3 Formatted Input/Output

In seeking the unattainable, simplicity only gets in the way.

`scanf` and `printf`, which support formatted reading and writing, are two of the most frequently used functions in C. As this chapter shows, both are powerful but tricky to use properly. Section 3.1 describes `printf`, and Section 3.2 covers `scanf`. Neither section gives complete details, which will have to wait until Chapter 22.

3.1 The `printf` Function

The `printf` function is designed to display the contents of a string, known as the ***format string,*** with values possibly inserted at specified points in the string. When it's called, `printf` must be supplied with the format string, followed by any values that are to be inserted into the string during printing:

`printf(`*string*`,` $expr_1$`,` $expr_2$`, ...);`

The values displayed can be constants, variables, or more complicated expressions. There's no limit on the number of values that can be printed by a single call of `printf`.

The format string may contain both ordinary characters and ***conversion specifications***, which begin with the `%` character. A conversion specification is a placeholder representing a value to be filled in during printing. The information that follows the `%` character *specifies* how the value is *converted* from its internal form (binary) to printed form (characters)—that's where the term "conversion specification" comes from. For example, the conversion specification `%d` specifies that `printf` is to convert an `int` value from binary to a string of decimal digits, while `%f` does the same for a `float` value.

Ordinary characters in a format string are printed exactly as they appear in the string; conversion specifications are replaced by the values to be printed. Consider the following example:

```
int i, j;
float x, y;

i = 10;
j = 20;
x = 43.2892f;
y = 5527.0f;

printf("i = %d, j = %d, x = %f, y = %f\n", i, j, x, y);
```

This call of `printf` produces the following output:

```
i = 10, j = 20, x = 43.289200, y = 5527.000000
```

The ordinary characters in the format string are simply copied to the output line. The four conversion specifications are replaced by the values of the variables `i`, `j`, `x`, and `y`, in that order.

C compilers aren't required to check that the number of conversion specifications in a format string matches the number of output items. The following call of `printf` has more conversion specifications than values to be printed:

```
printf("%d %d\n", i);   /*** WRONG ***/
```

`printf` will print the value of `i` correctly, then print a second (meaningless) integer value. A call with too few conversion specifications has similar problems:

```
printf("%d\n", i, j);   /*** WRONG ***/
```

In this case, `printf` prints the value of `i` but doesn't show the value of `j`.

Furthermore, compilers aren't required to check that a conversion specification is appropriate for the type of item being printed. If the programmer uses an incorrect specification, the program will simply produce meaningless output. Consider the following call of `printf`, in which the `int` variable `i` and the `float` variable `x` are in the wrong order:

```
printf("%f %d\n", i, x);   /*** WRONG ***/
```

Since `printf` must obey the format string, it will dutifully display a `float` value, followed by an `int` value. Unfortunately, both will be meaningless.

Conversion Specifications

Conversion specifications give the programmer a great deal of control over the appearance of output. On the other hand, they can be complicated and hard to read. In fact, describing conversion specifications in complete detail is too arduous a

task to tackle this early in the book. Instead, we'll just take a brief look at some of their more important capabilities.

In Chapter 2, we saw that a conversion specification can include formatting information. In particular, we used `%.1f` to display a `float` value with one digit after the decimal point. More generally, a conversion specification can have the form `%`*m*`.`*p*`X` or `%-`*m*`.`*p*`X`, where *m* and *p* are integer constants and *X* is a letter. Both *m* and *p* are optional; if *p* is omitted, the period that separates *m* and *p* is also dropped. In the conversion specification `%10.2f`, *m* is `10`, *p* is `2`, and *X* is `f`. In the specification `%10f`, *m* is `10` and *p* (along with the period) is missing, but in the specification `%.2f`, *p* is `2` and *m* is missing.

The ***minimum field width***, *m*, specifies the minimum number of characters to print. If the value to be printed requires fewer than *m* characters, the value is right-justified within the field. (In other words, extra spaces precede the value.) For example, the specification `%4d` would display the number 123 as `•123`. (In this chapter, I'll use • to represent the space character.) If the value to be printed requires more than *m* characters, the field width automatically expands to the necessary size. Thus, the specification `%4d` would display the number 12345 as `12345`—no digits are lost. Putting a minus sign in front of *m* causes left justification; the specification `%-4d` would display 123 as `123•`.

The meaning of the ***precision***, *p*, isn't as easily described, since it depends on the choice of *X*, the ***conversion specifier***. *X* indicates which conversion should be applied to the value before it's printed. The most common conversion specifiers for numbers are:

Q&A

- `d` — Displays an integer in decimal (base 10) form. *p* indicates the minimum number of digits to display (extra zeros are added to the beginning of the number if necessary); if *p* is omitted, it is assumed to have the value 1. (In other words, `%d` is the same as `%.1d`.)
- `e` — Displays a floating-point number in exponential format (scientific notation). *p* indicates how many digits should appear after the decimal point (the default is 6). If *p* is `0`, the decimal point is not displayed.
- `f` — Displays a floating-point number in "fixed decimal" format, without an exponent. *p* has the same meaning as for the `e` specifier.
- `g` — Displays a floating-point number in either exponential format or fixed decimal format, depending on the number's size. *p* indicates the maximum number of significant digits (*not* digits after the decimal point) to be displayed. Unlike the `f` conversion, the `g` conversion won't show trailing zeros. Furthermore, if the value to be printed has no digits after the decimal point, `g` doesn't display the decimal point.

The `g` specifier is especially useful for displaying numbers whose size can't be predicted when the program is written or that tend to vary widely in size. When used to print a moderately large or moderately small number, the `g` specifier uses fixed decimal format. But when used to print a very large or very small number, the `g` specifier switches to exponential format so that the number will require fewer characters.

specifiers for integers ➤7.1
specifiers for floats ➤7.2
specifiers for characters ➤7.3
specifiers for strings ➤13.3

There are many other specifiers besides `%d`, `%e`, `%f`, and `%g`. I'll gradually introduce many of them in subsequent chapters. For the full list, and for a complete explanation of the other capabilities of conversion specifications, consult Section 22.3.

PROGRAM

Using `printf` to Format Numbers

The following program illustrates the use of `printf` to print integers and floating-point numbers in various formats.

tprintf.c

```
/* Prints int and float values in various formats */

#include <stdio.h>

int main(void)
{
  int i;
  float x;

  i = 40;
  x = 839.21f;

  printf("|%d|%5d|%-5d|%5.3d|\n", i, i, i, i);
  printf("|%10.3f|%10.3e|%-10g|\n", x, x, x);

  return 0;
}
```

The | characters in the `printf` format strings are there merely to help show how much space each number occupies when printed; unlike `%` or `\`, the | character has no special significance to `printf`. The output of this program is:

```
|40|   40|40   |  040|
|   839.210| 8.392e+02|839.21    |
```

Let's take a closer look at the conversion specifications used in this program:

- `%d` — Displays `i` in decimal form, using a minimum amount of space.
- `%5d` — Displays `i` in decimal form, using a minimum of five characters. Since `i` requires only two characters, three spaces were added.
- `%-5d` — Displays `i` in decimal form, using a minimum of five characters; since the value of `i` doesn't require five characters, the spaces are added afterward (that is, `i` is left-justified in a field of length five).
- `%5.3d` — Displays `i` in decimal form, using a minimum of five characters overall and a minimum of three digits. Since `i` is only two digits long, an extra zero was added to guarantee three digits. The resulting number is only three characters long, so two spaces were added, for a total of five characters (`i` is right-justified).
- `%10.3f` — Displays `x` in fixed decimal form, using 10 characters overall,

with three digits after the decimal point. Since x requires only seven characters (three before the decimal point, three after the decimal point, and one for the decimal point itself), three spaces precede x.

- %10.3e — Displays x in exponential form, using 10 characters overall, with three digits after the decimal point. x requires nine characters altogether (including the exponent), so one space precedes x.
- %-10g — Displays x in either fixed decimal form or exponential form, using 10 characters overall. In this case, printf chose to display x in fixed decimal form. The presence of the minus sign forces left justification, so x is followed by four spaces.

Escape Sequences

The \n code that we often use in format strings is called an ***escape sequence***. Escape sequences enable strings to contain characters that would otherwise cause problems for the compiler, including nonprinting (control) characters and characters that have a special meaning to the compiler (such as "). We'll provide a complete list of escape sequences later; for now, here's a sample:

escape sequences ➤7.3

Alert (bell)	\a
Backspace	\b
New line	\n
Horizontal tab	\t

When they appear in printf format strings, these escape sequences represent actions to perform upon printing. Printing \a causes an audible beep on most machines. Printing \b moves the cursor back one position. Printing \n advances the cursor to the beginning of the next line. Printing \t moves the cursor to the next tab stop.

Q&A

A string may contain any number of escape sequences. Consider the following printf example, in which the format string contains six escape sequences:

```
printf("Item\tUnit\tPurchase\n\tPrice\tDate\n");
```

Executing this statement prints a two-line heading:

```
Item    Unit    Purchase
        Price   Date
```

Another common escape sequence is \", which represents the " character. Since the " character marks the beginning and end of a string, it can't appear within a string without the use of this escape sequence. Here's an example:

```
printf("\"Hello!\"");
```

This statement produces the following output:

```
"Hello!"
```

Incidentally, you can't just put a single \ character in a string; the compiler will assume that it's the beginning of an escape sequence. To print a single \ character, put two \ characters in the string:

```
printf("\\");   /* prints one \ character */
```

3.2 The `scanf` Function

Just as `printf` prints output in a specified format, `scanf` reads input according to a particular format. A `scanf` format string, like a `printf` format string, may contain both ordinary characters and conversion specifications. The conversions allowed with `scanf` are essentially the same as those used with `printf`.

In many cases, a `scanf` format string will contain only conversion specifications, as in the following example:

```
int i, j;
float x, y;

scanf("%d%d%f%f", &i, &j, &x, &y);
```

Suppose that the user enters the following input line:

```
1 -20 .3 -4.0e3
```

`scanf` will read the line, converting its characters to the numbers they represent, and then assign 1, –20, 0.3, and –4000.0 to `i`, `j`, `x`, and `y`, respectively. "Tightly packed" format strings like `"%d%d%f%f"` are common in `scanf` calls. `printf` format strings are less likely to have adjacent conversion specifications.

`scanf`, like `printf`, contains several traps for the unwary. When using `scanf`, the programmer must check that the number of conversion specifications matches the number of input variables and that each conversion is appropriate for the corresponding variable—as with `printf`, the compiler isn't required to check for a possible mismatch. Another trap involves the `&` symbol, which normally precedes each variable in a `scanf` call. The `&` is usually (but not always) required, and it's the programmer's responsibility to remember to use it.

Forgetting to put the `&` symbol in front of a variable in a call of `scanf` will have unpredictable—and possibly disastrous—results. A program crash is a common outcome. At the very least, the value that is read from the input won't be stored in the variable; instead, the variable will retain its old value (which may be meaningless if the variable wasn't given an initial value). Omitting the `&` is an extremely common error—be careful! Some compilers can spot this error and produce a warning message such as *"format argument is not a pointer."* (The term *pointer* is defined in Chapter 11; the `&` symbol is used to create a pointer to a variable.) If you get a warning, check for a missing `&`.

Calling scanf is a powerful but unforgiving way to read data. Many professional C programmers avoid scanf, instead reading all data in character form and converting it to numeric form later. We'll use scanf quite a bit, especially in the early chapters of this book, because it provides a simple way to read numbers. Be aware, however, that many of our programs won't behave properly if the user enters unexpected input. As we'll see later, it's possible to have a program test whether scanf successfully read the requested data (and attempt to recover if it didn't). Such tests are impractical for the programs in this book—they would add too many statements and obscure the point of the examples.

detecting errors in scanf ➤22.3

How scanf Works

scanf can actually do much more than I've indicated so far. It is essentially a "pattern-matching" function that tries to match up groups of input characters with conversion specifications.

Like the printf function, scanf is controlled by the format string. When it is called, scanf begins processing the information in the string, starting at the left. For each conversion specification in the format string, scanf tries to locate an item of the appropriate type in the input data, skipping blank space if necessary. scanf then reads the item, stopping when it encounters a character that can't possibly belong to the item. If the item was read successfully, scanf continues processing the rest of the format string. If any item is not read successfully, scanf returns immediately without looking at the rest of the format string (or the remaining input data).

As it searches for the beginning of a number, scanf ignores ***white-space characters*** (the space, horizontal and vertical tab, form-feed, and new-line characters). As a result, numbers can be put on a single line or spread out over several lines. Consider the following call of scanf:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

Suppose that the user enters three lines of input:

```
  1
-20   .3
   -4.0e3
```

scanf sees one continuous stream of characters:

```
••1¤-20•••.3¤•••-4.0e3¤
```

(I'm using • to represent the space character and ¤ to represent the new-line character.) Since it skips over white-space characters as it looks for the beginning of each number, scanf will be able to read the numbers successfully. In the following diagram, an *s* under a character indicates that it was skipped, and an *r* indicates it was read as part of an input item:

```
••1¤-20•••.3¤•••-4.0e3¤
ssrsrrrsssrrssssrrrrrr
```

scanf "peeks" at the final new-line character without actually reading it. This new-line will be the first character read by the next call of scanf.

What rules does scanf follow to recognize an integer or a floating-point number? When asked to read an integer, scanf first searches for a digit, a plus sign, or a minus sign; it then reads digits until it reaches a nondigit. When asked to read a floating-point number, scanf looks for

a plus or minus sign (optional), followed by

a series of digits (possibly containing a decimal point), followed by

an exponent (optional). An exponent consists of the letter e (or E), an optional sign, and one or more digits.

The %e, %f, and %g conversions are interchangeable when used with scanf; all three follow the same rules for recognizing a floating-point number.

Q&A When scanf encounters a character that can't be part of the current item, the character is "put back" to be read again during the scanning of the next input item or during the next call of scanf. Consider the following (admittedly pathological) arrangement of our four numbers:

```
1-20.3-4.0e3¤
```

Let's use the same call of scanf as before:

```
scanf("%d%d%f%f", &i, &j, &x, &y);
```

Here's how scanf would process the new input:

- Conversion specification: %d. The first nonblank input character is 1; since integers can begin with 1, scanf then reads the next character, -. Recognizing that - can't appear inside an integer, scanf stores 1 into i and puts the - character back.
- Conversion specification: %d. scanf then reads the characters -, 2, 0, and . (period). Since an integer can't contain a decimal point, scanf stores –20 into j and puts the . character back.
- Conversion specification: %f. scanf reads the characters ., 3, and -. Since a floating-point number can't contain a minus sign after a digit, scanf stores 0.3 into x and puts the - character back.
- Conversion specification: %f. Lastly, scanf reads the characters -, 4, ., 0, e, 3, and ¤ (new-line). Since a floating-point number can't contain a new-line character, scanf stores -4.0×10^3 into y and puts the new-line character back.

In this example, scanf was able to match every conversion specification in the format string with an input item. Since the new-line character wasn't read, it will be left for the next call of scanf.

Ordinary Characters in Format Strings

The concept of pattern-matching can be taken one step further by writing format strings that contain ordinary characters in addition to conversion specifications. The action that `scanf` takes when it processes an ordinary character in a format string depends on whether or not it's a white-space character.

- ***White-space characters.*** When it encounters one or more consecutive white-space characters in a format string, `scanf` repeatedly reads white-space characters from the input until it reaches a non-white-space character (which is "put back"). The number of white-space characters in the format string is irrelevant; one white-space character in the format string will match any number of white-space characters in the input. (Incidentally, putting a white-space character in a format string doesn't force the input to contain white-space characters. A white-space character in a format string matches *any* number of white-space characters in the input, including none.)
- ***Other characters.*** When it encounters a non-white-space character in a format string, `scanf` compares it with the next input character. If the two characters match, `scanf` discards the input character and continues processing the format string. If the characters don't match, `scanf` puts the offending character back into the input, then aborts without further processing the format string or reading characters from the input.

  For example, suppose that the format string is `"%d/%d"`. If the input is

```
•5/•96
```

`scanf` skips the first space while looking for an integer, matches `%d` with `5`, matches `/` with `/`, skips a space while looking for another integer, and matches `%d` with `96`. On the other hand, if the input is

```
•5•/•96
```

`scanf` skips one space, matches `%d` with `5`, then attempts to match the `/` in the format string with a space in the input. There's no match, so `scanf` puts the space back; the `•/•96` characters remain to be read by the next call of `scanf`. To allow spaces after the first number, we should use the format string `"%d /%d"` instead.

Confusing `printf` with `scanf`

Although calls of `scanf` and `printf` may appear similar, there are significant differences between the two functions; ignoring these differences can be hazardous to the health of your program.

One common mistake is to put `&` in front of variables in a call of `printf`:

```
printf("%d %d\n", &i, &j);   /*** WRONG ***/
```

Fortunately, this mistake is fairly easy to spot: `printf` will display a couple of odd-looking numbers instead of the values of `i` and `j`.

Since `scanf` normally skips white-space characters when looking for data items, there's often no need for a format string to include characters other than conversion specifications. Incorrectly assuming that `scanf` format strings should resemble `printf` format strings—another common error—may cause `scanf` to behave in unexpected ways. Let's see what happens when the following call of `scanf` is executed:

```
scanf("%d, %d", &i, &j);
```

`scanf` will first look for an integer in the input, which it stores in the variable `i`. `scanf` will then try to match a comma with the next input character. If the next input character is a space, not a comma, `scanf` will terminate without reading a value for `j`.

Although `printf` format strings often end with `\n`, putting a new-line character at the end of a `scanf` format string is usually a bad idea. To `scanf`, a new-line character in a format string is equivalent to a space; both cause `scanf` to advance to the next non-white-space character. For example, if the format string is `"%d\n"`, `scanf` will skip white space, read an integer, then skip to the next non-white-space character. A format string like this can cause an interactive program to "hang" until the user enters a nonblank character.

PROGRAM

Adding Fractions

To illustrate `scanf`'s ability to match patterns, consider the problem of reading a fraction entered by the user. Fractions are customarily written in the form *numerator/denominator*. Instead of having the user enter the numerator and denominator of a fraction as separate integers, `scanf` makes it possible to read the entire fraction. The following program, which adds two fractions, illustrates this technique.

addfrac.c

```
/* Adds two fractions */

#include <stdio.h>

int main(void)
{
  int num1, denom1, num2, denom2, result_num, result_denom;

  printf("Enter first fraction: ");
  scanf("%d/%d", &num1, &denom1);

  printf("Enter second fraction: ");
  scanf("%d/%d", &num2, &denom2);

  result_num = num1 * denom2 + num2 * denom1;
```

```
  result_denom = denom1 * denom2;
  printf("The sum is %d/%d\n", result_num, result_denom);

  return 0;
}
```

A session with this program might have the following appearance:

```
Enter first fraction: 5/6
Enter second fraction: 3/4
The sum is 38/24
```

Note that the resulting fraction isn't reduced to lowest terms.

Q & A

***Q: I've seen the `%i` conversion used to read and write integers. What's the difference between `%i` and `%d`? [p. 39]**

A: In a `printf` format string, there's no difference between the two. In a `scanf` format string, however, `%d` can only match an integer written in decimal (base 10) form, while `%i` can match an integer expressed in octal (base 8), decimal, or hexadecimal (base 16). If an input number has a `0` prefix (as in `056`), `%i` treats it as an octal number; if it has a `0x` or `0X` prefix (as in `0x56`), `%i` treats it as a hex number. Using `%i` instead of `%d` to read a number can have surprising results if the user should accidentally put `0` at the beginning of the number. Because of this trap, I recommend sticking with `%d`.

octal numbers ➤7.1

hexadecimal numbers ➤7.1

Q: If `printf` treats `%` as the beginning of a conversion specification, how can I print the `%` character?

A: If `printf` encounters two consecutive `%` characters in a format string, it prints a single `%` character. For example, the statement

```
printf("Net profit: %d%%\n", profit);
```

might print

```
Net profit: 10%
```

Q: The `\t` escape is supposed to cause `printf` to advance to the next tab stop. How do I know how far apart tab stops are? [p. 41]

A: You don't. The effect of printing `\t` isn't defined in C; it depends on what your operating system does when asked to print a tab character. Tab stops are typically eight characters apart, but C makes no guarantee.

Q: What does `scanf` do if it's asked to read a number but the user enters nonnumeric input?

A: Let's look at the following example:

```
printf("Enter a number: ");
scanf("%d", &i);
```

Suppose that the user enters a valid number, followed by nonnumeric characters:

```
Enter a number: 23foo
```

In this case, `scanf` reads the `2` and the `3`, storing 23 in `i`. The remaining characters (`foo`) are left to be read by the next call of `scanf` (or some other input function). On the other hand, suppose that the input is invalid from the beginning:

```
Enter a number: foo
```

In this case, the value of `i` is undefined and `foo` is left for the next `scanf`.

What can we do about this sad state of affairs? Later, we'll see how to test whether a call of `scanf` has succeeded. If the call fails, we can have the program either terminate or try to recover, perhaps by discarding the offending input and asking the user to try again. (Ways to discard bad input are discussed in the Q&A section at the end of Chapter 22.)

detecting errors in `scanf` ➤*22.3*

Q: I don't understand how `scanf` can "put back" characters and read them again later. [p. 44]

A: As it turns out, programs don't read user input as it is typed. Instead, input is stored in a hidden buffer, to which `scanf` has access. It's easy for `scanf` to put characters back into the buffer for subsequent reading. Chapter 22 discusses input buffering in more detail.

Q: What does `scanf` do if the user puts punctuation marks (commas, for example) between numbers?

A: Let's look at a simple example. Suppose that we try to read a pair of integers using `scanf`:

```
printf("Enter two numbers: ");
scanf("%d%d", &i, &j);
```

If the user enters

```
4,28
```

`scanf` will read the 4 and store it in `i`. As it searches for the beginning of the second number, `scanf` encounters the comma. Since numbers can't begin with a comma, `scanf` returns immediately. The comma and the second number are left for the next call of `scanf`.

Of course, we can easily solve the problem by adding a comma to the format string if we're sure that the numbers will *always* be separated by a comma:

```
printf("Enter two numbers, separated by a comma: ");
scanf("%d,%d", &i, &j);
```

Exercises

Section 3.1

1. What output do the following calls of `printf` produce?

   (a) `printf("%6d,%4d", 86, 1040);`
   (b) `printf("%12.5e", 30.253);`
   (c) `printf("%.4f", 83.162);`
   (d) `printf("%-6.2g", .0000009979);`

W 2. Write calls of `printf` that display a `float` variable `x` in the following formats.

   (a) Exponential notation; left-justified in a field of size 8; one digit after the decimal point.
   (b) Exponential notation; right-justified in a field of size 10; six digits after the decimal point.
   (c) Fixed decimal notation; left-justified in a field of size 8; three digits after the decimal point.
   (d) Fixed decimal notation; right-justified in a field of size 6; no digits after the decimal point.

Section 3.2

3. For each of the following pairs of `scanf` format strings, indicate whether or not the two strings are equivalent. If they're not, show how they can be distinguished.

   (a) `"%d"` versus `" %d"`
   (b) `"%d-%d-%d"` versus `"%d -%d -%d"`
   (c) `"%f"` versus `"%f "`
   (d) `"%f,%f"` versus `"%f, %f"`

*4. Suppose that we call `scanf` as follows:

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

If the user enters

```
10.3 5 6
```

what will be the values of `i`, `x`, and `j` after the call? (Assume that `i` and `j` are `int` variables and `x` is a `float` variable.)

W *5. Suppose that we call `scanf` as follows:

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

If the user enters

```
12.3 45.6 789
```

what will be the values of `x`, `i`, and `y` after the call? (Assume that `x` and `y` are `float` variables and `i` is an `int` variable.)

6. Show how to modify the `addfrac.c` program of Section 3.2 so that the user is allowed to enter fractions that contain spaces before and after each `/` character.

---

*Starred exercises are tricky—the correct answer is usually not the obvious one. Read the question thoroughly, review the relevant section if necessary, and be careful!

Programming Projects

W 1. Write a program that accepts a date from the user in the form *mm/dd/yyyy* and then displays it in the form *yyyymmdd*:

```
Enter a date (mm/dd/yyyy): 2/17/2011
You entered the date 20110217
```

2. Write a program that formats product information entered by the user. A session with the program should look like this:

```
Enter item number: 583
Enter unit price: 13.5
Enter purchase date (mm/dd/yyyy): 10/24/2010

Item            Unit            Purchase
                Price           Date
583             $  13.50        10/24/2010
```

The item number and date should be left justified; the unit price should be right justified. Allow dollar amounts up to $9999.99. *Hint:* Use tabs to line up the columns.

W 3. Books are identified by an International Standard Book Number (ISBN). ISBNs assigned after January 1, 2007 contain 13 digits, arranged in five groups, such as 978-0-393-97950-3. (Older ISBNs use 10 digits.) The first group (the *GS1 prefix*) is currently either 978 or 979. The *group identifier* specifies the language or country of origin (for example, 0 and 1 are used in English-speaking countries). The *publisher code* identifies the publisher (393 is the code for W. W. Norton). The *item number* is assigned by the publisher to identify a specific book (97950 is the code for this book). An ISBN ends with a *check digit* that's used to verify the accuracy of the preceding digits. Write a program that breaks down an ISBN entered by the user:

```
Enter ISBN: 978-0-393-97950-3
GS1 prefix: 978
Group identifier: 0
Publisher code: 393
Item number: 97950
Check digit: 3
```

Note: The number of digits in each group may vary; you can't assume that groups have the lengths shown in this example. Test your program with actual ISBN values (usually found on the back cover of a book and on the copyright page).

4. Write a program that prompts the user to enter a telephone number in the form (xxx) xxx-xxxx and then displays the number in the form xxx.xxx.xxx:

```
Enter phone number [(xxx) xxx-xxxx]: (404) 817-6900
You entered 404.817.6900
```

5. Write a program that asks the user to enter the numbers from 1 to 16 (in any order) and then displays the numbers in a 4 by 4 arrangement, followed by the sums of the rows, columns, and diagonals:

```
Enter the numbers from 1 to 16 in any order:
16 3 2 13 5 10 11 8 9 6 7 12 4 15 14 1
```

```
16  3  2 13
 5 10 11  8
 9  6  7 12
 4 15 14  1

Row sums: 34 34 34 34
Column sums: 34 34 34 34
Diagonal sums: 34 34
```

If the row, column, and diagonal sums are all the same (as they are in this example), the numbers are said to form a ***magic square.*** The magic square shown here appears in a 1514 engraving by artist and mathematician Albrecht Dürer. (Note that the middle numbers in the last row give the date of the engraving.)

6. Modify the `addfrac.c` program of Section 3.2 so that the user enters both fractions at the same time, separated by a plus sign:

```
Enter two fractions separated by a plus sign: 5/6+3/4
The sum is 38/24
```