

The entire `digitsum.c` program is shown in Figure 4-5.

**FIGURE 4-5** Main program from `digitsum.c`

```
main()
{
    int n, dsum;

    printf("This program sums the digits in an integer.\n");
    printf("Enter a positive integer: ");
    n = GetInteger();
    dsum = 0;
    while (n > 0) {
        dsum += n % 10;
        n /= 10;
    }
    printf("The sum of the digits is %d\n", dsum);
}
```

## Infinite loops

When you use a `while` loop in a program, it is important to make sure that the condition used to control the loop will eventually become `FALSE`, so that the loop can exit. If the condition in the `while` control line always evaluates to `TRUE`, the computer will keep executing cycle after cycle without stopping. This situation is called an **infinite loop**.

As an example, suppose that you had carelessly written the `while` loop in the `digitsum.c` program with a `>=` operator in the control line instead of the correct `>` operator, as shown below:

```
while (n >= 0) {
    dsum += n % 10;
    n /= 10;
}
```



This loop will never stop running.

The loop no longer stops when `n` is reduced to 0, as it does in the correctly coded example. Instead, the computer keeps executing the body over and over and over again, with `n` equal to 0 every time.

To stop an infinite loop, you must type a special command sequence on the keyboard to interrupt the program and forcibly cause it to quit. This command sequence differs from machine to machine, and you should be sure to learn what command to use on your own computer.

## Solving the loop-and-a-half problem

The `while` loop is designed for situations in which there is some test condition that can be applied at the beginning of a repeated operation, before any of the statements in the body of the loop are executed. If the problem you are trying to solve fits this structure, the `while` loop is the perfect tool. Unfortunately, many programming problems do not fit easily into the standard `while` loop paradigm. Instead of allowing a convenient test at the beginning of the operation, some problems are structured in such a way that the test you would like to

### COMMON PITFALLS

Think carefully about the conditional expression you use in a `while` loop so that you can be sure the loop will eventually exit. A loop that never finishes is called an *infinite* loop..

write to determine if the loop is complete falls most naturally somewhere in the middle of the loop.

Consider for example, the problem of reading input data until a sentinel value appears, which was discussed in the section on “Sentinel-based loops” in Chapter 3. When expressed in English, the structure of the sentinel-based loop consists of repeating the following steps:

1. Read in a value.
2. If the value is equal to the sentinel, exit from the loop.
3. Perform whatever processing is required for that value.

Unfortunately, there is no test you can perform at the very beginning of the loop to determine whether the loop is finished. The termination condition for the loop is reached when the input value is equal to the sentinel; in order to check this condition, the program must have first read in some value. If the program has not yet read in a value, the termination condition doesn’t make sense. Before the program can make any meaningful test, it must have executed the part of the loop that reads in the input value. When a loop contains some operations that must be performed before testing for completion, it represents an instance of what programmers call the **loop-and-half problem**.

One way to solve the loop-and-a-half problem in C is to use the `break` statement, which, in addition to its use in the `switch` statement, has the effect of immediately terminating the innermost enclosing loop. By using `break`, it is possible to code the loop structure for the sentinel problem in a form that follows the natural structure of the problem:

```
while (TRUE) {  
    prompt user and read in a value  
    if (value == sentinel) break;  
    process the data value  
}
```

The initial line

```
while (TRUE)
```

needs some explanation. The `while` loop is defined so that it continues until the condition in parentheses becomes `FALSE`. The symbol `TRUE` is a constant, so it can never become `FALSE`. Thus, as far as the `while` statement itself is concerned, the loop will never terminate. The only way this program can exit from the loop is by executing the `break` statement inside it.

It is possible to code this sort of loop without using the `while (TRUE)` control line or the `break` statement. To do so, however, you must change the order of operations within the loop and request input data in two places: one before the loop begins and then again inside the loop body. When structured in this way, the paradigm for the sentinel-based loop is

```
prompt user and read in the first value  
while (value != sentinel) {  
    process the data value  
    prompt user and read in a new value  
}
```

Figure 4-6 shows how this paradigm can be used to implement the `addlist.c` program presented in Chapter 3 without using a `break` statement.

**Figure 4-6** Revised main program from `addlist.c`

```
main()
{
    int value, total;

    printf("This program adds a list of numbers.\n");
    printf("Signal end of list with a 0.\n");
    total = 0;
    printf(" ? ");
    value = GetInteger();
    while (value != 0) {
        total += value;
        printf(" ? ");
        value = GetInteger();
    }
    printf("The total is %dn", total);
}
```

Unfortunately, there are two drawbacks to using this strategy. First, the order of operations in the loop is not what most people would expect. In any English explanation of the solution strategy, the first step is to get a number and the second is to add it to the total. The `while` loop paradigm used in Figure 4-6 reverses the order of the statements within the loop and makes the program more difficult to follow. The second problem is that this paradigm requires two copies of the statements that read in a number. Duplication of code presents a serious maintenance problem because subsequent edits to one set of statements might not be made to the other. Empirical studies have shown that students who learn to solve the loop-and-a-half problem using the `break` statement form are more likely to write correct programs than those who don't.<sup>1</sup>

Despite the disadvantages, some instructors dislike using `break` to solve the loop-and-a-half problem. The principal reason for doing so is that it is easy to overuse the `break` statement in C. One way to guard against the overuse of the `break` statement is disallow its use entirely. To me, such an approach seems overly draconian. In this text, I use the `break` statement within a `while` loop only to solve the loop-and-a-half problem and not in other, more complex situations where its use is likely to obscure the program's structure.

## 1-7 The `for` statement

One of the most important control statements in C is the `for` statement, which is most often used in situations in which you want to repeat an operation a particular number of times. The general form of the `for` statement is shown in the syntax box to the right.

The operation of the `for` loop is determined by the three italicized expressions on the `for` control line: *init*, *test*, and *step*. The *init* expression indicates how the `for` loop should be initialized and usually sets the initial value of the index variable. For example, if you write

---

<sup>1</sup> The best known study corroborating this finding is "Cognitive strategies and a looping constructs: and empirical study" by Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich (Communications of the ACM, November 1983).

```
for (i = 0; ...
```

the loop will begin by setting the index variable *i* to 0. If the loop begins

```
for (i = -7; ...
```

the variable *i* will start as  $-7$ , and so on.

The *test* expression is a conditional test written exactly like the test in a while statement. As long as the test expression is TRUE, the loop continues. Thus, in the loop that has served as our canonical example up to now

```
for (i = 0; i < n; i++)
```

the loop begins with *i* equal to 0 and continues as long as *i* is less than *n*, which turns out to represent a total of *n* cycles, with *i* taking on the values 0, 1, 2, and so for the , up to the final value  $n-1$ . The loop

```
for (i = 1; i <= n; i++)
```

begins with *i* equal to 1 and continues as long as *i* is less than or equal to *n*. This loop also runs for *n* cycles, with *i* taking on the values 1, 2, and so forth, up to *n*.

The *step* expression indicates how the value of the index variable changes from cycle to cycle. The most common form of step specification is to increment the index variable using the ++ operator used in the for loops throughout Chapter 3, but its is not the only possibility. For example, one can count backward by using the - operator or count by twos by using +=2 instead of ++.

As an illustration of counting in the reverse direction, the program `liftoff.c` in Figure 4-7 counts down form 10 to 0.

### **FIGURE 4-7** `liftoff.c`

```
/*
 * File: liftoff.c
 * -----
 * simulates a countdown for a rocket launch.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Coustant: StartingCount
 * -----
 * Change this constant to use a different starting value
 * for the countdown.
 */

#define StartingCount 10

/* Main program */

main()
{
    int t;

    for (t = StartingCount; t >= 0; t--) {
        printf("%2d\n", t);
    }
}
```

#### **SYNTAX for the for statement:**

```
for (init, test, step) {
    statements
}
```

Where:

*init* is an expression evaluated to initialize the loop  
*test* is a conditional test used to determine whether  
the loop should continue, just as in the while  
statement cycle  
*statements* are the statements to be repeated

```

}
print("Liftoff!\n");
}

```

When `liftoff.c` is run, it generates the following sample run:

```

10
9
8
7
6
5
4
3
2
1
0
Liftoff!

```

The `liftoff.c` program demonstrates that any variable can be used as an index variable. In this case, the variable is called `t`, presumably because that is the traditional variable for a rocket countdown, as in “T minus 10 seconds and counting.” In any case, the index variable must be declared at the beginning of the program just like any other variable.

The expressions *init*, *test*, and *step* are each optional, but the semicolons must appear. If *init* is missing, no initialization is performed. If *test* is missing, it is assumed to be `TRUE`. If *step* is missing, no action occurs between loop cycles. Thus the control line

```
for (;)
```

is identical in operation to

```
while (TRUE)
```

## Nested for loops

As your programs become more complicated, you will often need to nest one for statement inside another. In this case, the inner `for` loop is then executed through its entire set of cycles for each iteration of the outer `for` loop. Each `for` loop must have its own index variable so that the variables do not interfere with one another.

As an example of nested for loops, consider the `timestab.c` program in Figure 4-8.

### **FIGURE 4-8** `timestab.c`

```

/*
 * File: timestab.c
 * -----
 * Generates a multiplication table where each axis
 * runs from LowerLimit to UpperLimit.
 */

#include <stdio.h>
#include "genlib.h"

/*
 * Constants
 * -----

```

```

* LowerLimit – Starting value for the table
* UpperLimit – Final value for the table
*/

#define LowerLimit 1
#define UpperLimit 10

/* Main Program */

mainn()
{
    int i, j;

    for (i = LowerLimit; i <= UpperLimit; i++) {
        for (j = LowerLimit; j <= UpperLimit; j++) {
            printf(" %4d", i * j);
        }
        printf("\n");
    }
}

```

The `timestab.c` program displays the following 10×10 multiplication table:

1	2	3	4	5	6	7	8	9	10
2	4	6	8	10	12	14	16	18	20
3	6	9	12	15	21	24	32	36	30
4	8	12	16	20	24	28	32	36	40
5	10	15	20	25	20	35	40	45	50
6	12	18	24	30	36	42	48	54	60
7	14	21	28	35	42	49	56	63	70
8	16	24	32	40	48	56	64	72	80
9	18	27	36	45	54	63	72	81	100

The outer for loop, which uses `i` as its index variable, runs through each row of the table. For each row, the inner for loop runs through each column in that row, displaying the individual entry, which is the value of `i * j` (the row number times the column number). Note that the `printf("\n")` call that advances the cursor to the next line appears in the outer loop, because this statement should only be executed once at the end of each row, and not after every value in the row.

## The relationship between `for` and `while`

As it happens, the `for` statement

```

for (init, test, step) {
    statements;
}

```

is identical in operation to the `while` statement

```

init;
while (test) {
    statements;
    step;
}

```

Even though the `for` statement can easily be rewritten using `while`, there are considerable

advantages to using the `for` statement when it makes sense to do so. With a `for` statement, all the information you need to understand exactly which cycles will be executed is contained in the control line of the statement. For example, whenever you see the statement

```
for (i = 0; i < 10; i++) {  
    ... body ...  
}
```

in a program, you know that the statements in the body of the loop will be executed 10 times, once for each of the values of `i` between 0 and 9. In the equivalent `while` loop form

```
i = 0;  
while (i < 10) {  
    ... body ...  
    i++;  
}
```

the increment operation at the bottom of the loop can easily get lost if the body is large.

## Using `for` with floating-point data


Because the *init*, *test*, and *step* components of the `for` loop can be arbitrary expressions, there is no obvious reason why the loop index in a `for` loop has to be an integer. The fact that it is possible to count from 0 to 10 by twos using the `for` loop

```
for (i = 0; i <= 10; i += 2)...
```

suggests that it might also be possible to count from 1.0 to 2.9 in increments of 0.1 by declaring `x` as a double and then using

```
for (x = 1.; x <= 2.0; x += 0.1) ...  This test may fail.
```

On some machines, this statement has the desired effect. On others, it might fail to include the last value. For example, when the `for` loop

```
for (x = 1.0; x <= 2.0; x += 0.1) {  This loop might not include the value 2.0  
    printf("%.1fn", x);  
}
```

```
1.0  
1.1  
1.2  
1.3  
1.4  
1.5  
1.6  
1.7  
1.8  
1.9
```

is run on the computer system I used to produce this text, it generates the following output:

Notice that the value 2.0, which you would expect to see from looking at the loop control line, is missing.

The problem here is that floating-point numbers are not exact. The value 0.1 is very

### COMMON PITFALLS

Be very careful when testing floating-point numbers for equality. Because floating-point numbers are only approximations, they might not behave in the same way as real numbers in mathematics. In general, it is best to avoid using a floating point variable as a `for` loop index.

close to the mathematical fraction 1/10 but is almost certainly not precisely equal to it. As 0.1 is added to the index variable *x*, the inaccuracy can accumulate to the point that, when *x* is tested against 2.0 to determine whether the loop is finished, its value may be 2.000000001 or something similar, which is not less than or equal 2.0. The condition in the for loop is therefore not satisfied, and the loop terminates after running for what seems to be one too few cycles. The best way to fix this problem is to restrict yourself to using integers as index variables in for loops. Because integers are exact, the problem never arises.

If you really want to count from 1.0 to 2.0 by increments of 0.1, you could count from 10 to 20 and then divide the index by 10:

```
for (i = 10 ; i <= 20; i++){
    x = i / 10.0;
    printf("%.1f\n", x);
}
```

This for loop correctly produces the 11 values in the sequence 1.0, 1.1, 1.2, ... , 2.0.

The same warning about comparing floating-point numbers for equality applies in many other circumstances besides the for loop. Numbers that seem as if they should be exactly equal might not be, given the limitations on the accuracy of floating-point numbers stored in a particular machine.

## SUMMARY

In Chapter 3, you looked at the process of programming from a holistic perspective that emphasized problem solving. Along the way, you learned about several control statements in an informal way. In this chapter, you were able to investigate how those statements work in more detail. You were also introduced to a new type of data called Boolean data. Although this data type contains only two values—TRUE and FALSE—being able to use Boolean data effectively is extremely important to successful programming and is well worth a little extra practice.

This chapter also introduced several new operators, and at this point it is helpful to review the precedence relationships for all the operators you have seen so far. That information is summarized in Table 4-1 the operators are listed from highest to lowest precedence.

Operator	Associativity
unary -   ++   --   !   (type cast)	right-to-left
*   /   %	left-to-right
+   -	left-to-right
<   <=   >   >=	left-to-right
==   !=	left-to-right
&&	left-to-right
	left-to-right
?:	right-to-left
=   op=	right-to-left

**TABLE 4-1**

Precedence table for operators used through Chapter 4

The important points introduced in this chapter include:



- *Simple statements* consist of an expression followed by a semicolon.
- The = used to specify assignment is an operator in C. Assignments are therefore legal expressions, which makes it possible to write *embedded* and *multiple assignments*.
- Individual statements can be collected into *compound statements*, more commonly called *blocks*.
- Control statements fall into two classes: *conditional* and *iterative*.
- The genlib library defines a data type called bool that is used to represent Boolean data. The type bool has only two values: TRUE and FALSE.
- You can generate Boolean values using the *relational operator* (<, <=, >, >=, ==, and !=) and combine them using the *logical operators* (&&, ||, and !).
- The logical operators && and || are evaluated in left-to-right order in such a way that the evaluation stops as soon as the program can determine the result. This behavior is called *short-circuit evaluation*.
- The if statement is used to express conditional execution when a section of code should be executed only in certain cases or when the program needs to choose between two alternate paths.
- The switch statement is used to express conditional execution when a problem has the following structure: in case 1, do this; in case 2; do that; and so forth.
- The while statement specifies repetition that occurs as long as some condition is met.
- The for statement specifies repetition in which some action is needed on each cycle in order to update the value of an index variable.

## REVIEW QUESTIONS

1. Is the construction
 

```
    17;
```

 a legal statement in C? Is it useful?
2. Describe the effect of the following statement, assuming that i, j, and k are declared as integer variables:
 

```
    i = (j + 4) * (k = 16);
```
3. What single statement would you write to set both x and y (which you may assume are declared to be type double) to 1.0?
4. What is meant by the term associativity? What is unusual about the associativity of assignment with respect to that of the other operators you have seen?
5. What is a block? What important fact about blocks is conveyed by the term compound statement, which is another name for the same concept?
6. What are the two classes of control statements?
7. What does it mean to say that two control statements are nested?
8. What are the two values of the data type bool?
9. What happens when a programmer tries to use the mathematical symbol for equality in a conditional expression?

10. What restriction does C place on the types of values that can be compared using the relational operators?
11. How would you write a Boolean expression to test whether the value of the integer variable *n* was in the range 0 to 9, inclusive?
12. Describe in English what the following conditional expression means:

`(x != 4) || (x != 17)`

for what values of *x* is this condition TRUE?

13. What does the term *short-circuit* evaluation mean?
14. Assuming that `myFlag` is declared as a Boolean variable, what is the problem with writing the following if statement?

`if (myFlag == TURE) ...`

15. What are the four different formats of the if statement used in this text?
16. Describe in English the general operation of the switch statement.
17. Suppose the body of a while loop contains a statement that, when executed, causes the condition for that while loop to become FALSE. Does the loop terminate immediately at that point or does it complete the current cycle?
18. Why is it important for the `digitsum.c` program in Figure 4-5 to specify that the integer is positive?
19. What is the loop-and-a-half problem? What two strategies are presented in the text for solving it?
20. What is the purpose of each of the three expressions that appear in the control line of a for statement?
21. What for loop control line would you use in each of the following situations:
  - a) Counting form 1 to 100.
  - b) Counting by sevens starting at 0 until the number has more than two digits.
  - c) Counting backward by twos form 100 to 0.
22. Why is it best to avoid using a floating-point variable as the index variable in a for loop?

## **PROGRAMMING EXERCISES**

1. As a way to pass the time on long bus trips, young people growing up in the United States have been known to sing the following rather repetitive song:

99 bottles of beer on the wall.  
 99 bottles of beer.  
 You take one down, pass it around.  
 98 bottles of beer on the wall.

98 bottole of beer on the wall...

Any way, you get the idea. Write a C program to generate the lyrics to this song. (Since you probably never actually finished singing this song, you should decide how you want it to end.) In testing your program, it would make sense to use some constant

other than 99 as the initial number of bottles.

2. While we're on the subject of silly songs, another old standby is "This old Man," for which the first verse is

This old man, he played 1.  
He played knick-knack on my thumb.  
With a knick-knack, paddy-whack,  
Give your dog a bone.  
This old man came rolling home.

Each subsequent verse is the same, except for the number and the rhyming word at the end of the second line, which gets replaced as follows:

2—shoe 5—hive 8—pate  
3—knee 6—sticks 9—spine  
4—door 7—heaven 10—shin

Write a program to display all 10 verses of this song.

3. Write a program that reads in a positive integer  $N$  and then calculates and displays the sum of the first  $N$  odd integers. For example, if  $N$  is 4, your program should display the value 16, which is  $1 + 3 + 5 + 7$ .

4. Why is *everything either at sixes or at sevens*?

— Gilbert and Sullivan, H.M.S. Pinafore, 1878

Write a program that displays the integers between 1 and 100 that are divisible by either 6 or 7.

5. Repeat exercise 4, but this time have your program display only those numbers that are divisible by 6 or 7 but not both.
6. Rewrite the `liftoff.c` program given in Figure 4-7 so that it uses a while loop instead of a for loop.
7. Rewrite the `digitsum.c` program given in Figure 4-5 so that instead of adding the digits in the number, it generates the number that has the same digits in the reverse order, as

```
This program reverses the digits in an integer.  
Enter a positive integer: 1729 ↵  
The reversed number is 9271
```

illustrated by this sample run:

8. In mathematics, there is a famous sequence of numbers called the Fibonacci sequence after the thirteenth-century Italian mathematician Leonardo Fibonacci. The first two terms in this sequence are 0 and 1, and every subsequent term is the sum of the preceding two. Thus the first several numbers in the Fibonacci sequence are as follows:

$$F_0 = 0$$

$$\begin{aligned}
 F_1 &= 1 \\
 F_2 &= 1 \quad (0 + 1) \\
 F_3 &= 2 \quad (1 + 1) \\
 F_4 &= 3 \quad (1 + 2) \\
 F_5 &= 5 \quad (2 + 3) \\
 F_6 &= 8 \quad (3 + 5)
 \end{aligned}$$

Write a program to display the values in this sequence from  $F_0$  through  $F_{15}$ . Make sure the value line up as shown in the following sample run:

```

This program lists the Fibonacci sequence.
F(1) = 0
F(2) = 1
F(2) = 1
F(3) = 2
F(4) = 3
F(5) = 5
F(6) = 8
F(7) = 13
F(8) = 21
F(9) = 34
F(10) = 55
F(11) = 89
F(12) = 144
F(13) = 233
F(14) = 377
F(15) = 610

```

9. Modify the program in the preceding exercise so that instead of specifying the index of the final term, the program displays those terms in the Fibonacci sequence that are less than 10,000.
10. Write a program to display the following diagram on the screen. The number of rows in the figure should be a `#define` constant, which has the value 8 for this sample run:

```

*
**
***
****
*****
          *
         ***
        *****
       ********
      *********
     ***********
    *************
   *************
  *************

```

11. Modify the program you wrote in exercise 10 so that it generates a different triangle. In this triangle, each line contains two more points than the previous line does, and the point of the triangle faces upward, as follows: