Chapter 6

More Repetition and Decisions

This chapter continues the discussion of control statements, which are used to alter the normal top-to-bottom execution of the statements in a program. As each new kind of statement is covered, its corresponding flow diagram will be shown. We present the syntax, flowcharts, and examples of use for these statements.

There are three kinds of loop statements in C, each of which is used to repeat a block of code. The `while` loop was introduced in Chapter 3. This chapter presents two additional loop statements, `for` and `do...while`. We also discuss various ways that loops can be used.

Conditional control statements are used to determine whether to execute or skip certain blocks of code. The C language has three kinds of conditional control statements: the `if...else` statement and the simple `if` statement without an `else` clause, which are introduced in Chapter 3, and a multibranched conditional statement, the `switch` statement. In this chapter we examine the `switch` statement and a new way to use the simple `if` statement.

We briefly cover the `break` and `continue` statements, which interrupt the normal flow of control by transferring control from inside a block of code to its end or its beginning. C also supports the `goto` control statement. However, we neither explain it nor illustrate its use, because it is almost never needed in C and its unrestricted nature makes it error prone. Using a `goto` statement is considered very poor programming practice because it leads to programs that are hard to both debug and maintain.

6.1 New Loops

6.1.1 The `for` Loop

The `for` loop in C implements the same control pattern as the `while` loop; anything that can be done by a `while` loop can be done in the same way using a `for` loop and vice versa. A `for` loop could be thought of as a shorter, integrated notation for a `while` loop that brings together the initialization(s), the loop test, and the update step in a single header at the top of the loop, as shown in Figure 6.1.

In this simple `while` statement and its corresponding `for` statement, `k` is used as a counter to keep track of the number of loop repetitions. When `k` reaches `10`, the body will have been executed 10 times and the loop will exit. Following the loop, the values of `k` and `sum` will be printed. The output will be `10 45`.

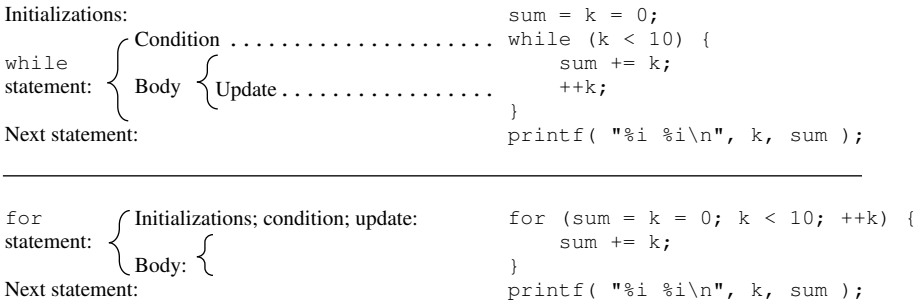


Figure 6.1. The `for` statement is a shorthand form of a `while` statement.

```
/* ---------------------------------
**   Counting with a for loop.
*/
#include <stdio.h>
int main( void )
{
    int k;
    for (k = 0; k < 5; ++k) {
        printf( " %i \n", k );
    }
    puts( "-----\n" );
}
```

The output is

```
 0
 1
 2
 3
 4
-----
```

Figure 6.2. A simple program with a `for` statement.

Thus, `for` is not really a necessary part of the language. However, it probably is more widely used than either of the other loops because it is very convenient and it captures the nature of a loop that is controlled by a counter. After gaining some experience, most programmers prefer using the `for` statement for many kinds of loops, especially for **counted loops** like that in Figure 6.1. An extremely simple program containing a `for` loop is shown, with its output, in Figure 6.2. This illustrates how a loop should be laid out in the context of a program.

The syntax and flow diagram for `for`. The `for` statement has a header consisting of the keyword `for`, followed by a parenthesized list of three expressions separated by semicolons: an initialization expression, a

The general form of a `for` flow diagram is shown here. Control passes through the loop in the order indicated by the arrows. Note that this order is not the same as the order in which the parts of the loop are written in the code.

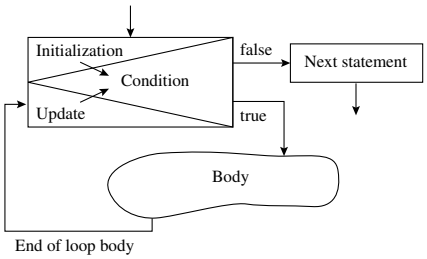


Figure 6.3. A flow diagram for the `for` statement.

condition, and an update expression. All three expressions can be arbitrarily complicated. Usually, however, the first part is a single assignment, the second is a comparison, and the third is an increment expression, as shown in in Figure 6.1.

Since `for` and `while` implement very similar control structures, a `for` loop can be diagrammed in the same manner as a `while` loop: using separate boxes for the initialization, test, and update steps. However, there is a more compact, single-box diagram that combines these pieces into one multipart control box with a section for each part of the loop header. (See Figure 6.3). Control enters through the initialization section at the top, then goes through the upper diagonal line into the condition section on the right. If the condition is true, control leaves through the lower exit, going through the boxes in the body of the loop, coming back into the update section of the `for` box, and finally, going through the lower diagonal line into the test again. If the test is false, control leaves the loop through the upper exit. The flow diagrams in Figure 6.4 compare the equivalent `while` and `for` loops of Figure 6.1. Both the statement syntax and the diagram for the `for` loop are more concise—fewer parts are needed and those parts are more tightly organized—than in a corresponding `while` loop.

Initializing the loop variable. An initialization statement must be written before a `while` loop so that the loop variable has some meaningful value before the loop test is performed the first time. In a `for` loop, this initialization is written as the first expression in the loop header. When the `for` loop is executed, this expression will be evaluated only once, before the loop begins.

The `,` (comma operator) can be used in a `for` loop to permit more than one item to be initialized or updated. For example, the loop in Figure 6.1 could be written with separate initializations for `sum` and `k`:

```
for (sum = 0, k = 0; k < 10; ++k) ...
```

Technically, both assignment and comma are operators. They build expressions, not complete statements, so we are permitted to use either or both any time the language syntax calls for an expression.

The flow diagram for the `while` loop in Figure 6.1 is shown on the left and the diagram for the `for` loop is on the right. Note that the parts of the two loops are executed in exactly the same order, as control flows into, through, and out of the boxes along the paths indicated by the arrows.

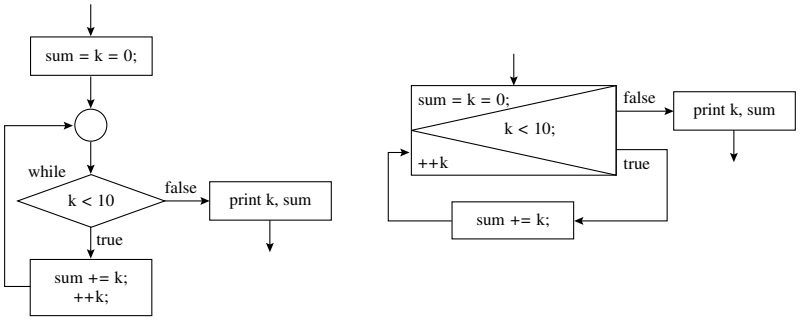


Figure 6.4. Diagrams of corresponding `while` and `for` loops.

The loop test and exit. The condition in a `for` statement obeys the same rules and has the same semantics as that of a `while` condition. It is executed after the initializations, when the loop is first entered. On subsequent trips through the loop, it is executed after the update. If the condition does computations or causes side effects, such as input or output, those effects will happen each time around the loop. (This programming style is not recommended. It may be concise, but it sacrifices clarity and does not work well with some on-line debuggers.) Finally, note that the body of the loop will not be executed at all if the condition is false the first time it is tested.

Updating the loop variable. The update expression is evaluated every time around the loop, after the loop body and before re-evaluating the loop condition. Note the mismatch between *where* the update is written on the page and *when* it happens. It is written after the condition and before the body, but it happens after the body and before the condition. Beginning programmers sometimes are confused by this inverted order. Let the flow diagram be your guide to the proper order of evaluation.

In a counted `for` loop with loop variable `k`, the update expression often is as simple as `k++` or `++k`. A lot of confusion surrounds the question of which of these is correct and why. The answer is straightforward: As long as the update section is just a simple increment or decrement expression, it does not matter whether you use the prefix or postfix form of the operation. Whichever way it is written, the loop variable will be increased (or decreased) by 1 after the loop body and before the condition is retested.

The loop body. The loop body is executed after the condition and before the update expression. In Figure 6.1, compare the `for` loop body, which contains only one statement, with the two-statement body of the `while` loop. The update expression must be in the body of a `while` loop but becomes part of the loop header of a `for` loop, shortening the body of the loop by at least one statement. Often, this reduces

The general form of a `do...while` flow diagram is shown on the left. Control passes through the loop body before reaching the test at the end of the loop. On the right is a diagram of a summing loop, equivalent to the loops in Figure 6.4 that use `while` and `for` statements.

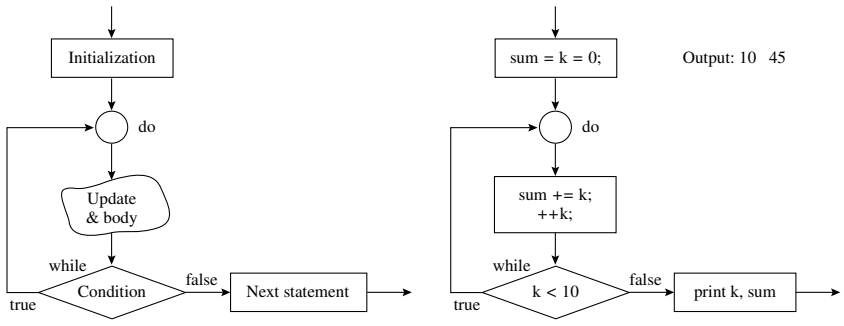


Figure 6.5. A flow diagram for the `do...while` statement.

the body of a `for` loop to a single statement, which permits us to write the entire loop on one line, without braces, like this:

```
for (sum = 0, k = 0; k < 10; ++k) sum += k;
```

6.1.2 The `do...while` Loop

The `do...while` loop implements a different control pattern than the `while` and `for` loops. The body of a `do...while` loop is executed at least once; this is illustrated by the flow diagram in Figure 6.5. The condition, written after the keyword `while`, is tested after executing the loop body. Therefore, unlike the `while` and `for` loops, the body of a `do...while` loop can initialize the variables used in the test. This makes the `do...while` loop useful for repeating a process, as shown in Figure 6.11.

6.1.3 Other Control Statements

C supports four statements whose purpose is to transfer control to another part of the program. Two of these, `break` and `continue`, are used in conjunction with loops to create additional structured control patterns. The third, `return`, is used at the end of a function definition. The fourth, `goto`, has little or no place in modern programming and should not be used. Neither `break` nor `continue` is necessary in C; any program can be written without them. However, when used skillfully, a `break` or `continue` statement can simplify program logic and shorten the code. This is highly desirable because decreasing complexity decreases errors.

A `continue` statement takes control directly to the top of at `while` or `do` loop.

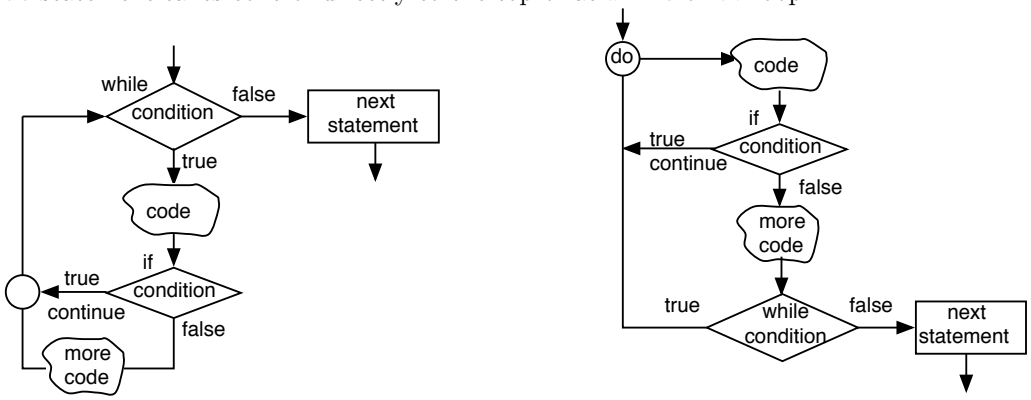


Figure 6.6. Using continue with while and do.

The `break` statement. The `break` statement interrupts the normal flow of control by transferring control from inside a loop or a `switch`[1] to the statement after its end. If the `break` is within a `for` loop, execution continues with the increment or update step. In a `while` loop, execution continues with the loop test. In a `do...while` loop, execution continues with the first statement in the body of the loop.

The `break` statement is diagrammed as an arrow because it interrupts the normal flow of control. An `if` statement whose true clause is a `break` statement is commonly used inside a `while` loop or a `for` loop. We will call this combination an `if...break` statement; it is diagrammed as an arrow that leaves the normal control path at an `if` statement inside the loop and rejoins the normal path at the statement after the loop. (See Figure 6.20.) This is discussed further in Section 6.2.7.

The `continue` statement. The `continue` statement interrupts the normal flow of control by transferring control from inside a loop to its beginning. If it is within a `for` loop, execution continues with the increment step, as shown on the right in Figure 6.6. In a `while` or `do...while` loop, execution continues with the loop test. The `continue` statement is also diagrammed as an arrow. An `if...continue` statement is diagrammed as an arrow that leaves the normal control path at the `if` statement and rejoins the loop at the top.

The `continue` statement is not commonly used but occasionally can be helpful in simplifying the logic when one control structure is nested within another, as when a loop contains a `switch` statement. This control pattern is used in menu processing and will be illustrated in Chapter 12 where `continue` is used within the switch to handle an invalid menu selection.

The `return` statement. By now, the `return` statement should be familiar to the reader; it has been used in every program example since Chapter 2. Here, we restate the rules for using `return`, and discuss a few

[1] Section 6.3 deals with the `switch` statement.

A `continue` statement takes control to the increment step in the header of a `for` loop.

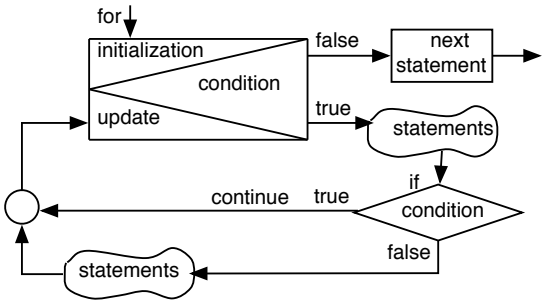


Figure 6.7. Using continue with for.

usage options.

1. Executing a `return` statement causes control to leave a function immediately.
2. A `void` function may contain a simple `return` statement, in which the keyword `return` is followed immediately by a semicolon. If this statement is omitted, the function will return when control reaches the final closing brace.
3. Every non-void function must contain a `return` statement with an expression between the keyword `return` and the semicolon. The type of this expression must match the declared return type or be coercible to that type[2]. When control reaches this statement, the expression will be evaluated and its value will be returned.
4. It is syntactically legal to have more than one `return` statement in a function. However, this is generally considered to be poor style. There is genuine debate, however, about whether multiple `return`s are sometimes appropriate. On one side, many teachers and employers prohibit this practice because it breaks the "one way in – one way out" design rule. On the other side, some experts believe that code simplicity is more important than "one in – one out" construction, and permit use of multiple return statements in a few situations where the extra return statements significantly shorten or reduce the nesting level of the code.

6.1.4 Defective Loops

If a loop does not update its loop variable, it will loop forever and be called an **infinite loop**. The most common cause of an infinite loop is that the programmer simply forgets to write a statement that updates the loop variable. Another source of error is missing or misplaced punctuation. The body of a loop starts at the right parenthesis that ends the condition. If the next character is a left curly bracket, the loop extends

[2]Type coercion is discussed in Chapter 7.

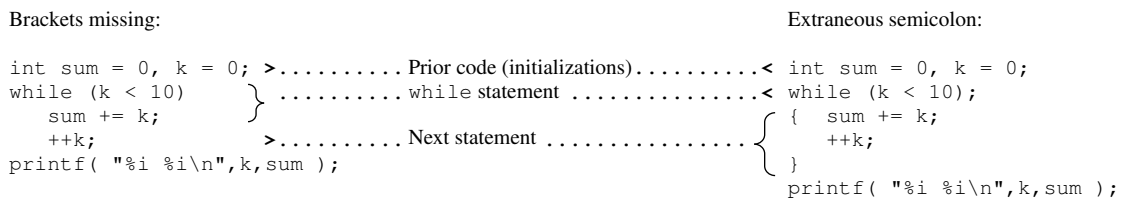


Figure 6.8. No update and no exit: Two defective loops.

to the matching right curly bracket. Otherwise, the body consists of the single statement that follows the condition and the loop ends at the first semicolon. Figure 6.8 shows two loops that are infinite in nature because they do not update their loop variable. The loop on the left is like the `while` loop in Figure 6.1 except that the braces are missing. Because of this, the loop ends after the statement `sum += k;` and does not include the `++k` statement. This kind of omission will not be caught by the compiler; it is perfectly legal to write a loop with no braces and with only one statement in its body. The compiler does not check that the programmer has updated the loop variable.[3]

The loop on the right has an extraneous semicolon following the loop condition. This semicolon ends the loop, resulting in a null or empty loop body, which is legal in C. Since `k`, the loop variable, is not updated before this semicolon, it never changes. (The bracketed series of statements that follows the semicolon is outside the loop.) One way to avoid this kind of error is to write the left curly bracket that begins the loop body *on the same line* as the keyword `while`. The left bracket is a visible reminder that the line should not end in a semicolon.

6.2 Applications of Loops

Knowing the correct syntax for writing a loop is important but only part of what a programmer needs to understand. This chapter and later ones present several common applications of loops and paradigms for their implementation in C. These applications include

Sentinel loops. Introduced in Figure 6.23 and discussed in Section 6.2.1.

Query loops. Presented in Section 6.2.2.

Counted loops. Introduced in Figure 3.14 and treated in greater depth in Section 6.2.3.

Input validation loops. Introduced in Figure 3.15 and revisited in Section 6.2.4.

Nested loops. : Presented in Section 6.2.5.

[3] A mathematical technique, called a *loop invariant*, can be used to find loops that do not accomplish the design goals. This technique is beyond the scope of an introductory textbook.

Delay loops. Presented in Section 6.2.6.

Flexible-exit loops. Presented in Section 6.2.7.

Counted sentinel loops. Based on the flexible-exit loop, this pattern is presented in Section 6.2.8.

Search loops. Introduced in Section 6.2.9 and illustrated in several later chapters.

Table processing loops are introduced in Chapter 13.

End-of-file loops are introduced in Chapter 14.

6.2.1 Sentinel Loops

A **sentinel loop** keeps reading, inspecting, and processing data values until it comes across a predefined value that the programmer has designated to mean "end of data." Looping stops when the program recognizes this value, which is called a **sentinel value**, because it stands guard at the end of the data.

The value used as a sentinel depends on the application; to choose an appropriate sentinel value, the programmer must understand the nature of the data. Most functions that process strings use the null character as a sentinel value. Loops that read and process input data often use the newline character as a sentinel. If the data values are integers and a program processes only nonzero data values, then 0 can be used as a sentinel. If all data values are nonnegative, then -1 can be used as the sentinel. If every integer is admissible, the value `INT_MAX` (the largest representable integer) often is used as a sentinel.

In all cases, a sentinel value must be of the same data type as the ordinary data values being processed, because it must be stored in the same type of variable or read using the same conversion specifier in a format string. Also, a sentinel must not be contained in the set of legal data values because it must be an unambiguous signal that there are no more data sets to process.

A sentinel loop is used when the number of data items to be processed varies from session to session, depending on the user's needs, and cannot be known ahead of time. This happens in many contexts, including

- When reading a series of input data sets.
- When processing string data.[4]
- When processing data that are stored in an array or a list in which the sentinel value is stored at the end of the data[5].

Input-controlled sentinel loops are the only kind that we are ready to examine at this time[6]. An input-controlled sentinel program reads and processes input values; the sentinel value must be entered from the keyboard as a signal that there is no more input. The loop compares each input value to the sentinel and ends the input process if a match is found. Such programs follow this general form:

[4]Strings will be introduced in Chapter 12.
[5]Arrays will be introduced in Chapter 10 and lists in Chapter 21.
[6]The other sentinel loops will be introduced in Chapters 10, 12, and 21.

```
/* Comments that explain the purpose of the program. */
#include commands.
#define the sentinel value.

int main( void )
{
    Declaration of input variable and others.
    Output statement that identifies the program.

    Use scanf() to initialize the input variable.
    while (input != sentinel value)  {
         Process the input data.
         Prompt for and read another input.
    }
    Print program results and termination comment.
    return 0;
}
```

The user prompts must give clear instructions about the sentinel value. Otherwise, the user will be unable to end the loop. For example, consider the cash register program in Figure 6.9, which uses a simple input-controlled sentinel loop. The initial prompt gives clear instructions about how to end the processing loop. Sentinel loops are implemented with `while` or `for(;;)` statements, rather than a `do...while` statement, because it is important not to try to process the sentinel value. The `do...while` statement processes every value before making the loop test, whereas the `while` loop makes the test before processing the value. Thus, a `while` loop can watch for the sentinel value and leave the loop when it appears without processing it. Figures 6.23 and 6.24 show two ways that a `for` statement can be used to implement a sentinel loop.

6.2.2 Query Loops

If the loop variable is either initialized or updated by `scanf()` or some other input function, we say that it is an **input-controlled loop**. Such loops are very important because they allow a program to respond to the real-world environment. There are several variations on this theme, including query loops, sentinel loops (Section 6.2.1), and input validation loops (Section 6.2.4).

A useful interactive technique, the **repeat query**, is introduced in Figure 6.10 and used in Figure 6.11. (It will be refined, later, in Chapter 8.) To develop and debug a program, the programmer must test it with several sets of input so that its performance with different kinds of data can be checked. We use a repeat query loop to automate this process of rerunning a program. Until now, you have needed to execute a program once for each line in your test plan. After each run, the output must be captured and printed. At best, the process is awkward. At worst, the programmer is tempted to shortcut the testing process. A typical program with a query loop follows the general form shown in Figure 6.10.

The testing process can be simplified by writing a function that processes one line of the test plan. The main program contains a `do...while` loop that calls the function once and asks whether the user wishes to do it again. This provides a simple, convenient way to let the user decide whether to continue running the program or quit, rather than restarting it every time. Furthermore, all the output for all of the tests ends up in one place at one time. The basic technique is illustrated in Figures 6.10 through 6.12.

Compute the sum of a series of prices entered by the user.

```
#include <stdio.h>
#define SENTINEL  0

int main ( void )
{
    double input;                               /* Price of one item.  */
    double sum = 0;                             /* Total price of all items.  */
    printf( " Cash Register Program.\n Enter the prices; use 0 to quit.\n> " );

    scanf ( "%lg", &input );      /* Read first price. */
    while (input != SENTINEL) {   /* Sentinel value is 0 */
        sum += input;             /* Process the input. */
        printf( "> " );           /* Get next input. */
        scanf( "%lg", &input );
    }

    printf( " Your order costs $%g\n", sum );
    return 0;
}
```

Output from a sample run:

```
 Cash Register Program.
 Enter the prices; use 0 to quit.
> 3.10
> 29.98
> 2.34
> 0
 Your order costs $35.42
```

Figure 6.9. A cash register program.

Notes on Figure 6.10. Form of a query loop.

- Programs that use this technique will have a few statements at the beginning of `main()` that may open files, clear the screen, or print output headings.
- At the end are statements to print final results and any closing message.
- In between is a `do...while` loop. The loop body consists entirely of a call on a function that performs the work of the program, followed by a prompt to ask the user whether or not to repeat the process. The response is read and immediately tested. If the user enters the code 0,[7] control leaves the loop. If 1 (or an erroneous response) is entered, the process is repeated.

[7] We use a 1 or 0 response here because it is simple. In Chapter 8, we show how to process y or n responses.

Many programs perform a process repeatedly, until the user asks to stop. This is the general form of such a program. The process is performed by a function called from the main loop.

```
/* Comments that explain the purpose of the program. */
#include and #define commands.

int main( void )
{
    Declaration of variable for query response;

    Output statement that identifies the program;
    do {
        Process one data set or call a function to do so;
        Ask the user whether to continue (1) or quit (0);
        Read the response;
    } while (response != 0);

    Print program results and termination comment;
}
```

Figure 6.10. Form of a query loop.

Figure 6.13 shows a diagram of a query loop (on the left) used to repeat a calculation (the function diagrammed on the right). The dotted lines show how control goes from the function call to the entry point of the function and from the function return back to the call box.

Notes on Figure 6.11. Repeating a calculation. We put almost all of the program's code into a function called `work()`. The main program contains only greeting and closing messages, a loop that calls the `work()` function to do all the work, and the input statements needed to control the loop. Figure 6.13 is a flow diagram of this program.

First box: prototypes for the two programmer-defined functions. Two programmer-defined functions that follow the main program in the source code file are shown in Figure 6.12. The `work()` function is called from `main()`. It, in turn, calls `drop()`.

Outer box: the main process loop.

- Compare this version of the program to the versions in Figures 3.10 and 5.15. The input, calculations, and output have been removed from `main()` and placed in a separate function, named `work()`. These lines have been replaced by a loop that will call the `work()` function repeatedly (inner box), processing several data sets. This keeps the logic of `main()` simple and easy to follow.
- We use a loop because we expect to process a series of inputs. Since we do not know how many will be needed ahead of time, we ask the user to tell us what to do after each loop repetition.
- We prompt for a 'y' to do the process again or a 'n' to quit; the response is read into the variable `do_it_again`.
- As long as the values we read for `do_it_again` are not 'n', we call the `work()` function to read another input value and perform the computation and output processing. If the user's input is an error, that is, it is neither a 'y' nor an 'n' , this program continues; it quits only if the user enters 'n'.

This main program consists of statements to print the program titles and a loop that will repeatedly call the `work()` function, given in Figure 6.12, until the user asks to quit. This main program can be used to repeat any process by changing the titles and the `work()` function.

```
/* ------------------------------------------------------------------------
** Determine the time it takes for a grapefruit to hit the ground when it
** is dropped, with no initial velocity, from a helicopter hovering
** at height h.  Also determine the velocity of the fruit at impact.
* ------------------------------------------------------------------------ */
#include <stdio.h>

void work( void );
double drop( double height );

int main( void )
{
    char do_it_again;     /* repeat-or-stop switch */

    puts( "\n Calculate the time it would take for a grapefruit\n"
          " to fall from a helicopter at a given height.\n" );

    do { work();
        printf( " \n Enter 'y' to continue or 'n' to quit: " );
        scanf( " %c", &do_it_again );
    } while (do_it_again != 'n');

    return 0;
}
```

Figure 6.11. Repeating a calculation.

Notes on Figure 6.12: The `work()` and `drop()` functions.

Background. Most of the code from the `main()` program in Figure 5.15 has been moved into the `work()` function. This reduces the complexity of `main()` and makes it easy to repeat the gravity calculations with several inputs. The `work()` function contains all the declarations and code that relate to the computation. The `main()` program contains only start-up code, termination code, and the loop that calls the `work()` function.

The flow of control. The function call in `main()` sends control into the `work()` function, where we read one input and calculate the time of fall by calling the `drop()` function. After returning from `drop()`, we calculate the terminal velocity and print the time and velocity. Then we return to `main()`. In the flow

The `work()` function is called from the program in Figure 6.11. The `drop()` function is a more concise version of the one in Figure 5.18.

```
/* ------------------------------------------------------ */
/* Perform one gravity calculation and print the results. */
void
work( void )
{
    double h;           /* height of fall (m) */
    double t;           /* time of fall (s) */
    double v;           /* terminal velocity (m/s) */

    printf( " Enter height of helicopter (meters): " );
    scanf( "%lg", &h );

    t = drop( h );      /* Call drop with the argument h. */
    v = GRAVITY * t;    /* velocity of grapefruit at impact */
    printf( "    Time of fall = %g seconds\n", t );
    printf( "    Velocity of the object = %g m/s\n", v );
}

/* ------------------------------------------------------ */
/* Calculate time of fall from a given height. ---------- */
double
drop( double height )
{
    double answer = 0;
    if (height > 0) answer = sqrt( 2 * height / GRAVITY );
    return answer;
}
```

Figure 6.12. The `work()` and `drop()` functions.

diagram (Figure 6.13), these shifts of control are represented by dotted lines.

The output. Lines printed by the `main()` program are intermixed with lines from the `work()` function. Here is a sample dialog:

```
Calculate the time it would take for a grapefruit
to fall from a helicopter at a given height.

Enter height of helicopter (meters): 20
    Time of fall = 2.01962 seconds
    Velocity of the object = 19.8057 m/s

Enter 'y' to continue or 'n' to quit:  1
Enter height of helicopter (meters): 906.5
```

This is a flow diagram of the program in Figures 6.11 and 6.13. The dotted lines show how the control sequence is interrupted when the function call sends control to the beginning of the function. Control flows through the function then returns via the lower dotted line to the box from which it was called.

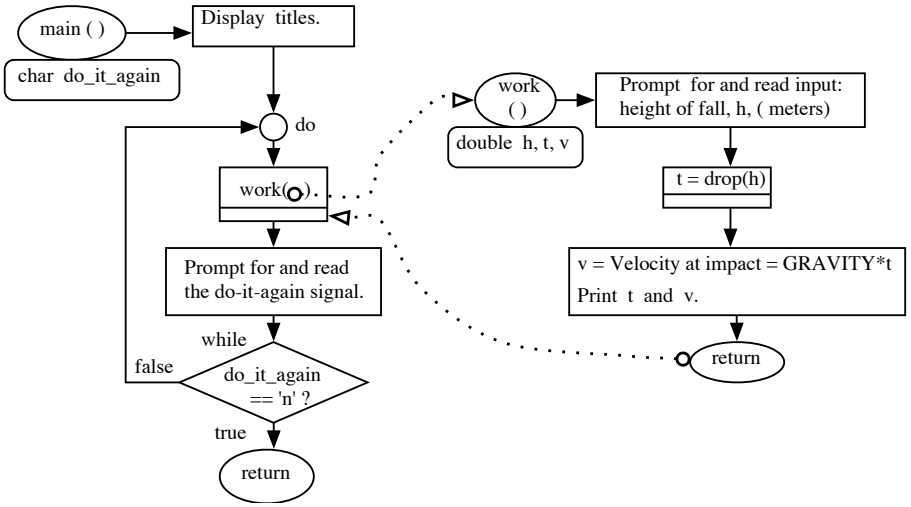


Figure 6.13. Flow diagram for repeating a process.

```
    Time of fall = 13.597 seconds
    Velocity of the object = 133.34 m/s

Enter 'y' to continue or 'n' to quit:  2
Enter height of helicopter (meters):  2000.5
    Time of fall = 20.1987 seconds
    Velocity of the object = 198.082 m/s

Enter 'y' to continue or 'n' to quit:  0
```

As you begin to write programs, incorporate a processing loop into each one. It then will be convenient for you to test your code on a variety of inputs and demonstrate that it works correctly under all circumstances.

6.2.3 Counted Loops

Many loops are controlled by a counter. In such a **counted loop**, an initialization statement at the top of the loop usually sets some variable, say `k`, to 0 or 1. The update statement increments `k`, and the loop test asks whether `k` has reached or exceeded some goal value, `N`. To calculate the **trip count**, that is, the number of times the loop body will be executed,

- Let the initial value of the loop variable be I and the goal value be N.

This program computes the sum of the first N terms of the series $1/n$.

```
#include <stdio.h>
#define N  10

double f( double x) { return 1.0 / x; }    /* The function to sum. */

int main( void )
{
    int n;                  /* Loop counter */
    double sum;             /* Accumulator  */

    printf( "\n Summing 1/n where n goes from 1 to %i \n", N );

    sum = 0;                        /* Start accumulator at 0. */
    for (n = 1; n <= N; ++n) {      /* Sum series from 1 to N. */
        sum += f( n );
    }

    printf( " The sum is %g.\n", sum );

    return 0;
}
```

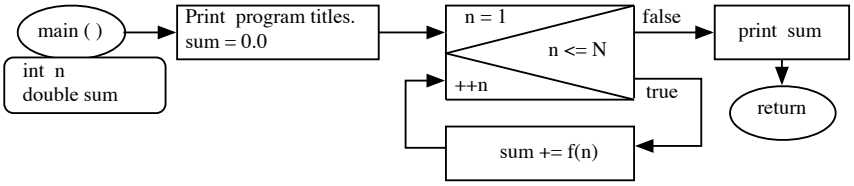


Figure 6.14. Summing a series.

- If the loop test has the form `k < N`, the trip count is $N - I$.
- If the loop test has the form `k <= N`, the trip count is $N - I + 1$.

The loops diagrammed in both Figures 6.4 and 6.14 are counted loops. In the first example, the loop variable is `k`, the initial value is 0, and the test is `k<10`; so this loop will be executed 10 times, with `k` taking on the values $0 \ldots 9$, successively. If an initial value of 1 and a loop test of `k <= N` were used, the loop body still would be executed 10 times, but the sum would be different, because `k` would have the values $1 \ldots 10$. Both patterns of loop control are common. A frequent source of program error is using the wrong initial value or the wrong comparison operator in the loop test.

A summation loop. Our next example, in Figure 6.14, shows how a counted loop can be used to sum a series of numbers. This example demonstrates a typical programming pattern in which two variables are used: a *counter* (to record the number of repetitions) and an *accumulator* (to hold the sum as it is accumulated). Both variables are initialized before the loop begins and both are changed on every trip through the loop. The flow diagram for this program (shown following the code) is very similar to that in Figure 6.4, since the same loop structure is being used.

Notes on Figure 6.14. SumUp.

First box: the function. The same program could be used to sum a different function by simply changing the expression in the `return` statement here and the `printf()` statement in `main()`.

Second box: the declarations.

- The type `int` is appropriate for counters such as `n`, that are used to count the repetitions of the loop.
- An accumulator is a numeric variable used to accumulate the sum of a series of values. These might be computed values, as here, or input values, illustrated in Figure 6.9. We use a variable of type `double` for the accumulator because it will be used to compute the total of various fractions.

Third box: the loop.

- Before entering any loop, the variables used in the loop must be initialized. We set `n = 1` because we want to sum the series from 1 to 10. We set `sum = 0-->`.
- The `for` statement is ideally suited for counted loops because the loop header organizes all the information about where the counter starts and stops and how it changes at each step. Each time around the loop, we add `1.0/n` to the sum. This loop starts with `n` equal to 1 and ends when `n` reaches 11. Therefore, the loop body will be executed $11 - 1 = 10$ times, summing the fractions `1.0/1 ... 1.0/10`.
- Note that `n` will have the value 1 (not 0 or 2) the first time we compute `1.0/n`. This is important. First, we do not want to start computing the series at the wrong point. Second, we need to be careful to avoid dividing by 0.
- We are permitted to divide the `double` value 1.0 by the integer value `n`. The result is a fractional value of type `double`. It is important that the constant 1.0 be used rather than 1 in this expression, because the latter will give an incorrect result. The reason for this is discussed in Chapter 7.

Fourth box: the output.

- We use the format `%g` to print the `double` value. Up to six digits (C's default precision) will be printed with the result rounded to the sixth place. Trailing 0's, if any, are suppressed after the decimal point.
- The output from this program is

```
Summing 1/n where n goes from 1 to 10
The sum is 2.92897.
```

6.2.4 Input Validation Loops

Figure 6.15 contains a validation loop based on a `while` statement. It provides good user feedback but is long and requires duplicating lines of code. We can write a shorter, simpler validation loop that uses a `do...while`

This input validation loop is a fragment of the miles-per-hour program in Figure 3.15. Some output from the loop and the following `printf()` statement follow.

```
printf( " Duration of trip in hours and minutes: " );
scanf( "%lg%lg", &hours, &minutes );
hours = hours + ( minutes / 60 );
while (hours < 0) {
    printf( " Please re-enter; time must be >= 0: " );
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + ( minutes / 60 );
}
```

Output:

```
Duration of trip in hours and minutes: -148 43
Please re-enter; time must be >= 0: 1 -70
Please re-enter; time must be >= 0: 148 -17
Average speed was 1.94291
```

Figure 6.15. Input validation using a `while` statement.

statement. However, this kind of validation loop has a severe defect: It gives the same prompt for the initial input and for re-entry after an error. This is a human-engineering issue. The error may go unnoticed if the program does not give distinctive feedback. A third kind of validation loop can be written with `for` and `if...break` statements that combines the advantages of the other two forms; it avoids duplication of code and provides informative feedback when the user makes an error (see Figure 6.15).

Notes on Figure 6.15: Input validation using a `while` statement.

1. The input prompt `scanf()` and calculation statements are written before the loop and again in the body of this loop. This duplication is undesirable because both sets of statements must be edited every time the prompt or input changes. The duplication is unavoidable, though, because the loop variable, `hours`, must be given a value before the `while` test at the top of the loop. It then must be given a new value within the loop.
2. The `while` test checks whether the value of `hours` is within the legal range. If not, we enter the body of the loop, print an error comment, and prompt for and read another input. Then control returns to the top of the loop to test the data again.
3. Since only a legal input will get the user out of this loop, an informative prompt is very important. If the user is not sure what data values are legal, the program may become permanently stuck inside this loop and it may be necessary to reboot the computer to regain control.

This program prints a multiplication table with 10 rows and 12 columns. The line number and a vertical line are printed along the left margin.

```
#include <stdio.h>
#define R 10
#define C 12

int main( void )
{
        int row, col;

        banner();
        printf( "\n\n Multiplication Table \n\n" );

        for (row = 1; row <= R; ++row) {       /* Print R rows. */
            printf( "%2i.  |", row );          /* Print left edge of row. */
            for (col = 1; col <= C; ++col) {   /* Print C columns in each row. */
                printf( "%4i", row * col );
            }
            printf( "\n     |\n" );            /* Print blank row. */
        }

        printf( "\n\n" );
        return 0;
}
```

Figure 6.16. Printing a table with nested `for` loops.

6.2.5 Nested Loops

A general rule of programming is that function follows form, that is, the form or shape of the input or output data frequently determines the way that code is written to process it. This is never more true than when processing tables. The rows and columns of a table are reflected in the code in the form of a loop to process the columns written within a loop that process the rows. We call such a control structure a *nested loop*[8]. This control structure is illustrated in a simple but general form by the program in Figure 6.16, which prints a 10-by-12 multiplication table. Its flow diagram is given in Figure 6.17.

Notes on Figure 6.16. Printing a table with nested `for` loops. This program prints an R-by-C multiplication table, where `R` and `C` are `#define`d as 10 and 12.

[8]For processing two-dimensional arrays, a `for` loop within a `for` loop is the dominant control pattern. This will be fully explored in Chapter 18.

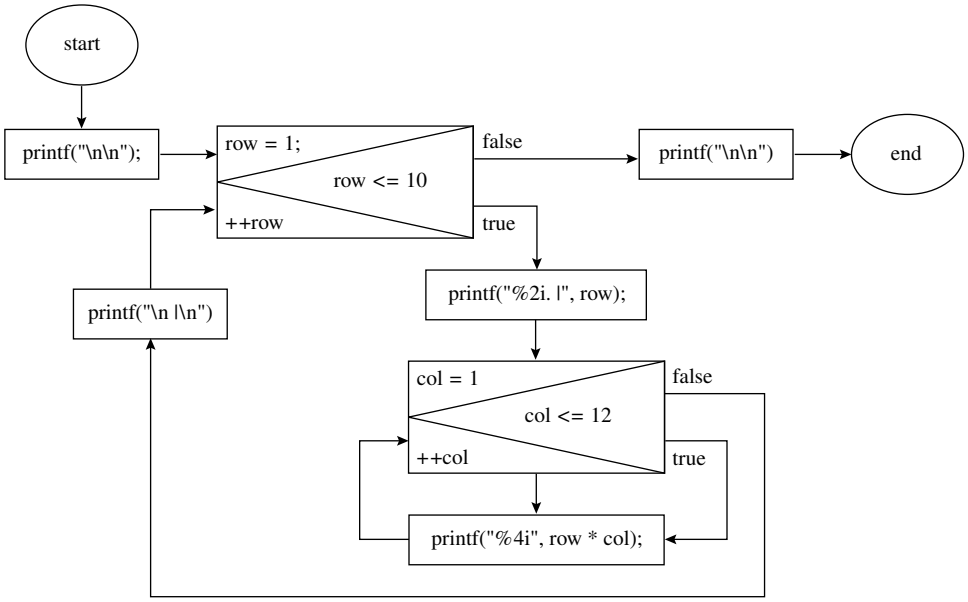


Figure 6.17. Flow diagram for the multiplication table program.

Outer box: The `row` ***loop.*** The outer loop is executed 10 times, once for each row of the table. Its body consists of the code to process one row; it prints a row label, processes all columns (the inner loop), and finishes the row by printing a newline and a vertical bar on the next line. This is a very typical processing pattern for a nested loop that does output. The output from one repetition, one row of numbers followed by a blank row looks like this:

```
1. |   1   2   3   4   5   6   7   8   9  10  11  12
   |
```

Inner box: the column loop. The output from each repetition of the inner loop is one column of one row of the table, consisting of two or three spaces and a number. The `printf()` in the inner loop will be executed 12 times per trip through the outer loop. After the 12th number has been printed, the inner loop exits and control goes to the `printf()` at the end of the outer box.

After 120 trips. In this program, control goes through the outer loop 10 times. For each trip through the outer loop, control goes through the inner loop 12 times. Therefore, control passes through the body of the inner loop (a `printf()` statement) a total of 120 times, processing every column in every row of the table. After finishing the 10th row, control goes to the final `printf()` and the `return` statement. The complete output is

```
Multiplication Table

 1. |   1   2   3   4   5   6   7   8   9  10  11  12
    |
 2. |   2   4   6   8  10  12  14  16  18  20  22  24
    |
 3. |   3   6   9  12  15  18  21  24  27  30  33  36
    |
 4. |   4   8  12  16  20  24  28  32  36  40  44  48
    |
 5. |   5  10  15  20  25  30  35  40  45  50  55  60
    |
 6. |   6  12  18  24  30  36  42  48  54  60  66  72
    |
 7. |   7  14  21  28  35  42  49  56  63  70  77  84
    |
 8. |   8  16  24  32  40  48  56  64  72  80  88  96
    |
 9. |   9  18  27  36  45  54  63  72  81  90  99 108
    |
10. |  10  20  30  40  50  60  70  80  90 100 110 120
    |
```

6.2.6 Delay Loops

A loop that executes many times but does nothing useful can be used to make the computer wait for a while before proceeding. Such a loop is called a **delay loop**. Delay loops often are used like timers to control the length of time between repeated events (see Figure 6.18). For example, a program that controls an automated factory process might use a delay loop to regulate sending analog signals to (or receiving them from) a device such as a motor generator.

Notes on Figures 6.19 and 6.18. Delaying progress, the `delay()` function. The `delay()` function implements a delay loop (see Figure 6.19). It calls the C library function `time()` to read the computer's real-time clock and return the current time, in units of seconds, represented as an integer so that we can do arithmetic with it. The type `time_t` is defined[9] by your local C system to be the right kind of integer[10] for storing the time on your system.

We add the desired number of seconds of delay to the current time to get the goal time, then store it in the variable `goal`. The loop calls `time()` continuously until the current time reaches the goal time. This loop is all test and no body. Technically, it is called a **busy wait** loop because it keeps the processor busy while waiting for time to pass. It is busy doing nothing, that is, wasting time[11]. On a typical personal computer, the `time()` function might end up being called 100,000 times or more during a delay of a few seconds. Busy waiting is an appropriate technique to use when a computer is dedicated to monitoring a single experiment or process. It is not a good technique to use on a shared computer that is serving other purposes simultaneously.

[9] Type definitions are discussed in Chapters 8, 12, 13, and 18.
[10] The various kinds of integers are discussed in Chapter 7.
[11] This is legal in C; a loop is not required to have any code in its body.

A delay loop is used here to regulate repetitions of a process.

```
#include <stdio.h>
void delay( int seconds );

int main( void )
{
    int j, max, seconds;

    printf( "This is an exercise program.\n\n"
            "How many pushups are you going to do? " );
    scanf( "%i", &max );
    if (max < 0) {
            printf( "Can't do %i pushups!\n", max );
            exit( 1 );
    }
    printf( "How many seconds between pushups? " );
    scanf( "%i", &seconds );
    if (seconds < 3) {
            printf( "Can't do a pushup in %i seconds!\n", seconds );
            exit( 1 );
    }
    printf( "OK, we will do %i pushups, one every %i seconds.\n",
            "Do one pushup each time you hear the beep.\n", max, seconds );

    for (j = 1; j <= max; ++j) {
        printf( "%i \a\n", j );   /* Do one. */
        delay( seconds );         /* Wait specified # of seconds. */
    }

    puts( "Good job.  Come again." );
}
```

Figure 6.18. Using a delay loop.

A delay loop usually is used inside another loop, which must perform a process repeatedly at a particular rate that is compatible with human response or a process being monitored. For example, the boxed loop in Figure 6.18 is used to time repetitions of an exercise. It outputs `\a` (a beep), then calls `delay()`, which waits the number of seconds specified by the user before returning. The output looks like this:

```
This is an exercise program.

How many pushups are you going to do? 5
How many seconds between pushups? 3
```

You may wish to put this function in your personal `mytools` library.

```
#include <time.h>
void delay( int seconds )
{
    time_t goal = time( NULL ) + seconds;   /* Add seconds to current time. */
    do {    /* Nothing */    } while (time( NULL ) < goal);
}
```

Figure 6.19. Delaying progress, the `delay()` function.

```
OK, we will do 5 pushups, one every 3 seconds.
Do one pushup each time you hear the beep.
1
2
3
4
5
Good job.  Come again.
```

6.2.7 Flexible-Exit Loops

Some languages support a kind of loop that permits the programmer to place the loop test anywhere between the beginning and the end of the loop body. Such a loop takes the place of the `while` and `do...while` loops in C and also provides other options. At the same time, it remains a one-in, one-out control structure, and therefore, is consistent with "structured programming". This sort of flexibile-exit structured loop can be imitated in C by a combination of three control statements: a `for` statement with empty loop test[12], where the only exit is by way of an `if` statement and a `break` statement somewhere within the loop body[13]. The skeleton of a `for` loop with an `if...break` is shown below and diagrammed on the left in Figure 6.20.

```
for (initialization; ; update) {
    statements
    if (condition)  break;
    more statements
}
next statement    /* Control comes here after the break. */
```

This degenerate statement sometimes is called an **infinite `for` loop** because the loop header has no test that can end the execution of the loop. A real infinite loop is not particularly useful because it never ends. However, an infinite `for` loop normally contains an `if...break` statement and, therefore, is not infinite because the `if...break` provides a way to leave the loop. Applications of the infinite `for` loop are shown in Figure 6.21, where it is used for data validation, and in Figure 12.31, where an infinite loop is combined with a `switch` to implement a complex control structure.

[12]It is legal to omit one, two, or all of the expressions in the loop header. However, the two semicolons must be present.

[13]Some practitioners use `while (1)` or `while (true)` instead of `for (;;)`. However, B. Kernighan, one of the inventors of C, wrote to me that he prefers the `for(;;)`, possibly because it avoids writing a meaningless test condition.

The only exit from this loop is through the `break` statement. The `for` header is used to initialize and update a counter but it has no exit test.

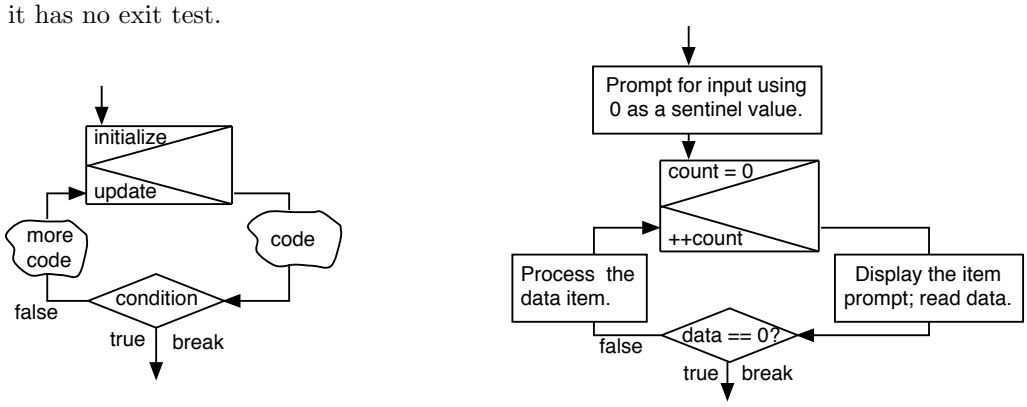


Figure 6.20. A structured loop with break.

Compare this input validation loop to the version in Figure 6.15 that is built on the `while` statement. This form is simpler and avoids duplication. The output is identical to the output from Figure 6.15.

```
printf( " Duration of trip in hours and minutes: " );
for (;;) {
    scanf( "%lg%lg", &hours, &minutes );
    hours = hours + ( minutes / 60 );
    if (hours >= 0) break;              /* Leave loop if input is valid. */
    printf( " Please re-enter; time must be >= 0: " );
}
```

Figure 6.21. Input validation loop using a `for` statement.

A loop that exits by means of an `if...break` statement has one big advantage over an ordinary loop: the loop body can have some statements before the loop test and more statements after it. This flexibility makes it easier to write code in many situations: it provides a straightforward, clear, nonredundant way to do jobs that are awkward when done with other loops. Its primary application is in loops that depend on some property of data that is being read and processed. The first part of the loop does the input or calculation, followed immediately by the test that stops the looping when data with a specified property is found.

Notes on Figure 6.21. Input validation using a `for` statement. Compare the code in Figure 6.21 to the validation loop in Figure 6.15.

Two designs are given for a counted sentinel loop. The version on the left is simpler. The version on the right avoids use of `break`.

```
for (k = 0; k < LIMIT; ++k)  {
    Prompt for and read an input.
    if (input == SENTINEL) break;
    Process the input data.
}
```

```
int done = 0;  /* false */

for (k = 0; k < LIMIT && !done; ++k) {
    Prompt for and read an input.
    if (input == SENTINEL) done = 1; /* true */
    else {
        Process the input data.
    }
}
```

Figure 6.22. Skeleton of a counted sentinel loop.

1. The original input prompt is written before the loop because it will not be repeated. (The error prompt is different.)
2. This loop can be written using a `while` statement as in Figure 6.15. However, since the `while` test is at the top of the loop, this implementation requires the input and calculation statements to be written twice, once before the loop and once in the loop body.
3. In contrast, if we use a flexible-exit loop, there is no need to write the `scanf()` and computation twice. The resulting code is simpler and clearer.
4. The input and calculation statements are done before the loop test, which is in an `if...break` statement. If the input is valid, control leaves the loop.
5. If not, we print an error prompt, return to the top of the loop, and read new data.
6. Since only a legal input will get the user out of this loop, an informative prompt is very important. If the user is not sure what data values are legal, the program may become permanently stuck inside this loop, and it may be necessary to reboot the computer to regain control.

6.2.8 Counted Sentinel Loops

Earlier in this section, we discussed a sentinel loop based on `while`. Often, we combine the sentinel test with a loop counter to make a counted sentinel loop. The general design for such a loop is given on the left in Figure 6.22. In this design, `LIMIT` is the maximum number of times the loop should be repeated, and `SENTINEL` is the designated sentinel value. To illustrate this pattern, we add a counter to the cash register program from Figure 6.9. The improved program is shown in Figure 6.23.

Moving the exit test to the top. A counted sentinel loop can be written two ways, with and without the use of `break`. Some professionals believe that the `break` statement should never be used because it is possible to overuse `break` and use it as a substitute for clean logic design. It is possible to implement the flexible-exit loop without the `break` statement by adding only a few lines of code. The loop design on the right of Figure 6.22 shows the kind of additions that are necessary to avoid the `break`.

We have added a status variable, `done`, that is set to false initially and then to true when the designated sentinel value is read. The "more code" section of the loop body is enclosed in an `else` clause so that the sentinel value will not be processed. Then the status variable is tested again and the loop ends. This

We use a `for` loop to count the number of data items that are entered, and a `break` statement to leave the loop when the input is a designated sentinel value.

```
#include <stdio.h>
#define SENTINEL 0   /* Signal for end of input. */
int main( void )
{
    double input; /* Price of one item. */
    double sum;   /* Total price of all items. */
    int count;    /* Number of items. */
    puts( " Cash Register Program.\n"
          " Enter prices; 0 to quit." );
    for (count=sum=0;  ;++count) {
        printf( "--> " );
        scanf( "%lg", &input );
        if (input == SENTINEL) break;
        sum += input;
        printf( "\t Input:  %g\n", input );
    } printf( " Your %i items cost $%g\n", count, sum );
    return 0;
}
```

The output is :

```
 Cash Register Program.
 Enter prices; 0 to quit.
--> 3.17
        Input:  3.17
--> 2.35
        Input:  2.35
--> 0.78
        Input:  0.78
--> 10.52
        Input:  10.52
--> 0
 Your 4 items cost $16.82
```

Figure 6.23. Breaking out of a loop.

is slightly less efficient and slightly longer to write than the version that uses break. Note, also, that the counter will be incremented one extra time and, therefore, we must subtract one from its value to get the true number of items that have been processed. The program in Figure 6.24 uses this logic.

6.2.9 Search Loops

A **search loop** examines a set of possibilities, looking for one that matches a given "key" value. The data items being searched can be stored in memory or calculated. The key value could be the entire item or part of it and it could be of any data type. The requirements for searching are almost identical in all cases:

- The program must know what key value to look for.
- There must be some orderly way to examine the possibilities, one at a time, until all have been checked.
- The loop must compare the key value to the current possibility. If they match, control must leave the loop.
- The search loop must know how many possibilities are to be searched or have some way to know when no possibilities remain. The loop must end when this occurs.

This program uses a status flag (done) instead of a break instruction to leave the loop. Compare it the simpler logic of Figure 6.23, that relies on the break.

```
#include <stdio.h>
#define SENTINEL 0   /* Signal for end of input. */
int main( void )
{
    double input;   /* Price of one item. */
    double sum;     /* Total price of all items. */
    int count;      /* Number of items. */
    int done = 0;   /* Set to 1 when sentinel is entered. */

    puts( " Cash Register Program.\n Enter prices; 0 to quit." );
    for (count=sum=0; !done ;++count) {
        printf( "--> " );
        scanf( "%lg", &input );
        if (input == SENTINEL) done = 1;
        else {
            sum += input;
            printf( "\t Input:  %g\n", input );
        }
    } printf( " Your %i items cost $%g\n", count-1, sum );
    return 0;
}
```

Figure 6.24. Avoiding a break-out.

Therefore, the search loop will terminate for two possible reasons: The key item has been found or the possibilities have been used up. The most straightforward way to implement this control pattern is to use a counted sentinel loop, with either the status flag or the break statement. The general pattern of a search loop is shown in Figure 6.25.

A search loop based on data input is shown in Figure 6.31. Search loops based on computed possibilities are illustrated in Figures **??** and in the Newton's method program on the text website. Search loops become increasingly important when there are large amounts of stored data. Loops that search arrays are illustrated in Chapter 18.

6.3 The switch Statement

C provides three ways to make a choice: the if...else statement, the switch statement, and the conditional operator. The if...else was fully covered in Chapter 3, this section is devoted to the switch, and the

Two designs are given for a search loop, with and without the use of `break`.

```
Read or select the key value.
for (k = 0; k < LIMIT; ++k) {
    Calculate or select next item to  test.
    if (current_data == key_value) break;
}
```

```
int done = 0;  /* false */

Read or select the key value.
for (k = 0; k < LIMIT && !done;  ++k) {
    Calculate or select next item to  test.
    if (current_data == key_value) done = 1;
}
```

Figure 6.25. Skeleton of a search loop.

conditional operator is explained in Appendix D.

6.3.1 Syntax and Semantics

A `switch` implements the same control pattern as a series of `if...else` statements. (Refer to Figure 3.11.) In both, we wish to select one action from a set of possible actions. A `switch` could be thought of as a shorter, integrated notation for the same logic as a nested `if...else` statement. Normally, one would use `if...else` when there are only two alternatives; a `switch` is only helpful when there are more than two choices.

The syntax of a `switch` statement. To illustrate the syntax and use of the `switch` statement, we use a program whose specification is given in Figure 6.26 and program code in Figure 6.28. A `switch` statement has several parts, in this order:

1. The keyword `switch`
2. An expression in parentheses that computes a value of some integral type[14]
3. Braces enclosing a series of labeled `cases`. The case labels must be constants or constant expressions[15] of the same type as (2).
4. Each case contains a series of statements. The last statement in the series is usually, but not always, a `break` statement.
5. One of the cases may be labeled `default`. If a `default` case is present, it is traditional (but not necessary) to place it last.
6. If several cases require the same processing, several case labels may be placed before the same statement. The last label may be `default`.

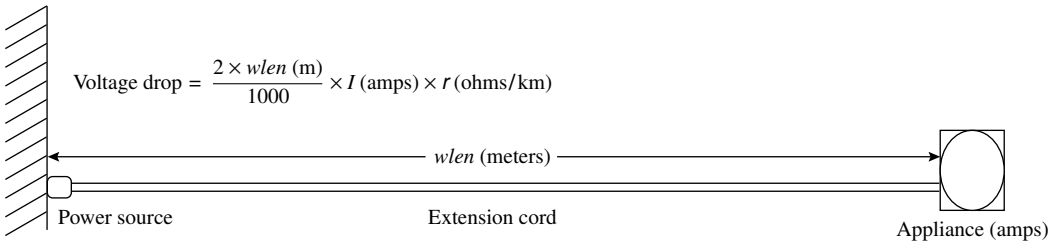
Execution of a `switch` statement. When a `switch` is executed, the expression in (2) is evaluated and its value compared to the case labels. If any case label matches, control will go to the statement following that label. Control then proceeds to the following statement and through all the statements after that until it reaches the bottom of the `switch`. This is *not* normally what a programmer wishes to do. It is far more

[14] C has several integral types in addition to `int`; they will be introduced in the next few chapters. These include `char`, `short`, `long`, `unsigned short`, `unsigned int`, and `unsigned long`.

[15] A **constant expression** is composed of operators and constant operands.

Problem scope: A contractor wants to automate the process of selecting an appropriate gauge wire for extension cords and temporary wiring at construction sites. Various long wires are used to supply electricity to power tools and other appliances at the site. All wires are standard annealed copper; calculations are to be made for a standard temperature of 20°C. There is a voltage drop in an extension cord due to the resistivity of its wire; heavier cords (lower-numbered gauges) have lower resistivity and incur less drop than lighter cords (higher-numbered gauges). The voltage drop is proportional to the length of the wire, so the drop can be significant in a long wire. This is an issue because appliances designed to operate at one voltage may overheat if operated at a voltage that is significantly lower. This program should evaluate a proposed wiring scheme and answer whether the wire will be adequate for its intended purpose.

Formula: An extension cord n meters long contains $2n$ meters of wire. The voltage drop is



where *wlen* is the length of the extension cord (in meters), ρ (rho) is the resistivity of the wire, and I is the current flowing in the wire, in amperes.

Constants: A voltage drop of up to 5 volts is acceptable. The resistivity, ρ, of copper wire at 20°C for the gauges used by this contractor are

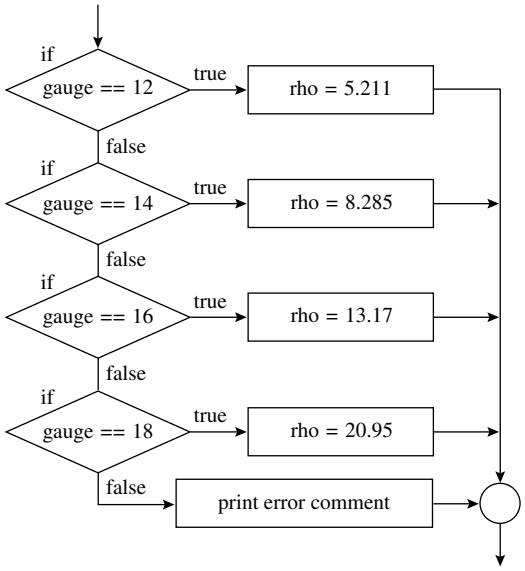
Gauge	ρ
12	5.211
14	8.285
16	13.17
18	20.95

Inputs: The contractor will type in the length of the extension cord he needs, the wire gauge, and the current rating (amps) of the appliance he will be using. The program should reject any gauge that is not in the table above.

Output required: Program headings and a termination message should be printed. Input values should be echoed. The voltage drop and the answer (gauge is adequate or not adequate) should be given. Three digits of precision are adequate.

Figure 6.26. Problem specification: Wire gauge adequacy evaluation.

We diagram a series of `if...else` statements that implements the specification in Figure 6.26.



We diagram the same logic again using a `switch` statement instead of an `if...else` statement.

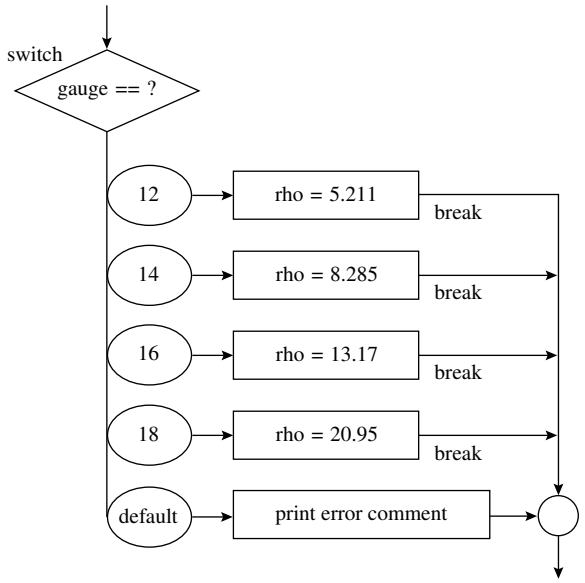


Figure 6.27. Diagrams for a nested conditional and a corresponding `switch`.

common to want the cases to be mutually exclusive. This is why each group of statements normally ends with a `break` statement that causes control to skip around all the following cases and go directly to the end of the `switch` statement. Programmers sometimes absentmindedly forget a `break` statement. In this case, the logic flows into the next case below it instead of to the end of the `switch` statement. Remember, this is not an error in the eyes of the compiler and it will not cause an error comment.

If no case label matches the value of the expression, control goes to the statement following the `default` label. If there is no `default` label, control just leaves the `switch` statement. This does not cause an error at either compile time or run time.

Diagramming a `switch` statement. The diagram of a nested `if...else` statement has a series of diamond-shaped `if` boxes, each enclosing a test, as shown on the left in Figure 6.27. The `true` arrow from each box leads to an action and the `false` arrow leads to the next `if` test. These tests will be made in sequence until some result is `true`.

In contrast, the diagram of a `switch` statement has a single diamond-shaped test box with a branching "out" arrow. This box encloses an expression whose value will be compared to the case labels. One branch is selected and the corresponding actions in the body of the `switch` statement are executed. Each set of

actions must end in a `break` statement. Control normally enters a box from the case label and leaves by an arrow that goes directly to the connector at the end of the `switch` statement. The diagram on the right of Figure 6.27 shows the `switch` statement that is used in Figure 6.28. Compare this to the `if...else` to its left. They implement the same logic; however, the version that uses `switch` is simpler.

6.3.2 A `switch` Application

Figure 6.28 shows a program that implements this `switch` statement to solve the problem specified in Figure 6.26. It illustrates "messy" integer case labels; that is, they are not consecutive numbers starting at 0 or 1[16].

Notes on Figure 6.28: Using a `switch`.

First box: input for the switch. We display a list of available gauges and prompt the user for a choice. The user sees this set of choices:

```
Wire Gauge Adequacy Evaluation

Please choose gauge of wire:
    12 gauge
    14 gauge
    16 gauge
    18 gauge
Enter selected gauge:
```

Second box: the `switch` statement.

- A `switch` that processes an integer input must have integer constants for case labels. This `switch` has four cases to process the four gauges plus a `default` case for errors.
- Each "correct" case contains one assignment statement that stores the resistivity value for the selected gauge wire. Each assignment is followed by a `break` that ends the case. A `default` case does not need a `break` because it is always last.
- If more extensive processing is needed, a program might call a different function to process each case.
- Error handling is done smoothly in this program. The `default` case intercepts inputs that are not supported and calls `fatal()` to print an error comment and abort the program. By calling `fatal()`, we avoid printing meaningless answers. An example would be

```
Enter selected gauge: 11
Gauge 11 is not supported.

WireGuage has exited with status 1.
```

[16] More applications of the `switch` statement are shown in Figures 13.5, 12.29, and 15.21.

The specifications for this program are in Figure 6.26.

```
#include <stdio.h>
#include <stdlib.h>
#define  MAXDROP   5.0      /* volts */

int main( void )
{
    int gauge;      /* selected gauge of wire */
    double rho;     /* resistivity of selected gauge of wire */
    double amps;    /* current rating of appliance */
    double wlen;    /* length of wire needed */
    double drop;    /* voltage drop for selected parameters */

    printf( "\n Wire Gauge Adequacy Evaluation" );

    printf( "\n Please choose gauge of wire:\n"
            "\t 12 gauge \n\t 14 gauge \n"
            "\t 16 gauge \n\t 18 gauge \n"
            " Enter selected gauge: " );
    scanf( "%i", &gauge );

    switch (gauge) {
      case 12:  rho = 5.211;   break;
      case 14:  rho = 8.285;   break;
      case 16:  rho = 13.17;   break;
      case 18:  rho = 20.95;   break;
      default:  printf( " Gauge %i is not supported.\n", gauge );
                exit(1);
    }

    printf( " Enter current rating for appliance, in amps: " );
    scanf ( "%lg", &amps );
    printf( " Enter the length of the wire, in meters: " );
    scanf ( "%lg", &wlen );
    drop = 2 * wlen / 1000 * rho * amps ;

    printf( "\n For %i gauge wire %g m long and %g amp appliance,\n"
            " voltage drop in wire = %g volts.  (Limit is %g.)  \n"
            gauge, wlen, amps, drop, MAXDROP );

    if (drop < MAXDROP)
        printf( "\n Selected gauge is adequate.\n" );
    else
        printf( "\n Selected gauge is not adequate.\n" );

    return 0;
}
```

Figure 6.28. Using a `switch`.

Third box: calculating the voltage drop.

- Control goes to this box after every `break`. It does not go here after executing the `default` clause because that clause calls `fatal()`, which aborts execution.
- This box prompts for and reads the rest of the input data. This is done after the `switch` statement, not before, because the menu selection can be an invalid choice and there is no point reading the rest of the data until we know the gauge is one of those listed in the table.
- We calculate the voltage drop as soon as all the data have been read. The formula for voltage drop uses the resistivity value selected by the `switch` statement. We divide by 1,000 because ρ is given in ohms/kilometer and the wire length is given in meters. We multiply by 2 because each extension cord contains a pair of wires running its full length.

Fourth box: the answers. We ran the program and tested two cases. Each time the program was run, the greeting comment, menu, and termination comment were printed; for brevity, these are not repeated here. Dashed lines are used to separate the runs.

```
Enter selected gauge: 12
Enter the current rating for the appliance, in amps: 10
Enter the length of the wire, in meters: 30

For 12 gauge wire 30 meters long and 10 amp appliance,
voltage drop in wire = 3.1266 volts.  (Limit is 5 volts.)

Selected gauge is adequate.
-----------------------------------------------------
Enter selected gauge: 16
Enter the current rating for the appliance, in amps: 10
Enter the length of the wire, in meters: 30

For 16 gauge wire 30 meters long and 10 amp appliance,
voltage drop in wire = 7.902 volts.  (Limit is 5 volts.)

Selected gauge is not adequate.
```

6.4 Search Loop Application: Guess My Number

The program specified in Figure 6.29 and shown in Figure 6.30 is a simple interactive game in which the player is given a limited number of turns to guess (and enter) the hidden number. The game ends sooner if the player's input equals the program's hidden number (the sentinel value). The implementation uses a counted sentinel loop with `break`. The `for` statement is used to count the player's guesses and the `if...break` is used in the usual way to implement a possible early exit after a correct guess. Figure 6.31 is a flow diagram for this program[17].

Notes on Figures 6.31 and 6.30: A sentinel loop using `for` and `if...break` statements.

First box, Figure 6.30: the concealed number. This program is a simplification of an old game that asks the user to guess a concealed number. The computer responds to each guess by telling the user whether the guess was too large, too small, or right on target. In a complete program, the concealed number would be randomly chosen and the number of guesses allowed would be barely enough (or not quite enough) to

[17] We will revisit and elaborate on this game in Chapter 7.

1. **Problem scope:** Write a program to play an interactive guessing game with the user. The user is given a fixed number of guesses to find a hidden number.
2. **Constants:** The hidden number will be between 1 and 30; the user will be given up to 5 guesses.
3. **Inputs:** The user will enter a series of guesses.
4. **Output required:** The program will respond each time by saying the guess is too low, correct, or too high. If the guess is correct, the program should display the message"you win". If the available guesses are used up, the program should display the message"You lose".
5. **Other:** An input value outside of the specified range will be counted as a wrong guess. If the user makes optimal guesses, he can always win.

Figure 6.29. Problem specification: Guess my number.

win the game every time. We give a full version of this program in Chapter 7. In this simplification, we arbitrarily choose 17 as the concealed number. The five guesses allowed are enough to uncover this number if the user makes no mistakes.

Outer box: operation of the loop.

- This `for` loop prompts for, reads, checks, and counts the guesses. It will allow the user up to five tries to enter the correct number. On each trial, the program gives feedback to guide the user in making the next guess. After the fifth unsuccessful try, the loop test will terminate the loop.
- We initialize the loop counter to 1 (rather than the usual 0) because we want to print the counter value as part of the prompt. The computer does not care about the counter values, but users prefer counters that start at 1, not 0.
- We use `<=` to compare the loop counter to the loop limit because the loop variable was initialized to 1, not 0, and we want to execute the loop when the counter equals the number of allotted trials.
- Each guess has three possibilities: It can be correct, too low, or too high. To check for three possibilities, we need two `if` statements. The first `if` statement is in the inner box. If the input matches the concealed number, the break is executed. Control will leave the `for` loop and go to the first statement after the loop. The second `if` statement prints an appropriate comment depending on whether the guess is too high (the true clause) or too low (the false clause). Control then goes back to the top of the loop.

Inner box: There are two ways to leave the loop: either the guess was correct or the guesser was unable to find the hidden number in the number of tries allowed. Here, we test the number of guesses that were used to distinguish between these two cases, then print a success or failure comment. This is a typical way to handle a loop with two exit routes.

The diagram: Figure 6.31.

- A `break` statement is represented by an arrow and a connector, not by a rectangle, diamond, or ellipse. Note the word `break` on the `true` arrow of the `if` condition diamond. The circular connector on the loop's exit arrow is for the `break`.

This program illustrates the simplest form of search loop: we search the input data for a value that matches the search key. The implementation uses a counted sentinel loop with `break`.

```
#include <stdio.h>
#define TRIES 5

int main( void )
{
    int k = 0;              /* Loop counter. */
    int guess;              /* User's input. */
    int num = 17;

    printf( " Can you guess my number?  It is between 1 and 30.\n"
            " Enter a guess at each prompt; You have %i tries.\n", TRIES );

    for (k = 1; k <= TRIES; ++k) {
        printf( "\n Try %i: ", k );
        scanf( "%i", &guess );
        if (guess == num) break;
        if (guess > num) printf( " No, that is too high.\n" );
        else printf( " No, that is too low.\n" );
    }

    if (guess == num) printf( " YES!!  That is just right.  You win!  \n" );
    else printf( " Too bad --- You lose again!\n" );
    return 0;
}
```

Figure 6.30. An input-driven search loop.

- The `if...break` statement is the first diamond in the `for` loop. If the guess is not correct, control stays in the loop and enters the `if...else` statement at the end of the loop body. If the guess is correct, control flows out of the loop along the `break` arrow and enters the code that follows the loop.

6.5 What You Should Remember

6.5.1 Major Concepts

- The `while` loop tests the loop exit condition before executing the loop body. The body, therefore, is executed zero or more times.

This is a flow diagram of the program in Figure 6.30.

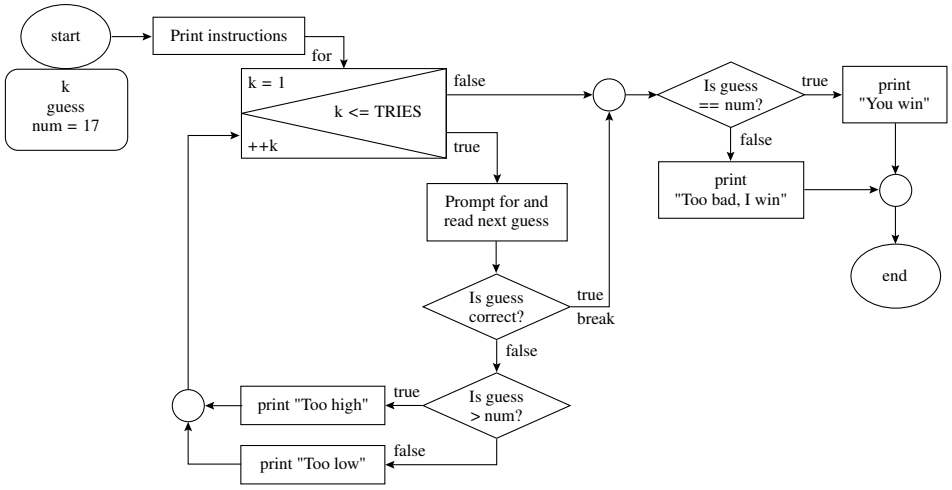


Figure 6.31. A counted sentinel loop.

- The `while` loop is used for sentinel loops, delay loops, processing data sets of unknown size, and data validation loops when it is important to give the user an error comment different from an ordinary prompt.
- The `for` loop implements the same control pattern as the `while` loop but has a different and more compact syntax.
- The `for` loop is used for counted loops and processing any set of data whose exact size or maximum size is known ahead of time.
- The `do...while` loop executes the loop body before performing the loop exit test. Therefore, the body is always executed one or more times.
- The `do...while` statement is used to form query loops that call a `work()` function repeatedly. It also can be used for data validation.
- A nested loop is used to process the rows and columns of a table.
- A `continue` statement within any kind of loop transfers control back to the top of the loop.
- An `if...break` statement can be used to leave any kind of loop but normally is used to leave a `for` loop before the iteration limit is reached.
- Any one or all the expressions in the loop header of a `for` statement can be omitted. If the loop test is omitted, an `if...break` is used to end the loop. This combination is used for data validation loops.

- The `switch` statement has several clauses, called *cases*, each labeled by a constant. At run time, a single value is compared to each constant and the clause corresponding to the matching constant is executed. A `default` case will be executed if none of the case constants match.
- Most cases in `switch` statements (except the last one) end with a `break` statement. If a case has no `break`, the statements for that case `and` the next case will be executed.

6.5.2 Programming Style

Many programs can be improved by eliminating useless work and simplifying nested logic. The resulting code always is simpler and easier to debug and usually is substantially shorter. Some specific suggestions for improving program style follow:

- The golden rule of style is this: Indent your code consistently and properly.
- Line up the first letter of `for`, `while`, or `switch` with the curly bracket that closes the following block of statements. Indent all the statements within the block.
- Use the `switch` statement instead of a series of nested `if...else` statements when the condition being tested is simple enough.
- Do not compute the same expression twice—compute it once and save the answer in a variable. Any time you do an action twice and expect to get the same answer, you create an opportunity for disaster if the program is modified.
- If two statements are logically related, put them near each other in the program. Examples of this principle are

  1. Initialize a loop variable just before the beginning of the loop.
  2. Do the input just before the conditional that tests it.

- Use the `for` loop effectively, putting all initializations and increment steps in the part of the loop.
- When one control structure is placed within the body of another, we call it *nested logic*. For example, a `switch` statement can be nested inside a loop, and a loop can be nested inside one clause of an `if` statement. Most real applications require the use of nested logic. Some generally accepted guidelines are these:

  1. Keep it simple. An `if` statement nested inside another `if` statement inside a loop has excessive complexity. Many times, the `if` statement can be moved outside the loop or the second `if` statement can be eliminated by using a logical operator.
  2. Establish a regular indenting style and stick to it without exception.
  3. Limit the number of levels of nesting to two or three.
  4. If your application seems to require deeper nesting, break up the nesting by defining a separate function that contains the innermost one or two levels of logic.

- To improve efficiency, move everything possible outside of a loop. For example, when you must compute the average of some numbers, use the loop to count the numbers and add them to a total. Do not do the division within the loop. Moving actions out of the loop frequently shortens the code; it always simplifies it and makes it more efficient. In poorly written programs, "fat loops" account for many of the extra lines. Take advantage of any special properties of the data, such as data that may be sorted, may have a sentinel value on the end, or may be validated by a prior program step and not need validation again when processed.
- During the primary debugging phase of an application, every loop should contain some screen output. This allows the programmer to monitor the program's progress and detect an infinite loop or a loop that is executed too many or too few times. Debugging output lines should print the loop variable, any input read within the loop, and any totals or counters changed in the loop.

6.5.3 Sticky Points and Common Errors

Semicolons in the wrong places. The header of a `for` loop or the condition clause of a `while` loop or an `if` statement is, normally, *not* followed by a semicolon.

Off-by-one errors. The programmer must take care with every loop to make sure it repeats the correct number of times. A small error can cause one too many or one too few iterations. Every loop should be tested carefully to ensure that it executes the proper number of times. In some counted loops, the loop counter is used as a subscript or as a part of a computation. In these loops, the results may be wrong even when the loop is repeated the correct number of times. This kind of error happens when the value of the loop counter is **off by one** because it is not incremented at the correct time in relation to the expression or expressions that use the counter's value. Consider these two counted loops:

```
for (sumj = j = 0; j < 10; ++j)
    sumj += j;                  /* Sum from 0 to 9 */

// ------------------------------------------------
sumk = k = 0;
while (k < 10) {                /* Sum from 1 to 10 */
    ++k;
    sumk += k;
}
```

These loops are very similar, but `k` is incremented before adding it to `sumk` and `j` is incremented after adding it to the sum. A programmer must decide which timing pattern is correct and be careful to write the correct form.

Infinite loops. The `for` loop with no loop test sometimes is called an *infinite loop*, although most such loops contain an `if...break` statement that stops execution under appropriate conditions. Such loops are useful tools. However, a real infinite loop is not useful and should be avoided. Such loops are the result of forgetting to include an update step in the loop body. (In a counted loop, the update step increments

the loop variable. In an input-controlled loop, it reads a new value for the loop variable.) During the construction and debugging phases of a program, it is a good idea to put a `puts()` statement in every loop so that you can easily identify an infinite loop when it occurs.

Nested logic. When using nested logic, the programmer must know how control will flow into, through, and out of the unit. Statements such as `break` and `continue` affect the flow of control in ways that are simple when single units are considered but become complex when control statements are nested. Be sure you understand how your control statements interact.

6.5.4 Where to Find More Information

- Strings will be introduced in Chapter 12; a sentinel loop that processes a string is shown in Figure 12.12.
- Chapter 19 presents the code for quicksort, one of the best sorting algorithms. Arrays are used with sentinel loops are used in that program.
- Chapter 21 presents two kinds of linked lists; both are typically processed using a sentinel loop.
- Chapter 8, Figure ex-char-work, we show how to use a query loop with character inputs (y/n), which makes a better human interface than numeric responses (1/0).
- Figure 6.31 implements a simple version of a familiar guessing game. We revisit and elaborate on this game in the Random Numbers program on the website.
- The `switch` statement is used with an enumerated type in Figure 13.5. Figures 12.29 and 15.21 show a very typical use of `switch` to process single-character selections from a menu-interface.
- Appendix D describes the properties and usage of the conditional operator, `? :`, which is the only operator in `C` that has three operands. The first operand (before the `?` is a condition. The second operand gives an expression to evaluate if the condition is true. If it is false, the expression after the `:` is evaluated. The result of the conditional operator is the result of whichever expression was evaluated.

6.5.5 New and Revisited Vocabulary

These terms and concepts have been defined or expanded in the chapter:

for loop loop header	trip count	search loop
loop variable	input validation loop	input-controlled loop
sentinel loop	nested loops	premature exit
sentinel value	delay loop	avoiding `break`
repeat query	busy wait	constant expression
counted loop	flexible-exit loop	off-by-one error
	infinite `for` loop	

The following `C` keywords were discussed in this chapter:

`for` loop	`break` statement	`switch` statement
`,` (comma operator)	`if...break` statement	`case` statement
`do...while` loop	`continue` statement	`default` clause

6.6 Exercises

6.6.1 Self-Test Exercises

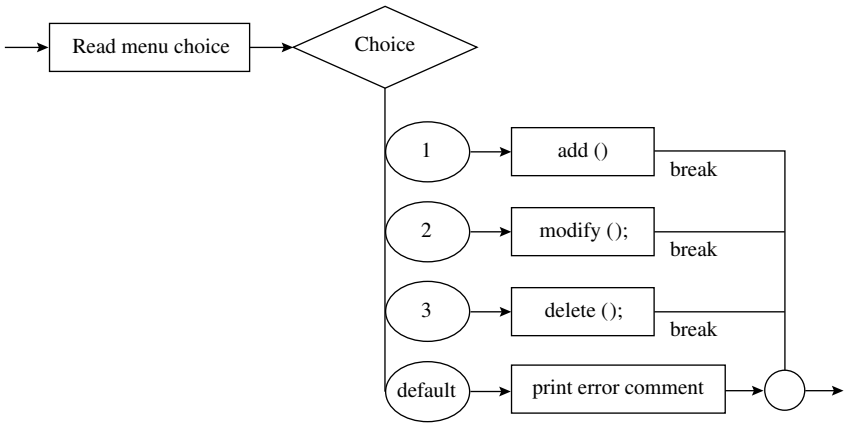
1. Explain the fundamental differences between a series of `if...else` statements and a `switch` statement. Under what conditions would you use a `switch` statement? Give an example of a problem for which you could not use a `switch` statement.
2. Explain the fundamental differences between a `while` loop and a `do...while` loop. In what situation would you use each?
3. The following program contains a loop. What is its output? Rewrite the loop using `for` instead of `while`. Make sure the output does not change.

```
#include <stdio.h>
int main ( void )
{   int k, sum;

    sum = k = 0;
    while (k < 10) {
        sum += k;
        ++k;
    }
    printf( "A. %i %i \n", k, sum );
}
```

4. The following program contains a loop. What is its output? Rewrite the loop using `while` instead of `do...while`. Make sure the output does not change.

```
#include <stdio.h>
int main ( void )
{   int k, sum;

    printf( " Please enter an exponent >= 0: " );
    scanf( "%i", &k );
    sum = 1;
    do {
        if (k > 0) sum = 2*sum;
        --k;
    } while (k > 0);
    printf( "B. %i \n", sum );
}
```

5. Given the following fragment of a flow diagram, write the code that corresponds to it and define any necessary variables:



6. Draw a flow diagram for the cash register program in Figure 6.9.
7. Rewrite the following `switch` statement as an `if...else` sequence. (Write code, not a flowchart.)

```
switch (k) {
    case 2:
    case 12:  puts( "You lose" ); break;
    case 7:
    case 11:  puts( "You win" ); break;
    default:  puts( "Try again" );
    }
```

8. Analyze the following loop and draw a flow diagram for it. Then trace the execution of the loop using the initial values shown. Use a storage diagram to show the succession of values stored in each variable. Finally, show the output, clearly labeled and in one place.

```
for (k = 0,j = 1; j < 3; j++)  {
    k += j;
    printf( "\t %i\t %i\n", j, k );
}
printf( "\t %i\t %i\n", j,k );
```

9. What does each of the following loops print? They are supposed to print the numbers from 1 to 3. What is wrong with them?

(a)
```
for(k = 0; k < 3; ++k) printf( "k = %i", k );
```

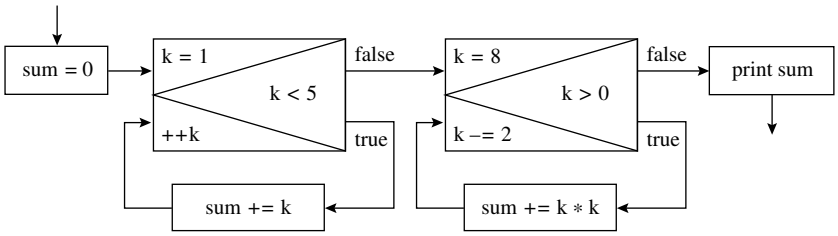
(b)
```
k = 1;
do {
    printf( "k = %i", k );
    k++;
} while (k < 3);
```

6.6.2 Using Pencil and Paper

1. Explain the fundamental differences between a counted loop and a sentinel loop. What C statement would you use to implement each?
2. Explain the fundamental similarity between a `while` loop and a `for` loop. In what situation would you use each?
3. Rewrite the following `if...else` sequence as a `switch` statement. Write code, not a flowchart.

```
if (i == 0) puts( "bad" );
else if (i >= 1 && i < 3) puts( "better" );
else if (i == 4 || i == 5) puts( "good" );
else puts( "sorry" );
puts("------"):
```

4. Draw a flow diagram for the cash register program in Figure 6.23.
5. Given the following fragment of a flow diagram, write the code that corresponds to it and define any necessary variables. What number will be printed by the last box?



6. The following program contains a loop. What is its output? Rewrite the loop using `do...while` instead of `for`. Keep the output the same.

```
#include <stdio.h>
int main ( void )
{   int k, sum;
    for ( sum = k = 0; k < 5; ++k ) sum += k;
    printf( "C. %i %i \n", k, sum );
}
```

7. The following program contains a loop. What is its output? Rewrite the loop without using `break`. Keep the output the same.

```
#include <stdio.h>
int main ( void )
{   int k, sum;
    for (sum = k = 1 ; k < 10; k++) {
```

```
            sum *= k;
            if (sum > 10*k) break;
        }
        printf( "D. %i %i \n", k, sum );
    }
```

8. What does the following loop print? It is supposed to print out the numbers from 1 to 3. What is wrong with it?

```
for(k=1; k<=3; ++k);
    printf( "k=%i", k );
```

9. Analyze each loop that follows and draw a flow diagram for it. Then trace execution of the loop using the initial values shown. Use a storage diagram to show the succession of values stored in each variable. Finally, show the output from each exercise, clearly labeled and in one place.

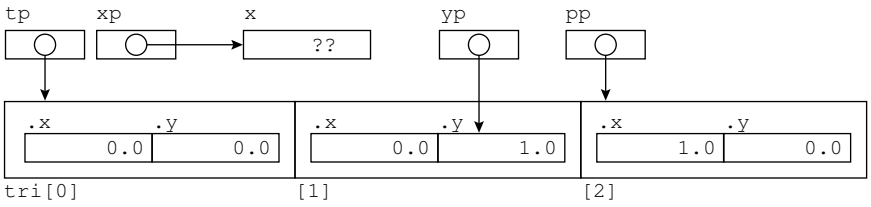
   (a)
```
k=2;
do { printf("\t %i\n", k); --k; } while (k>=0);
```

   (b)
```
j=k=0;
while (j<3){ k+=j; printf("\t %i\t %i\n", j, k); ++j; }
```

10. Given the following code,

```
#include <stdio.h>
int main( void )
{ int j,k;

   for (j = 1; j < 3; j++) {
       for (k = 0; k < 5; k += 2) {
           if (k != 4) printf( " %i, %i ", k, j );
           else k--;
       }
       putchar( '\n' );
       if (j % 2 == 1) printf( ">>> %i , %i <<<\n", j, k );
   }
}
```

   (a) Draw a flowchart that corresponds to the program.
   (b) Using a table like the one that follows, trace the execution of the program. Show the initial values of `j` and `k`. For each trip through a loop, show how the values of `j` and `k` change and what is displayed on the screen. Draw a vertical line between the columns that correspond to each trip through the inner loop.



6.6.3 Using the Computer

1. Sum a series.

   Write a program that uses a `for` loop to sum the first 100 terms of the series $1/x^2$. Use Figure 6.14 as a guide. Develop a test plan and carry it out. Turn in the source code and output of your program when run on the test data.

2. Sum a function.

   Write a function with parameter x that will compute the value of $f(x) = (3 \times x + 1)^{\frac{1}{2}}$.

   Write a main program that sums $f(x)$ from $x = 0$ to $x =$1,000 in increments of 2. Use a `for` loop. Print the result.

3. Find the best.

   Write a program that will allow an instructor to enter a series of exam scores. After the last score, the instructor should enter a negative number as a signal that there is no more input. Print the average of all the scores and the highest score entered.

4. Gas prices.

   The example at the end of Chapter 3 (Figure 3.18) is a program that converts a Canadian gas price to an equivalent U.S. gas price. This program does the calculation for only one price. Modify it so that it can convert a series of prices, as follows:

   (a) Remove the price-per-liter input, computation, and output from the main program and put them in a separate function, named `convert()`.
   (b) In place of this code in the main program, substitute a query loop that will allow you to enter a series of Canadian prices. For each, call `convert()` to do the work.

5. An increasing voltage pattern.

   Write a program that will calculate and print out a series of voltages.

   (a) Prompt the user for a value of $v$max and restrict it to the range `12 <= `$v$max` <= 24`. Let time `t` start at `0` and increase by 1 at each step until the voltage $v > 95\%$ of $v$max; `v` is a function of time according to the following formula. Print the time and voltage at each step.

$$v = v\text{max} \times \left(1 - \mathrm{e}^{(-0.1 \times t)}\right)$$

(b) Add a delay loop so that the voltage output is timed more slowly. See Figure 6.18.

6. Sine or cosine?

   (a) Write a double→double function, `f()`, with one `double` parameter, `x`, that will evaluate either $f_1(x) = x^2 \sin(x) + e^{-x}$ if $x \leq 1.0$ or $f_2(x) = x^3 \cos(x) - \log(x)$ if $x > 1.0$, and return the result. Write a prototype that can be included at the top of a program.
   (b) Start with the main program in Figure 6.11. Modify the program title and write a new `work()` function that will input a value for `x`, call `f()` using the value of `x`, and output the result.
   (c) Design a test plan for your program.
   (d) Incorporate all the pieces of your program in one file and compile it. Then carry out the test plan to verify the program's correctness.

7. A voltage signal.

   An experiment will be carried out repeatedly using an ac voltage signal. The signal is to have one of three values, depending on the time since the beginning of the experiment:

   (a) For time $t < 1$, volts$(t) = 0.5 \times \sin(2t)$.
   (b) For time $1.0 \leq t \leq 10.0$, volts$(t) = \sin(t)$.
   (c) For time $t > 10.0$, volts$(t) = \sin(10.0)$.

   (a) Write a function named `volts()` with t as a parameter that will calculate and return the correct voltage. Use the function in Figure 5.18 as a guide.
   (b) Using Figure 6.11 as a guide, write a main program that starts at time 0. Use a loop to call the `volts()` function and output the result. Increment the time by 0.5 after each repetition until the time reaches 12. Print the results in the form of a neat table.

8. Rolling dice.
   Over a large number of trials, a "fair" random number generator will return each of the possible values approximately an equal number of times. Therefore, in each set of 60 trials, if values are generated in the range $1 \ldots 6$, there should be about 10 ones and 10 sixes.

   (a) Using the loop in Figure 5.26 as a guide, write a function that will generate 60 random numbers in the range $1 \ldots 6$. Use `if` statements to count the number of times 1 and 6 each turns up. At the end, print out the number of ones and the number of sixes that occurred.
   (b) Following the example in Figure 6.11, write a main program that will call the function from part (a) 10 times. The main program should print table headings; the function will print each set of results under those headings. Look at your results; are the numbers of ones and sixes what you expected? Try it again. Are the results similar? Can you draw any conclusions about the quality of the algorithm for generating random numbers?

9. Prime number testing.
   A prime number is an integer that has no factors except itself and 1. The first several prime numbers are 2, 3, 5, 7, 11, 13, 17, and 19. Very large prime numbers have become important in the field of cryptography. The original public-key cryptographic algorithm is based on the fact that there is no fast way to find the prime factors of a 200-digit number that is the product of two 100-digit prime numbers. In this program, you will implement a simple but very slow way to test whether a number is prime.

   One method of testing a number N for primality, is by calculating `N % x`, where x is equal to every prime number from 2 to $R = \sqrt{N}$. If any of these results equals 0, then N is not a prime. We can stop testing at $\sqrt{N}$, since if N has any factor greater than R, it also must have a factor less than or equal to R. Unfortunately, keeping track of a list of prime numbers requires techniques that have not yet been presented. However, a less efficient method is to calculate `N % x` for $x = 2$ and every odd number between 3 and $\sqrt{N}$. Some of these numbers will be primes, most will not. But if any one value divides N evenly, we know that N is not a prime.

   Write a function that enters an integer N to test and prints the word `prime` if it is a prime number or `nonprime` otherwise. Write a main program with a query loop to test many numbers.