, since the `printf` function is used both when entering data (to create the computer's questions) and when displaying results. On the other hand, `scanf` is used only for actual data entry.

The basic ideas are illustrated in the following example.

EXAMPLE 4.32 Averaging Student Exam Scores This example presents a simple, interactive C program that reads in a student's name and three exam scores, and then calculates an average score. The data will be entered interactively, with the computer asking the user for information and the user supplying the information in a free format, as requested. Each input data item will be entered on a separate line. Once all of the data have been entered, the computer will compute the desired average and write out all of the data (both the input data and the calculated average).

The actual program is shown below.

```
#include <stdio.h>

main()     /* sample interactive program */

{
    char name[20];
    float score1, score2, score3, avg;

    printf("Please enter your name: ");          /* enter name */
    scanf(" %[^\n]", name);

    printf("Please enter the first score:  ");   /* enter 1st score */
    scanf("%f", &score1);

    printf("Please enter the second score: ");   /* enter 2nd score */
    scanf("%f", &score2);

    printf("Please enter the third score:  ");   /* enter 3rd score */
```

```
        scanf("%f", &score3);

        avg = (score1+score2+score3)/3;           /* calculate avg */

        printf("\n\nName: %-s\n\n", name);        /* write output */
        printf("Score 1: %-5.1f\n", score1);
        printf("Score 2: %-5.1f\n", score2);
        printf("Score 3: %-5.1f\n\n", score3);
        printf("Average: %-5.1f\n\n", avg);
    }
```

Notice that two statements are associated with each input data item. The first is a `printf` statement, which generates a request for the item. The second statement, a `scanf` function, causes the data item to be entered from the standard input device (i.e., the keyboard).

After the student's name and all three exam scores have been entered, an average exam score is calculated. The input data and the calculated average are then displayed, as a result of the group of `printf` statements at the end of the program.

A typical interactive session is shown below. To illustrate the nature of the dialog, the user's responses have been underlined.

```
Please enter your name: Robert Smith
Please enter the first score:  88
Please enter the second score: 62.5
Please enter the third score:  90

Name: Robert Smith

Score 1:       88.0
Score 2:       62.5
Score 3:       90.0

Average:       80.2
```

Additional interactive programs will be seen in many of the programming examples presented in later chapters of this book.

Review Questions

4.1 What are the commonly used input/output functions in C? How are they accessed?

4.2 What is the standard input/output header file called in most versions of C? How is the file included within a program?

4.3 What is the purpose of the `getchar` function? How is it used within a C program?

4.4 What happens when an end-of-file condition is encountered when reading characters with the `getchar` function? How is the end-of-file condition recognized?

4.5 How can the `getchar` function be used to read multicharacter strings?

4.6 What is the purpose of the `putchar` function? How is it used within a C program? Compare with the `getchar` function.

4.7 How can the `putchar` function be used to write multicharacter strings?

4.8 What is a character-type array? What does each element of a character-type array represent? How are character-type arrays used to represent multicharacter strings?

4.9 What is the purpose of the `scanf` function? How is it used within a C program? Compare with the `getchar` function.

4.10 What is the purpose of the control string in a `scanf` function? What type of information does it convey? Of what is the control string composed?

4.11 How is each character group within the control string identified? What are the constituent characters within a character group?

4.12 If a control string within a `scanf` function contains multiple character groups, how are the character groups separated? Are whitespace characters required?

4.13 If whitespace characters are present within a control string, how are they interpreted?

4.14 Summarize the meaning of the more commonly used conversion characters within the control string of a `scanf` function.

4.15 What special symbol must be included with the arguments, other than the control string, in a `scanf` function? In what way are array names treated differently than other arguments?

4.16 When entering data via the `scanf` function, what relationships must there be between the data items and the corresponding arguments? How are multiple data items separated from one another?

4.17 When entering data via the `scanf` function, must octal data be preceded by `0`? Must hexadecimal data be preceded by `0x` (or `0X`)? How must floating-point data be written?

4.18 When entering a string via the `scanf` function using an `s`-type conversion factor, how is the string terminated?

4.19 When entering a string via the `scanf` function, how can a single string which includes whitespace characters be entered?

4.20 Summarize a convenient method for entering a string of undetermined length, which may contain whitespace characters and all printable characters, and which is terminated by a carriage return. Answer this question relative to the type of conversion required within the control string of a `scanf` function.

4.21 What is meant by a field?

4.22 How can the maximum field width for a data item be specified within a `scanf` function?

4.23 What happens if an input data item contains more characters than the maximum allowable field width? What if the data item contains fewer characters?

4.24 How can short integer, long integer and double-precision arguments be indicated within the control string of a `scanf` function?

4.25 How can long double arguments be indicated within the control string of a `scanf` function? Is this feature available in most versions of C?

4.26 How can the assignment of an input data item to its corresponding argument be suppressed?

4.27 If the control string within a `scanf` function contains multiple character groups without interspersed whitespace characters, what difficulty can arise when using `c`-type conversion? How can this difficulty be avoided?

4.28 How are unrecognized characters within the control string of a `scanf` function interpreted?

4.29 What is the purpose of the `printf` function? How is it used within a C program? Compare with the `putchar` function.

4.30 In what ways does the control string within a `printf` function differ from the control string within a `scanf` function?

4.31 If the control string within a `printf` function contains multiple character groups, how are the character groups separated? How are the separators interpreted?

4.32 Summarize the meaning of the more commonly used conversion characters within the control string of a `printf` function. Compare with the conversion characters that are used in a `scanf` function.

4.33 In a `printf` function, must the arguments (other than the control string) be preceded by ampersands? Compare with the `scanf` function and explain any differences.

4.34 What is the difference between `f`-type conversion, `e`-type conversion and `g`-type conversion when outputting floating-point data with a `printf` function?

4.35 Compare the use of `s`-type conversion in the `printf` and the `scanf` functions. How does `s`-type conversion differ when processing strings containing whitespace characters?

4.36 How can the minimum field width for a data item be specified within the `printf` function?

4.37 What happens if an output data item contains more characters than the minimum field width? What if the data item contains fewer characters? Contrast with the field width specifications in the `scanf` function.

4.38 What is meant by the precision of an output data item? To what types of data does this apply?

4.39 How can the precision be specified within a `printf` function?

4.40 What happens to a floating-point number if it must be shortened to conform to a precision specification? What happens to a string?

4.41 Must a precision specification be accompanied by a minimum field width specification in a `printf` function?

4.42 How can short integer, long integer and double-precision arguments be indicated within the control string of a `printf` function? How can long double arguments be indicated?

4.43 How are uppercase conversion characters interpreted differently than the corresponding lowercase conversion characters in a `printf` function? To what types of conversion does this feature apply? Do all versions of C recognize this distinction?

4.44 Summarize the purpose of the flags that are commonly used within the `printf` function.

4.45 Can two or more flags appear consecutively within the same character group?

4.46 How are unrecognized characters within the control string of a `printf` function interpreted?

4.47 How can labeled data items be generated by the `printf` function?

4.48 Summarize the use of the `gets` and `puts` functions to transfer strings between the computer and the standard input/output devices. Compare the use of these functions with the string transfer features in the `scanf` and `printf` statements.

4.49 Explain, in general terms, how an interactive dialog can be generated by repeated use of pairs of `scanf` and `printf` functions.

Problems

4.50 A C program contains the following statements:

```
#include <stdio.h>

char a, b, c;
```

(*a*) Write appropriate `getchar` statements that will allow values for `a`, `b` and `c` to be entered into the computer.

(*b*) Write appropriate `putchar` statements that will allow the current values of `a`, `b` and `c` to be written out of the computer (i.e., to be displayed).

4.51 Solve Prob. 4.50 using a single `scanf` function and a single `printf` function rather than the `getchar` and `putchar` statements. Compare your answer with the solution to Prob. 4.50.

4.52 A C program contains the following statements:

```
#include <stdio.h>

char text[80];
```

(*a*) Write a `for` statement that will permit a 60-character message to be entered into the computer and stored in the character array `text`. Include a reference to the `getchar` function in the `for` loop, as in Example 4.4.

(*b*) Write a `for` statement that will permit the first 60 characters of the character array `text` to be displayed. Include a reference to the `putchar` function in the `for` loop, as in Example 4.4.

4.53 Modify the solution to Prob. 4.52(*a*) so that a character array whose length is unspecified can be read into the computer. Assume that the message does not exceed 79 characters, and that it is automatically terminated by a *newline* character (`\n`). (See Example 4.4.)

4.54 Solve Prob. 4.53 using a `scanf` statement in place of a `for` statement (see Example 4.8). What additional information is provided by the method described in Prob. 4.53?

4.55 A C program contains the following statements:

```
#include <stdio.h>
int i, j, k;
```

Write an appropriate `scanf` function to enter numerical values for `i, j` and `k`, assuming

(*a*) The values for `i`, `j` and `k` will be decimal integers.

(*b*) The value for `i` will be a decimal integer, `j` an octal integer and `k` a hexadecimal integer.

(*c*) The values for `i` and `j` will be hexadecimal integers and `k` will be an octal integer.

4.56 A C program contains the following statements:

```
#include <stdio.h>
int i, j, k;
```

Write an appropriate `scanf` function to enter numerical values for `i, j` and `k` into the computer, assuming

(*a*) The values for `i`, `j` and `k` will be decimal integers not exceeding six characters each.

(*b*) The value for `i` will be a decimal integer, `j` an octal integer and `k` a hexadecimal integer, with each quantity not exceeding 8 characters.

(*c*) The values for `i` and `j` will be hexadecimal integers and `k` will be an octal integer. Each quantity will be 7 or fewer characters.

4.57 Interpret the meaning of the control string in each of the following `scanf` functions.

(*a*) `scanf("%12ld %5hd %15lf %15le", &a, &b, &c, &d);`

(*b*) `scanf("%10lx %6ho %5hu %14lu", &a, &b, &c, &d);`

(*c*) `scanf("%12D %hd %15f %15e", &a, &b, &c, &d);`

(*d*) `scanf("%8d %*d %12lf %12lf", &a, &b, &c, &d);`

4.58 A C program contains the following statements:

```
#include <stdio.h>

int i, j;
long ix;
short s;
unsigned u;
float x;
double dx;
char c;
```

For each of the following groups of variables, write a `scanf` function that will allow a set of data items to be read into the computer and assigned to the variables. Assume that all integers will be read in as decimal quantities.

(*a*) `i, j, x` and `dx`

(*b*) `i, ix, j, x` and `u`

(*c*) `i, u` and `c`

(*d*) `c, x, dx` and `s`

4.59 A C program contains the following statements:

```
#include <stdio.h>

int i, j;
long ix;
short s;
unsigned u;
float x;
double dx;
char c;
```

Write an appropriate `scanf` function to accommodate each of the following situations, assuming that all integers will be read in as decimal quantities.

(*a*) Enter values for `i`, `j`, `x` and `dx`, assuming that each integer quantity does not exceed four characters, the floating-point quantity does not exceed eight characters, and the double-precision quantity does not exceed 15 characters.

(*b*) Enter values for `i`, `ix`, `j`, `x` and `u`, assuming that each integer quantity does not exceed five characters, the long integer does not exceed 12 characters, and the floating-point quantity does not exceed 10 characters.

(*c*) Enter values for `i`, `u` and `c`, assuming that each integer quantity does not exceed six characters.

(*d*) Enter values for `c`, `x`, `dx` and `s`, assuming that the floating-point quantity does not exceed nine characters, the double-precision quantity does not exceed 16 characters and the short integer does not exceed six characters.

4.60 A C program contains the following statements:

```
#include <stdio.h>

char text[80];
```

Write a `scanf` function that will allow a string to be read into the computer and assigned to the character array `text`. Assume that the string does not contain any whitespace characters.

4.61 Solve Prob. 4.60 assuming that the string contains only lowercase letters, blank spaces and newline characters.

4.62 Solve Prob. 4.60 assuming that the string contains only uppercase letters, digits, dollar signs and blank spaces.

4.63 Solve Prob. 4.60 assuming that the string contains anything other than an asterisk (i.e., assume that an asterisk will be used to indicate the end of the string).

4.64 A C program contains the following statements.

```
#include <stdio.h>

char a, b, c;
```

Suppose that `$` is to be entered into the computer and assigned to `a`, `*` assigned to `b` and `@` assigned to `c`. Show how the input data must be entered for each of the following `scanf` functions.

(*a*) `scanf("%c%c%c", &a, &b, &c);`

(*b*) `scanf("%c %c %c", &a, &b, &c);`

(*c*) `scanf("%s%s%s", &a, &b, &c);`

(*d*) `scanf("%s %s %s", &a, &b, &c);`

(*e*) `scanf("%1s%1s%1s", &a, &b, &c);`

4.65 A C program contains the following statements.

```
#include <stdio.h>

int a, b;
float x, y;
```

Suppose the value 12 is to be entered into the computer and assigned to `a`, -8 assigned to `b`, 0.011 assigned to `x` and -2.2×10^6 assigned to `y`. Show how the input data might most conveniently be entered for each of the following `scanf` functions.

(*a*) `scanf("%d %d %f %f", &a, &b, &x, &y);`

(*b*) `scanf("%d %d %e %e", &a, &b, &x, &y);`

(*c*) `scanf("%2d %2d %5f %6e", &a, &b, &x, &y);`

(*d*) `scanf("%3d %3d %8f %8e", &a, &b, &x, &y);`

4.66 A C program contains the following statements:

```
#include <stdio.h>

int i, j, k;
```

Write a `printf` function for each of the following groups of variables or expressions. Assume all variables represent decimal integers.

(*a*) `i`, `j` and `k`

(*b*) `(i + j), (i - k)`

(*c*) `sqrt(i + j), abs(i - k)`

4.67 A C program contains the following statements:

```
#include <stdio.h>

int i, j, k;
```

Write a `printf` function for each of the following groups of variables or expressions. Assume all variables represent decimal integers.

(*a*) `i`, `j` and `k`, with a minimum field width of three characters per quantity.

(*b*) `(i + j), (i - k)`, with a minimum field width of five characters per quantity.

(*c*) `sqrt(i + j), abs(i - k)`, with a minimum field width of nine characters for the first quantity, and seven characters for the second quantity.

4.68 A C program contains the following statements:

```
#include <stdio.h>

float x, y, z;
```

Write a `printf` function for each of the following groups of variables or expressions.

(*a*) `x`, `y` and `z`

(*b*) `(x + y), (x - z)`

(*c*) `sqrt(x + y), fabs(x - z)`

4.69 A C program contains the following statements:

```
#include <stdio.h>

float x, y, z;
```

Write a `printf` function for each of the following groups of variables or expressions, using `f`-type conversion for each floating-point quantity.

(*a*) `x`, `y` and `z`, with a minimum field width of six characters per quantity.

(*b*) `(x + y), (x - z)`, with a minimum field width of eight characters per quantity.

(*c*) `sqrt(x + y), abs(x - z)`, with a minimum field width of 12 characters for the first quantity and nine characters for the second.

4.70 Repeat the previous problem using `e`-type conversion.

4.71 A C program contains the following statements:

```
#include <stdio.h>

float x, y, z;
```

Write a `printf` function for each of the following groups of variables or expressions, using `f`-type conversion for each floating-point quantity.

(*a*) `x`, `y` and `z`, with a minimum field width of eight characters per quantity, with no more than four decimal places.

(*b*) `(x + y), (x - z)`, with a minimum field width of nine characters per quantity, with no more than three decimal places.

(*c*) `sqrt(x + y), abs(x - z)`, with a minimum field width of 12 characters for the first quantity and 10 characters for the second. Display a maximum of four decimal places for each quantity.

4.72 A C program contains the following statements:

```
#include <stdio.h>

float x, y, z;
```

Write a `printf` function for each of the following groups of variables or expressions, using e-type conversion for each floating-point quantity.

(*a*) `x, y` and `z`, with a minimum field width of 12 characters per quantity, with no more than four decimal places.

(*b*) `(x + y), (x - z)`, with a minimum field width of 14 characters per quantity, with no more than five decimal places.

(*c*) `sqrt(x + y), abs(x - z)`, with a minimum field width of 12 characters for the first quantity and 15 characters for the second. Display a maximum of seven decimal places for each quantity.

4.73 A C program contains the following statements:

```
#include <stdio.h>

int a = 0177, b = 055, c = 0xa8, d = 0x1ff;
```

Write a `printf` function for each of the following groups of variables or expressions.

(*a*) `a, b, c` and `d`

(*b*) `(a + b), (c - d)`

4.74 A C program contains the following statements:

```
#include <stdio.h>

int i, j;
long ix;
unsigned u;
float x;
double dx;
char c;
```

For each of the following groups of variables, write a `printf` function that will allow the values of the variables to be displayed. Assume that all integers will be shown as decimal quantities.

(*a*) `i, j, x` and `dx`

(*b*) `i, ix, j, x` and `u`

(*c*) `i, u` and `c`

(*d*) `c, x, dx` and `ix`

4.75 A C program contains the following statements:

```
#include <stdio.h>

int i, j;
long ix;
unsigned u;
float x;
double dx;
char c;
```

Write an appropriate `printf` function for each of the following situations, assuming that all integers will be displayed as decimal quantities.

(*a*) Display the values of `i`, `j`, `x` and `dx`, assuming that each integer quantity will have a minimum field width of four characters and each floating-point quantity is displayed in exponential notation with a total of at least 14 characters and no more than eight decimal places.

(*b*) Repeat part (*a*), displaying each quantity on a separate line.

(*c*) Display the values of `i`, `ix`, `j`, `x` and `u`, assuming that each integer quantity will have a minimum field width of five characters, the long integer will have a minimum field width of 12 characters and the floating-point quantity will be have at least 10 characters with a maximum of five decimal places. Do not include an exponent.

(*d*) Repeat part (*c*), displaying the first three quantities on one line, followed by a blank line and then the remaining two quantities on the next line.

(*e*) Display the values of `i`, `u` and `c`, with a minimum field width of six characters for each integer quantity. Place three blank spaces between each output quantity.

(*f*) Display the values for `j`, `u` and `x`. Display the integer quantities with a minimum field width of five characters. Display the floating-point quantity using `f`-type conversion, with a minimum field width of 11 and a maximum of four decimal places.

(*g*) Repeat part (*f*), with each data item left justified within its respective field.

(*h*) Repeat part (*f*), with a sign (either + or –) preceding each signed data item.

(*i*) Repeat part (*f*), with leading zeros filling out the field for each of the integer quantities.

(*j*) Repeat part (*f*), with a provision for a decimal point in the value of `x` regardless of its value.

4.76 Assume that `i`, `j` and `k` are integer variables, and that `i` represents an octal quantity, `j` represents a decimal quantity and `k` represents a hexadecimal quantity. Write an appropriate `printf` function for each of the following situations.

(*a*) Display the values for `i`, `j` and `k`, with a minimum field width of eight characters for each value.

(*b*) Repeat part (*a*) with each output data item left justified within its respective field.

(*c*) Repeat part (*a*) with each output data item preceded by zeros (0x, in the case of the hexadecimal quantity).

4.77 A C program contains the following variable declarations.

```
int i = 12345, j = -13579, k = -24680;
long ix = 123456789;
short sx = -2222;
unsigned ux = 5555;
```

Show the output resulting from each of the following `printf` statements.

(*a*) `printf("%d %d %d %ld %d %u", i, j, k, ix, sx, ux);`

(*b*) `printf("%3d %3d %3d\n\n%3ld %3d %3u", i, j, k, ix, sx, ux);`

(*c*) `printf("%8d %8d %8d\n\n%15ld %8d %8u", i, j, k, ix, sx, ux);`

(*d*) `printf("%-8d %-8d\n%-8d %-15ld\n%-8d %-8u", i, j, k, ix, sx, ux);`

(*e*) `printf("%+8d %+8d\n%+8d %+15ld\n%+8d %8u", i, j, k, ix, sx, ux);`

(*f*) `printf("%08d %08d\n%08d %015ld\n%08d %08u", i, j, k, ix, sx, ux);`

4.78 A C program contains the following variable declarations.

```
int i = 12345, j = 0xabcd9, k = 077777;
```

Show the output resulting from each of the following `printf` statements.

(*a*) `printf("%d %x %o", i, j, k);`

(*b*) `printf("%3d %3x %3o", i, j, k);`

(*c*) `printf("%8d %8x %8o", i, j, k);`

(*d*) `printf("%-8d %-8x %-8o", i, j, k);`

(*e*) `printf("%+8d %+8x %+8o", i, j, k);`

(*f*) `printf("%08d %#8x %#8o", i, j, k);`

4.79 A C program contains the following variable declarations.

```
float a = 2.5, b = 0.0005, c = 3000.;
```

Show the output resulting from each of the following `printf` statements.

(*a*) `printf("%f %f %f", a, b, c);`

(*b*) `printf("%3f %3f %3f", a, b, c);`

(*c*) `printf("%8f %8f %8f", a, b, c);`

(*d*) `printf("%8.4f %8.4f %8.4f", a, b, c);`

(*e*) `printf("%8.3f %8.3f %8.3f", a, b, c);`

(*f*) `printf("%e %e %e", a, b, c);`

(*g*) `printf("%3e %3e %3e", a, b, c);`

(*h*) `printf("%12e %12e %12e", a, b, c);`

(*i*) `printf("%12.4e %12.4e %12.4e", a, b, c);`

(*j*) `printf("%8.2e %8.2e %8.2e", a, b, c);`

(*k*) `printf("%-8f %-8f %-8f", a, b, c);`

(*l*) `printf("%+8f %+8f %+8f", a, b, c);`

(*m*) `printf("%08f %08f %08f", a, b, c);`

(*n*) `printf("%#8f %#8f %#8f", a, b, c);`

(*o*) `printf("%g %g %g", a, b, c);`

(*p*) `printf("%#g %#g %#g", a, b, c);`

4.80 A C program contains the following variable declarations.

```
char c1 = 'A', c2 = 'B', c3 = 'C';
```

Show the output resulting from each of the following `printf` statements.

(*a*) `printf("%c %c %c", c1, c2, c3);`

(*b*) `printf("%c%c%c", c1, c2, c3);`

(*c*) `printf("%3c %3c %3c", c1, c2, c3);`

(*d*) `printf("%3c%3c%3c", c1, c2, c3);`

(*e*) `printf("c1=%c c2=%c c3=%c", c1, c2, c3);`

4.81 A C program contains the following statements.

```
#include <stdio.h>

char text[80];
```

Write a `printf` function that will allow the contents of `text` to be displayed in the following ways.

(*a*) Entirely on one line.

(*b*) Only the first eight characters.

(*c*) The first eight characters, preceded by five blanks.

(*d*) The first eight characters, followed by five blanks.

4.82 A C program contains the following array declaration.

```
char text[80];
```

Suppose that the following string has been assigned to `text`.

```
Programming with C can be a challenging creative activity.
```

Show the output resulting from the following `printf` statements.

(*a*) `printf("%s", text);`

(*b*) `printf("%18s", text);`

(*c*) `printf("%.18s", text);`

(*d*) `printf("%18.7s", text);`

(*e*) `printf("%-18.7s", text);`

4.83 Write the necessary `scanf` or `printf` statements for each of the following situations.

(*a*) Generate the message

```
Please enter your name:
```

Then enter the name on the same line. Assign the name to a character-type array called `name`.

(*b*) Suppose that `x1` and `x2` are floating-point variables whose values are 8.0 and −2.5, respectively. Display the values of `x1` and `x2`, with appropriate labels; i.e., generate the message

```
x1 =  8.0   x2 = -2.5
```

(*c*) Suppose that `a` and `b` are integer variables. Prompt the user for input values of these two variables, then display their sum. Label the output accordingly.

4.84 Determine which conversion characters are available with your particular version of C. Also, determine which flags are available for data output.