C Program Control

C How to Program, 6/e

OBJECTIVES

In this chapter, you'll learn:

- The essentials of counter-controlled repetition.
- To use the `for` and `do...while` repetition statements to execute statements repeatedly.
- To understand multiple selection using the `switch` selection statement.
- To use the `break` and `continue` statements to alter the flow of control.
- To use the logical operators to form complex conditional expressions in control statements.
- To avoid the consequences of confusing the equality and assignment operators.

4.1 Introduction

- In this chapter, repetition is considered in greater detail, and additional repetition control statements, namely the `for` and the `do…while`, are presented.
- The `switch` multiple-selection statement is introduced.
- We discuss the `break` statement for exiting immediately from certain control statements, and the `continue` statement for skipping the remainder of the body of a repetition statement and proceeding with the next iteration of the loop.
- The chapter discusses logical operators used for combining conditions, and summarizes the principles of structured programming as presented in Chapter 3 and 4.

4.2 Repetition Essentials

- A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.
- We have discussed two means of repetition:
  - Counter-controlled repetition
  - Sentinel-controlled repetition
- Counter-controlled repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed.
- Sentinel-controlled repetition is sometimes called indefinite repetition because it's not known in advance how many times the loop will be executed.

4.2 Repetition Essentials (Cont.)

- In counter-controlled repetition, a control variable is used to count the number of repetitions.
- The control variable is incremented (usually by 1) each time the group of instructions is performed.
- When the value of the control variable indicates that the correct number of repetitions has been performed, the loop terminates and the computer continues executing with the statement after the repetition statement.

4.2 Repetition Essentials (Cont.)

- Sentinel values are used to control repetition when:
  - The precise number of repetitions is not known in advance, and
  - The loop includes statements that obtain data each time the loop is performed.
- The sentinel value indicates “end of data.”
- The sentinel is entered after all regular data items have been supplied to the program.
- Sentinels must be distinct from regular data items.

4.3 Counter-Controlled Repetition

- Counter-controlled repetition requires:
  - The name of a control variable (or loop counter).
  - The initial value of the control variable.
  - The increment (or decrement) by which the control variable is modified each time through the loop.
  - The condition that tests for the final value of the control variable (i.e., whether looping should continue).

4.3 Counter-Controlled Repetition (Cont.)

- Consider the simple program shown in Fig. 4.1, which prints the numbers from 1 to 10.
- The definition

```
int counter = 1; /* initialization */
```

  names the control variable (`counter`), defines it to be an integer, reserves memory space for it, and sets it to an initial value of `1`.
- This definition is not an executable statement.

```c
/* Fig. 4.1: fig04_01.c
   Counter-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter = 1; /* initialization */

   while ( counter <= 10 ) { /* repetition condition */
      printf ( "%d\n", counter ); /* display counter */
      ++counter; /* increment */
   } /* end while */

   return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.1 | Counter-controlled repetition. (Part 1 of 2.)

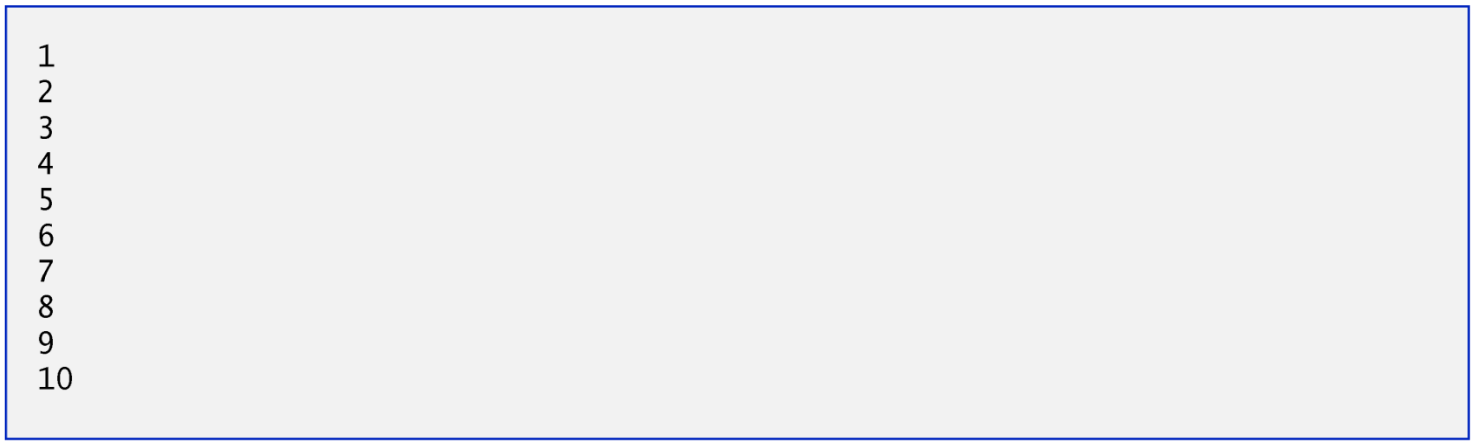

Fig. 4.1 | Counter-controlled repetition. (Part 2 of 2.)

4.3 Counter-Controlled Repetition (Cont.)

- The definition and initialization of `counter` could also have been written as

```
int counter;
counter = 1;
```

- The definition is not executable, but the assignment is.
- We use both methods of initializing variables.
- The statement

```
++counter; /* increment */
```

increments the loop counter by 1 each time the loop is performed.

4.3 Counter-Controlled Repetition (Cont.)

- The loop-continuation condition in the `while` statement tests if the value of the control variable is less than or equal to `10` (the last value for which the condition is true).
- The body of this `while` is performed even when the control variable is `10`.
- The loop terminates when the control variable exceeds `10` (i.e., `counter` becomes `11`).

4.3 Counter-Controlled Repetition (Cont.)

- You could make the program in Fig. 4.1 more concise by initializing `counter` to `0` and by replacing the `while` statement with

```
while ( ++counter <= 10 )
   printf( "%d\n", counter );
```

- This code saves a statement because the incrementing is done directly in the `while` condition before the condition is tested.
- Also, this code eliminates the need for the braces around the body of the `while` because the `while` now contains only one statement.
- Some programmers feel that this makes the code too cryptic and error prone.

Common Programming Error 4.1

Floating-point values may be approximate, so controlling counting loops with floating-point variables may result in imprecise counter values and inaccurate termination tests.

Error-Prevention Tip 4.1

Control counting loops with integer values.

Good Programming Practice 4.1

Too many levels of nesting can make a program difficult to understand. As a rule, try to avoid using more than three levels of nesting.

Good Programming Practice 4.2

The combination of vertical spacing before and after control statements and indentation of the bodies of control statements within the control-statement headers gives programs a two-dimensional appearance that greatly improves program readability.

4.4 `for` Repetition Statement

- The `for` repetition statement handles all the details of counter-controlled repetition.
- To illustrate its power, let's rewrite the program of Fig. 4.1.
- The result is shown in Fig. 4.2.

```c
/* Fig. 4.2: fig04_02.c
   Counter-controlled repetition with the for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter; /* define counter */

   /* initialization, repetition condition, and increment
      are all included in the for statement header. */
   for ( counter = 1; counter <= 10; counter++ ) {
      printf( "%d\n", counter );
   } /* end for */

   return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.2 | Counter-controlled repetition with the `for` statement.

4.4 `for` Repetition Statement (Cont.)

- When the `for` statement begins executing, the control variable `counter` is initialized to `1`.
- Then, the loop-continuation condition `counter <= 10` is checked.
- Because the initial value of `counter` is `1`, the condition is satisfied, so the `printf` statement (line 13) prints the value of `counter`, namely `1`.
- The control variable `counter` is then incremented by the expression `counter++`, and the loop begins again with the loop-continuation test.

4.4 `for` Repetition Statement (Cont.)

- Since the control variable is now equal to `2`, the final value is not exceeded, so the program performs the `printf` statement again.
- This process continues until the control variable `counter` is incremented to its final value of 11—this causes the loop-continuation test to fail, and repetition terminates.
- The program continues by performing the first statement after the `for` statement (in this case, the `return` statement at the end of the program).
- Figure 4.3 takes a closer look at the `for` statement of Fig. 4.2.

4.4 `for` Repetition Statement (Cont.)

- Notice that the `for` statement "does it all"—it specifies each of the items needed for counter-controlled repetition with a control variable.
- If there is more than one statement in the body of the `for`, braces are required to define the body of the loop.

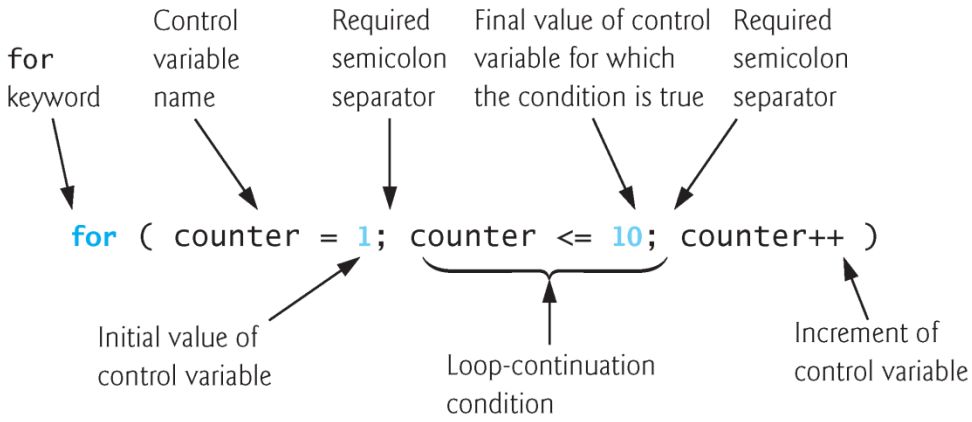

Fig. 4.3 | `for` statement header components.

4.4 `for` Repetition Statement (Cont.)

- Notice that Fig. 4.2 uses the loop-continuation condition `counter <= 10`.
- If you incorrectly wrote `counter < 10`, then the loop would be executed only 9 times.
- This is a common logic error called an off-by-one error.

Common Programming Error 4.2

Using an incorrect relational operator or using an incorrect initial or final value of a loop counter in the condition of a `while` *or* `for` *statement can cause off-by-one errors.*

Error-Prevention Tip 4.2

Using the final value in the condition of a `while` *or* `for` *statement and using the <= relational operator will help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be* `counter <= 10` *rather than* `counter < 11` *or* `counter < 10`.

4.4 for Repetition Statement (Cont.)

- The general format of the `for` statement is
  - `for ( expression1; expression2; expression3 )`
    statement

  where *expression1* initializes the loop-control variable, *expression2* is the loop-continuation condition, and *expression3* increments the control variable.
- In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

```
expression1;
while ( expression2 ) {
    statement
    expression3;
}
```

4.4 for Repetition Statement (Cont.)

- There is an exception to this rule, which we discuss in Section 4.9.
- Often, *expression1* and *expression3* are comma-separated lists of expressions.
- The commas as used here are actually comma operators that guarantee that lists of expressions evaluate from left to right.
- The value and type of a comma-separated list of expressions are the value and type of the right-most expression in the list.

4.4 `for` Repetition Statement (Cont.)

- The comma operator is most often used in the `for` statement.
- Its primary use is to enable you to use multiple initialization and/or multiple increment expressions.
- For example, there may be two control variables in a single `for` statement that must be initialized and incremented.

Software Engineering Observation 4.1

Place only expressions involving the control variables in the initialization and increment sections of a `for` *statement. Manipulations of other variables should appear either before the loop (if they execute only once, like initialization statements) or in the loop body (if they execute once per repetition, like incrementing or decrementing statements).*

4.4 `for` Repetition Statement (Cont.)

- The three expressions in the `for` statement are optional.
- If *expression2* is omitted, C assumes that the condition is true, thus creating an infinite loop.
- One may omit *expression1* if the control variable is initialized elsewhere in the program.
- *expression3* may be omitted if the increment is calculated by statements in the body of the `for` statement or if no increment is needed.
- The increment expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`.

4.4 for Repetition Statement (Cont.)

- Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

  are all equivalent in the increment part of the `for` statement.
- Many C programmers prefer the form `counter++` because the incrementing occurs after the loop body is executed, and the postincrementing form seems more natural.
- Because the variable being preincremented or postincremented here does not appear in a larger expression, both forms of incrementing have the same effect.
- The two semicolons in the `for` statement are required.

Common Programming Error 4.3

Using commas instead of semicolons in a `for` *header is a syntax error.*

Common Programming Error 4.4

Placing a semicolon immediately to the right of a `for` *header makes the body of that* `for` *statement an empty statement. This is normally a logic error.*

4.5 for Statement: Notes and Observations

- The initialization, loop-continuation condition and increment can contain arithmetic expressions. For example, if x = 2 and y = 10, the statement

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

  is equivalent to the statement

```
for ( j = 2; j <= 80; j += 5 )
```

- The "increment" may be negative (in which case it's really a decrement and the loop actually counts downward).
- If the loop-continuation condition is initially false, the loop body does not execute. Instead, execution proceeds with the statement following the `for` statement.

4.5 for Statement: Notes and Observations (cont.)

- The control variable is frequently printed or used in calculations in the body of a loop, but it need not be. It's common to use the control variable for controlling repetition while never mentioning it in the body of the loop.
- The `for` statement is flowcharted much like the `while` statement. For example, Fig. 4.4 shows the flowchart of the `for` statement

```
for ( counter = 1; counter <= 10; counter++ )
   printf( "%d", counter );
```

- This flowchart makes it clear that the initialization occurs only once and that incrementing occurs after the body statement is performed.

Error-Prevention Tip 4.3

Although the value of the control variable can be changed in the body of a `for` *loop, this can lead to subtle errors. It's best not to change it.*

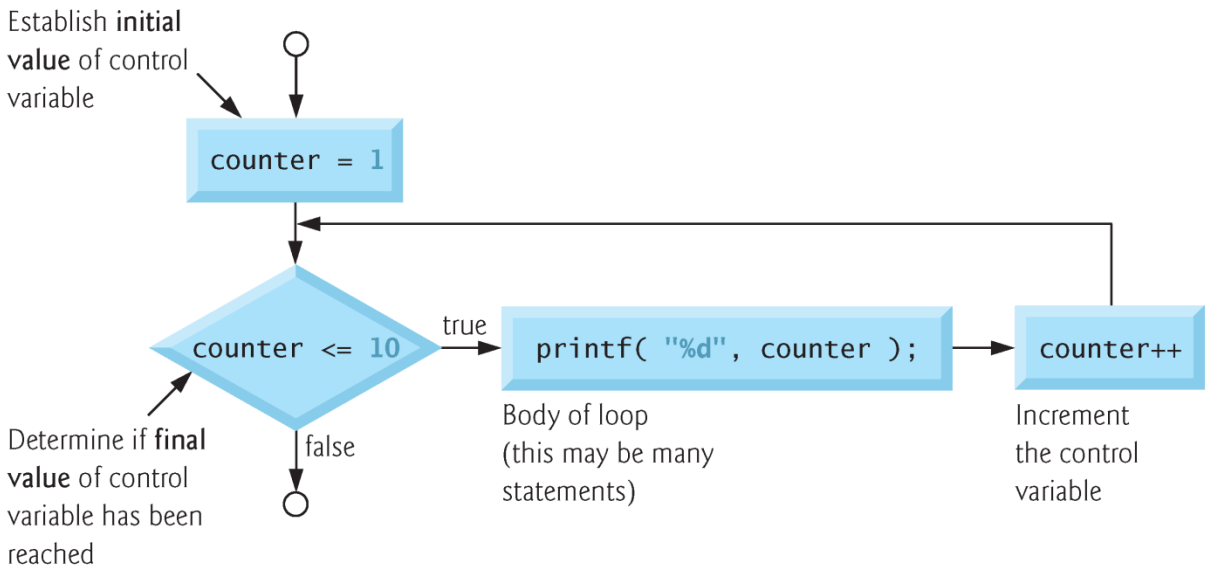

Fig. 4.4 | Flowcharting a typical `for` repetition statement.

4.6 Examples Using the `for` Statement

- The following examples show methods of varying the control variable in a `for` statement.
  - Vary the control variable from 1 to 100 in increments of 1.
    ```
    for ( i = 1; i <= 100; i++ )
    ```
  - Vary the control variable from 100 to 1 in increments of -1 (decrements of 1).
    ```
    for ( i = 100; i >= 1; i-- )
    ```
  - Vary the control variable from 7 to 77 in steps of 7.
    ```
    for ( i = 7; i <= 77; i += 7 )
    ```
  - Vary the control variable from 20 to 2 in steps of -2.
    ```
    for ( i = 20; i >= 2; i -= 2 )
    ```
  - Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.
    ```
    for ( j = 2; j <= 17; j += 3 )
    ```
  - Vary the control variable over the following sequence of values: 44, 33, 22, 11, 0.
    ```
    for ( j = 44; j >= 0; j -= 11 )
    ```

4.6 Examples Using the `for` Statement (Cont.)

- The next two examples provide simple applications of the `for` statement.
- Figure 4.5 uses the `for` statement to sum all the even integers from `2` to `100`.

```c
/* Fig. 4.5: fig04_05.c
   Summation with for */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int sum = 0; /* initialize sum */
   int number; /* number to be added to sum */

   for ( number = 2; number <= 100; number += 2 ) {
      sum += number; /* add number to sum */
   } /* end for */

   printf( "Sum is %d\n", sum ); /* output sum */
   return 0; /* indicate program ended successfully */
} /* end function main */
```

```
Sum is 2550
```

Fig. 4.5 | Using `for` to sum numbers.

4.6 Examples Using the `for` Statement (Cont.)

- The body of the `for` statement in Fig. 4.5 could actually be merged into the rightmost portion of the `for` header by using the comma operator as follows:

```
for ( number = 2; number <= 100; sum += number, number += 2 )
   ; /* empty statement */
```

- The initialization `sum = 0` could also be merged into the initialization section of the `for`.

Good Programming Practice 4.3

Although statements preceding a `for` *and statements in the body of a* `for` *can often be merged into the* `for` *header, avoid doing so because it makes the program more difficult to read.*

Good Programming Practice 4.4

Limit the size of control-statement headers to a single line if possible.

4.6 Examples Using the for Statement (Cont.)

- Consider the following problem statement:
  - A person invests $1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

    $a = p(1 + r)^n$

    where

    p is the original amount invested (i.e., the principal)
    r is the annual interest rate
    n is the number of years
    a is the amount on deposit at the end of the n^{th} year.
- This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit.
- The solution is shown in Fig. 4.6.

```
1  /* Fig. 4.6: fig04_06.c
2     Calculating compound interest */
3  #include <stdio.h>
4  #include <math.h>
5
6  /* function main begins program execution */
7  int main( void )
8  {
9     double amount; /* amount on deposit */
10    double principal = 1000.0; /* starting principal */
11    double rate = .05; /* annual interest rate */
12    int year; /* year counter */
13
14    /* output table column head */
15    printf( "%4s%21s\n", "Year", "Amount on deposit" );
16
17    /* calculate amount on deposit for each of ten years */
18    for ( year = 1; year <= 10; year++ ) {
19
20       /* calculate new amount for specified year */
21       amount = principal * pow( 1.0 + rate, year );
22
```

Fig. 4.6 | Calculating compound interest with `for`. (Part 1 of 2.)

```
23          /* output one table row */
24          printf( "%4d%21.2f\n", year, amount );
25       } /* end for */
26
27       return 0; /* indicate program ended successfully */
28    } /* end function main */
```

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Fig. 4.6 | Calculating compound interest with `for`. (Part 2 of 2.)

4.6 Examples Using the `for` Statement (Cont.)

- The `for` statement executes the body of the loop 10 times, varying a control variable from 1 to 10 in increments of 1.
- Although C does not include an exponentiation operator, we can use the Standard Library function `pow` for this purpose.
- The function `pow(x, y)` calculates the value of `x` raised to the `y`th power.
- It takes two arguments of type double and returns a `double` value.
- Type `double` is a floating-point type much like `float`, but typically a variable of type `double` can store a value of much greater magnitude with greater precision than `float`.

4.6 Examples Using the `for` Statement (Cont.)

- The header `<math.h>` (line 4) should be included whenever a math function such as `pow` is used.
- Actually, this program would malfunction without the inclusion of `math.h`, as the linker would be unable to find the `pow` function.
- Function `pow` requires two `double` arguments, but variable `year` is an integer.
- The `math.h` file includes information that tells the compiler to convert the value of `year` to a temporary `double` representation before calling the function.

4.6 Examples Using the `for` Statement (Cont.)

- This information is contained in something called `pow`'s function prototype.
- Function prototypes are explained in Chapter 5.
- We also provide a summary of the `pow` function and other math library functions in Chapter 5.
- Notice that we defined the variables `amount`, `principal` and `rate` to be of type `double`.
- We did this for simplicity because we're dealing with fractional parts of dollars.

Error-Prevention Tip 4.4

Do not use variables of type `float` *or* `double` *to perform monetary calculations. The impreciseness of floating-point numbers can cause errors that will result in incorrect monetary values. [In this chapter's exercises, we explore the use of integers to perform monetary calculations.]*

4.6 Examples Using the `for` Statement (Cont.)

- Here is a simple explanation of what can go wrong when using `float` or `double` to represent dollar amounts.
- Two `float` dollar amounts stored in the machine could be 14.234 (which with `%.2f` prints as 14.23) and 18.673 (which with `%.2f` prints as 18.67).
- When these amounts are added, they produce the sum 32.907, which with `%.2f` prints as 32.91.

4.6 Examples Using the `for` Statement (Cont.)

- Thus your printout could appear as
  - ```
      14.23
    + 18.67
    -------
      32.91
    ```
- Clearly the sum of the individual numbers as printed should be 32.90! You've been warned!
- The conversion specifier `%21.2f` is used to print the value of the variable `amount` in the program.
- The `21` in the conversion specifier denotes the field width in which the value will be printed.
    ```

4.6 Examples Using the for Statement (Cont.)

- A field width of 21 specifies that the value printed will appear in 21 print positions.
- The 2 specifies the precision (i.e., the number of decimal positions).
- If the number of characters displayed is less than the field width, then the value will automatically be right justified in the field.
- This is particularly useful for aligning floating-point values with the same precision (so that their decimal points align vertically).

4.6 Examples Using the `for` Statement (Cont.)

- To left justify a value in a field, place a `-` (minus sign) between the `%` and the field width.
- The minus sign may also be used to left justify integers (such as in `%-6d`) and character strings (such as in `%-8s`).

4.7 `switch` Multiple-Selection Statement

- Occasionally, an algorithm will contain a series of decisions in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken.
- This is called multiple selection.
- C provides the `switch` multiple-selection statement to handle such decision making.
- The `switch` statement consists of a series of `case` labels, an optional `default` case and statements to execute for each case.
- Figure 4.7 uses `switch` to count the number of each different letter grade students earned on an exam.

```
1  /* Fig. 4.7: fig04_07.c
2     Counting letter grades */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     int grade; /* one grade */
9     int aCount = 0; /* number of As */
10    int bCount = 0; /* number of Bs */
11    int cCount = 0; /* number of Cs */
12    int dCount = 0; /* number of Ds */
13    int fCount = 0; /* number of Fs */
14
15    printf(  "Enter the letter grades.\n"  );
16    printf(  "Enter the EOF character to end input.\n"  );
17
```

Fig. 4.7 | `switch` example. (Part 1 of 5.)

```
18      /* loop until user types end-of-file key sequence */
19      while ( ( grade = getchar() ) != EOF ) {
20
21         /* determine which grade was input */
22         switch ( grade ) { /* switch nested in while */
23
24            case 'A': /* grade was uppercase A */
25            case 'a': /* or lowercase a */
26               ++aCount; /* increment aCount */
27               break; /* necessary to exit switch */
28
29            case 'B': /* grade was uppercase B */
30            case 'b': /* or lowercase b */
31               ++bCount; /* increment bCount */
32               break; /* exit switch */
33
34            case 'C': /* grade was uppercase C */
35            case 'c': /* or lowercase c */
36               ++cCount; /* increment cCount */
37               break; /* exit switch */
38
```

Fig. 4.7 | `switch` example. (Part 2 of 5.)

```
39
40          case 'd': /* or lowercase d */
41             ++dCount; /* increment dCount */
42             break; /* exit switch */
43
44          case 'F': /* grade was uppercase F */
45          case 'f': /* or lowercase f */
46             ++fCount; /* increment fCount */
47             break; /* exit switch */
48
49          case '\n': /* ignore newlines, */
50          case '\t': /* tabs, */
51          case ' ': /* and spaces in input */
52             break; /* exit switch */
53
54          default: /* catch all other characters */
55             printf( "Incorrect letter grade entered." );
56             printf( " Enter a new grade.\n" );
57             break; /* optional; will exit switch anyway */
58       } /* end switch */
59    } /* end while */
60
```

Fig. 4.7 | `switch` example. (Part 3 of 5.)

```
61      /* output summary of results */
62      printf( "\nTotals for each letter grade are:\n" );
63      printf( "A: %d\n", aCount ); /* display number of A grades */
64      printf( "B: %d\n", bCount ); /* display number of B grades */
65      printf( "C: %d\n", cCount ); /* display number of C grades */
66      printf( "D: %d\n", dCount ); /* display number of D grades */
67      printf( "F: %d\n", fCount ); /* display number of F grades */c
68      return 0; /* indicate program ended successfully */
69   } /* end function main */
```

Fig. 4.7 | `switch` example. (Part 4 of 5.)

```
Enter the letter grades.
Enter the EOF character to end input.
a
b
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z ———— Not all systems display a representation of the EOF character

Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Fig. 4.7 | `switch` example. (Part 5 of 5.)

4.7 switch Multiple-Selection Statement (Cont.)

- In the program, the user enters letter grades for a class.
- In the `while` header (line 19),
  - `while ( ( grade = getchar() ) != EOF )`
- the parenthesized assignment `(grade = getchar())` executes first.
- The `getchar` function (from `<stdio.h>`) reads one character from the keyboard and stores that character in the integer variable `grade`.
- Characters are normally stored in variables of type `char`.
- However, an important feature of C is that characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer.

4.7 switch Multiple-Selection Statement (Cont.)

- Thus, we can treat a character as either an integer or a character, depending on its use.
- For example, the statement

```
printf( "The character (%c) has the value %d.\n", 'a', 'a' );
```

- uses the conversion specifiers %c and %d to print the character a and its integer value, respectively.
- The result is

```
The character (a) has the value 97.
```

- The integer 97 is the character's numerical representation in the computer.

4.7 `switch` Multiple-Selection Statement (Cont.)

- Many computers today use the ASCII (American Standard Code for Information Interchange) character set in which 97 represents the lowercase letter `'a'`.
- A list of the ASCII characters and their decimal values is presented in Appendix B.
- Characters can be read with `scanf` by using the conversion specifier `%c`.
- Assignments as a whole actually have a value.
- This value is assigned to the variable on the left side of =.
- The value of the assignment expression `grade = getchar()` is the character that is returned by `getchar` and assigned to the variable `grade`.

4.7 switch Multiple-Selection Statement (Cont.)

- The fact that assignments have values can be useful for setting several variables to the same value.
- For example,

```
a = b = c = 0;
```

- first evaluates the assignment `c = 0` (because the = operator associates from right to left).
- The variable `b` is then assigned the value of the assignment `c = 0` (which is 0).
- Then, the variable `a` is assigned the value of the assignment `b = (c = 0)` (which is also 0).
- In the program, the value of the assignment `grade = getchar()` is compared with the value of `EOF` (a symbol whose acronym stands for "end of file").

4.7 `switch` Multiple-Selection Statement (Cont.)

- We use `EOF` (which normally has the value `-1`) as the sentinel value.
- The user types a system-dependent keystroke combination to mean "end of file"—i.e., "I have no more data to enter." `EOF` is a symbolic integer constant defined in the `<stdio.h>` header (we'll see how symbolic constants are defined in Chapter 6).
- If the value assigned to `grade` is equal to `EOF`, the program terminates.
- Chose to represent characters in this program as `int`s because `EOF` has an integer value (normally `-1`).

Portability Tip 4.1

The keystroke combinations for entering EOF (end of file) are system dependent.

Portability Tip 4.2

Testing for the symbolic constant `EOF` *rather than* `-1` *makes programs more portable. The C standard states that* `EOF` *is a negative integral value (but not necessarily* `-1`*). Thus,* `EOF` *could have different values on different systems.*

4.7 switch Multiple-Selection Statement (Cont.)

- On Linux/UNIX/Mac OS X systems, the EOF indicator is entered by typing
  <Ctrl> d
- on a line by itself.
- This notation *<Ctrl> d* means to press the *Enter* key then simultaneously press both *Ctrl* and *d*.
- On other systems, such as Microsoft Windows, the EOF indicator can be entered by typing
  <Ctrl> z
- You may also need to press *Enter* on Windows.
- The user enters grades at the keyboard.

4.7 `switch` Multiple-Selection Statement (Cont.)

- When the *Enter* key is pressed, the characters are read by function `getchar` one character at a time.
- If the character entered is not equal to `EOF`, the `switch` statement (line 22) is entered.
- Keyword `switch` is followed by the variable name `grade` in parentheses.
- This is called the controlling expression.
- The value of this expression is compared with each of the `case` labels.
- Assume the user has entered the letter `C` as a grade.
- `C` is automatically compared to each `case` in the `switch`.

4.7 `switch` Multiple-Selection Statement (Cont.)

- If a match occurs (`case 'C':`), the statements for that `case` are executed.
- In the case of the letter `C`, `cCount` is incremented by `1` (line 36), and the `switch` statement is exited immediately with the `break` statement.
- The `break` statement causes program control to continue with the first statement after the `switch` statement.
- The `break` statement is used because the `case`s in a `switch` statement would otherwise run together.
- If `break` is not used anywhere in a `switch` statement, then each time a match occurs in the statement, the statements for all the remaining `case`s will be executed.

4.7 `switch` Multiple-Selection Statement (Cont.)

- (This feature is rarely useful, although it's perfect for programming the iterative song *The Twelve Days of Christmas!)*
- If no match occurs, the `default` case is executed, and an error message is printed.
- Each `case` can have one or more actions.
- The `switch` statement is different from all other control statements in that braces are not required around multiple actions in a `case` of a `switch`.
- The general `switch` multiple-selection statement (using a `break` in each `case`) is flowcharted in Fig. 4.8.
- The flowchart makes it clear that each `break` statement at the end of a `case` causes control to immediately exit the `switch` statement.

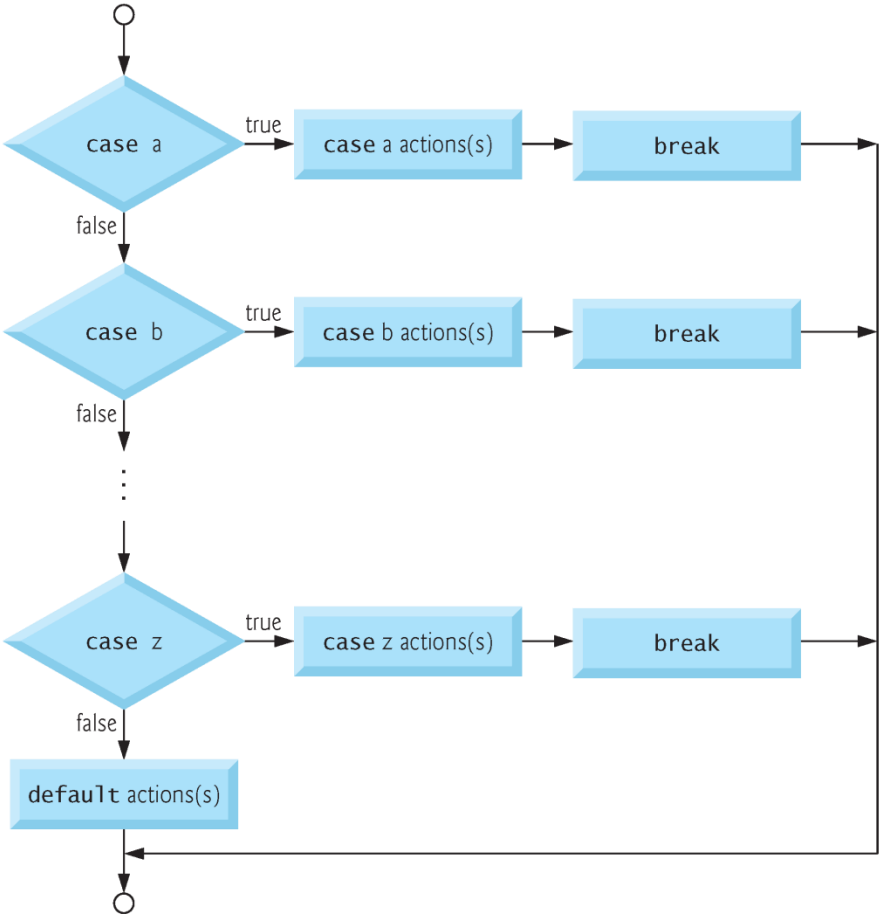

Fig. 4.8 | `switch` multiple-selection statement with `breaks`.

Common Programming Error 4.5

Forgetting a `break` *statement when one is needed in a* `switch` *statement is a logic error.*

Good Programming Practice 4.5

Provide a `default` *case in* `switch` *statements. Cases not explicitly tested in a* `switch` *are ignored. The* `default` *case helps prevent this by focusing the programmer on the need to process exceptional conditions. Sometimes no* `default` *processing is needed.*

Good Programming Practice 4.6

Although the `case` *clauses and the* `default` *case clause in a* `switch` *statement can occur in any order, it's considered good programming practice to place the* `default` *clause last.*

Good Programming Practice 4.7

In a `switch` *statement when the* `default` *clause is last, the* `break` *statement is not required. Some programmers include this* `break` *for clarity and symmetry with other* `case`*s.*

4.7 switch Multiple-Selection Statement (Cont.)

- In the `switch` statement of Fig. 4.7, the lines

```
case '\n': /* ignore newlines, */
case '\t': /* tabs, */
case ' ': /* and spaces in input */
   break; /* exit switch */
```

  cause the program to skip newline, tab and blank characters.
- Reading characters one at a time can cause some problems.

4.7 `switch` Multiple-Selection Statement (Cont.)

- To have the program read the characters, they must be sent to the computer by pressing the *Enter* key.
- This causes the newline character to be placed in the input after the character we wish to process.
- Often, this newline character must be specially processed to make the program work correctly.
- By including the preceding cases in our `switch` statement, we prevent the error message in the `default` case from being printed each time a newline, tab or space is encountered in the input.

Common Programming Error 4.6

Not processing newline characters in the input when reading characters one at a time can cause logic errors.

Error-Prevention Tip 4.5

Remember to provide processing capabilities for newline (and possibly other white-space) characters in the input when processing characters one at a time.

4.7 `switch` Multiple-Selection Statement (Cont.)

- Listing several case labels together (such as `case 'D': case 'd':` in Fig. 4.7) simply means that the same set of actions is to occur for either of these cases.
- When using the `switch` statement, remember that each individual `case` can test only a constant integral expression—i.e., any combination of character constants and integer constants that evaluates to a constant integer value.
- A character constant is represented as the specific character in single quotes, such as `'A'`.

4.7 `switch` Multiple-Selection Statement (Cont.)

- Characters must be enclosed within single quotes to be recognized as character constants—characters in double quotes are recognized as strings.
- Integer constants are simply integer values.
- In our example, we have used character constants.
- Remember that characters are represented as small integer values.

4.7 `switch` Multiple-Selection Statement (Cont.)

- Notes on integral types
  - Portable languages like C must have flexible data type sizes.
  - Different applications may need integers of different sizes.
  - C provides several data types to represent integers.
  - The range of values for each type depends on the particular computer's hardware.
  - In addition to `int` and `char`, C provides types `short` (an abbreviation of `short int`) and `long` (an abbreviation of `long int`).
  - C specifies that the minimum range of values for `short` integers is –32768 to +32767.
  - For the vast majority of integer calculations, `long` integers are sufficient.

4.7 `switch` Multiple-Selection Statement (Cont.)

- Notes on integral types
  - The standard specifies that the minimum range of values for `long` integers is –2147483648 to +2147483647.
  - The standard states that the range of values for an `int` is at least the same as the range for `short` integers and no larger than the range for `long` integers.
  - The data type `signed char` can be used to represent integers in the range –128 to +127 or any of the characters in the computer's character set.

4.8 do…while Repetition Statement

- The `do…while` repetition statement is similar to the `while` statement.
- In the `while` statement, the loop-continuation condition is tested at the beginning of the loop before the body of the loop is performed.
- The `do…while` statement tests the loop-continuation condition *after* the loop body is performed.
- Therefore, the loop body will be executed at least once.
- When a `do…while` terminates, execution continues with the statement after the `while` clause.

4.8 do…while Repetition Statement (Cont.)

- It's not necessary to use braces in the `do…while` statement if there is only one statement in the body.
- However, the braces are usually included to avoid confusion between the `while` and `do…while` statements.
- For example,

  `while` (*condition*)

- is normally regarded as the header to a `while` statement.

4.8 do…while Repetition Statement (Cont.)

- A `do…while` with no braces around the single-statement body appears as

```
do
   statement
while ( condition );
```

- which can be confusing.
- The last line—`while(` *condition* `);`—may be misinterpreted by as a `while` statement containing an empty statement.
- Thus, to avoid confusion, the `do…while` with one statement is often written as follows:

Good Programming Practice 4.8

To eliminate the potential for ambiguity, some programmers always include braces in a `do...while` *statement, even if the braces are not necessary.*

Common Programming Error 4.7

Infinite loops are caused when the loop-continuation condition in a `while`, `for` *or* `do...while` *statement never becomes false. To prevent this, make sure there is not a semicolon immediately after the header of a* `while` *or* `for` *statement. In a counter-controlled loop, make sure the control variable is incremented (or decremented) in the loop. In a sentinel-controlled loop, make sure the sentinel value is eventually input.*

4.8 do…while Repetition Statement (Cont.)

- Figure 4.9 uses a `do…while` statement to print the numbers from 1 to 10.
- The control variable `counter` is preincremented in the loop-continuation test.
- Note also the use of the braces to enclose the single-statement body of the `do…while`.

```c
/* Fig. 4.9: fig04_09.c
   Using the do/while repetition statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter = 1; /* initialize counter */

   do {
      printf( "%d  ", counter ); /* display counter */
   } while ( ++counter <= 10 ); /* end do...while */

   return 0; /* indicate program ended successfully */
} /* end function main */
```

```
1  2  3  4  5  6  7  8  9  10
```

Fig. 4.9 | do...while statement example.

4.8 do…while Repetition Statement (Cont.)

- Figure 4.10 shows the `do…while` statement flowchart, which makes it clear that the loop-continuation condition does not execute until after the action is performed at least once.

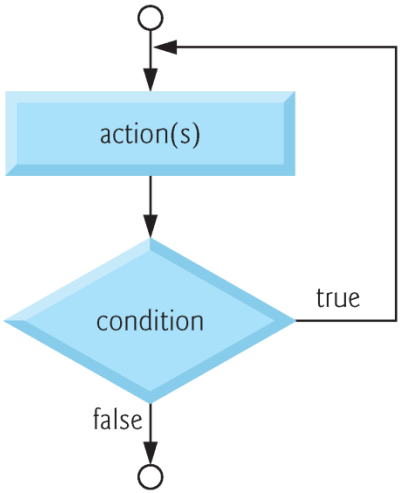

Fig. 4.10 | Flowcharting the `do...while` repetition statement.

4.9 break and continue Statements

- The `break` and `continue` statements are used to alter the flow of control.
- The `break` statement, when executed in a `while`, `for`, `do`…`while` or `switch` statement, causes an immediate exit from that statement.
- Program execution continues with the next statement.
- Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` statement (as in Fig. 4.7).

4.9 `break` and `continue` Statements (Cont.)

- Figure 4.11 demonstrates the break statement in a `for` repetition statement.
- When the `if` statement detects that `x` has become `5`, `break` is executed.
- This terminates the `for` statement, and the program continues with the `printf` after the `for`.
- The loop fully executes only four times.

```c
/* Fig. 4.11: fig04_11.c
   Using the break statement in a for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int x; /* counter */

   /* loop 10 times */
   for ( x = 1; x <= 10; x++ ) {

      /* if x is 5, terminate loop */
      if ( x == 5 ) {
         break; /* break loop only if x is 5 */
      } /* end if */

      printf( "%d ", x ); /* display value of x */
   } /* end for */

   printf( "\nBroke out of loop at x == %d\n", x );
   return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.11 | Using the `break` statement in a `for` statement. (Part 1 of 2.)

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.11 | Using the `break` statement in a `for` statement. (Part 2 of 2.)

4.9 break and continue Statements (Cont.)

- The `continue` statement, when executed in a `while`, `for` or `do…while` statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop.
- In `while` and `do…while` statements, the loop-continuation test is evaluated immediately after the `continue` statement is executed.
- In the `for` statement, the increment expression is executed, then the loop-continuation test is evaluated.

4.9 break and continue Statements (Cont.)

- Earlier, we said that the `while` statement could be used in most cases to represent the `for` statement.
- The one exception occurs when the increment expression in the `while` statement follows the `continue` statement.
- In this case, the increment is not executed before the repetition-continuation condition is tested, and the `while` does not execute in the same manner as the `for`.
- Figure 4.12 uses the `continue` statement in a `for` statement to skip the `printf` statement and begin the next iteration of the loop.

```c
/* Fig. 4.12: fig04_12.c
   Using the continue statement in a for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int x; /* counter */

   /* loop 10 times */
   for ( x = 1; x <= 10; x++ ) {

      /* if x is 5, continue with next iteration of loop */
      if ( x == 5 ) {
         continue; /* skip remaining code in loop body */
      } /* end if */

      printf( "%d ", x ); /* display value of x */
   } /* end for */

   printf( "\nUsed continue to skip printing the value 5\n" );
   return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.12 | Using the `continue` statement in a `for` statement. (Part 1 of 2.)

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 4.12 | Using the `continue` statement in a `for` statement. (Part 2 of 2.)

Software Engineering Observation 4.2

Some programmers feel that `break` *and* `continue` *violate the norms of structured programming. The effects of these statements can be achieved by structured programming techniques we'll soon learn, so these programmers do not use* `break` *and* `continue`.

Performance Tip 4.1

The `break` *and* `continue` *statements, when used properly, perform faster than the corresponding structured techniques that we'll soon learn.*

Software Engineering Observation 4.3

There is a tension between achieving quality software engineering and achieving the best-performing software. Often one of these goals is achieved at the expense of the other.

4.10 Logical Operators

- C provides logical operators that may be used to form more complex conditions by combining simple conditions.
- The logical operators are && (logical AND), || (logical OR) and ! (logical NOT also called logical negation).
- We'll consider examples of each of these operators.
- Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution.

4.10 Logical Operators (Cont.)

- In this case, we can use the logical operator && as follows:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

- This `if` statement contains two simple conditions.
- The condition `gender == 1` might be evaluated, for example, to determine if a person is a female.
- The condition `age >= 65` is evaluated to determine if a person is a senior citizen.
- The two simple conditions are evaluated first because the precedences of == and >= are both higher than the precedence of &&.

4.10 Logical Operators (Cont.)

- The `if` statement then considers the combined condition

```
gender == 1 && age >= 65
```

- This condition is true if and only if both of the simple conditions are true.
- Finally, if this combined condition is indeed true, then the count of `seniorFemales` is incremented by `1`.
- If either or both of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the `if`.
- Figure 4.13 summarizes the && operator.

4.10 Logical Operators (Cont.)

- The table shows all four possible combinations of zero (false) and nonzero (true) values for expression1 and expression2.
- Such tables are often called truth tables.
- C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1.
- Although C sets a true value to 1, it accepts any nonzero value as true.

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

Fig. 4.13 | Truth table for the logical AND (&&) operator.

4.10 Logical Operators (Cont.)

- Now let's consider the || (logical OR) operator.
- Suppose we wish to ensure at some point in a program that *either or both of two conditions are true before we choose a certain path of execution.*
- In this case, we use the || operator as in the following program segment

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   printf( "Student grade is A\n" );:
```

- This statement also contains two simple conditions.
- The condition `semesterAverage >= 90` is evaluated to determine if the student deserves an "A" in the course because of a solid performance throughout the semester.

4.10 Logical Operators (Cont.)

- The condition `finalExam >= 90` is evaluated to determine if the student deserves an "A" in the course because of an outstanding performance on the final exam.
- The `if` statement then considers the combined condition

```
semesterAverage >= 90 || finalExam >= 90
```

- and awards the student an "A" if either or both of the simple conditions are true.
- The message "`Student grade is A`" is not printed only when both of the simple conditions are false (zero).
- Figure 4.14 is a truth table for the logical OR operator (`||`).

expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Fig. 4.14 | Truth table for the logical OR (||) operator.

4.10 Logical Operators (Cont.)

- The && operator has a higher precedence than ||.
- Both operators associate from left to right.
- An expression containing && or || operators is evaluated only until truth or falsehood is known.
- Thus, evaluation of the condition

```
gender == 1 && age >= 65
```

- will stop if `gender` is not equal to `1` (i.e., the entire expression is false), and continue if `gender` is equal to `1` (i.e., the entire expression could still be true if `age >= 65`).
- This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.

Performance Tip 4.2

In expressions using operator &&, make the condition that is most likely to be false the leftmost condition. In expressions using operator ||, make the condition that is most likely to be true the leftmost condition. This can reduce a program's execution time.

4.10 Logical Operators (Cont.)

- C provides ! (logical negation) to enable a programmer to "reverse" the meaning of a condition.
- Unlike operators && and ||, which combine two conditions (and are therefore binary operators), the logical negation operator has only a single condition as an operand (and is therefore a unary operator).
- The logical negation operator is placed before a condition when we're interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
if ( !( grade == sentinelValue ) )
   printf( "The next grade is %f\n", grade );
```

- The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.
- Figure 4.15 is a truth table for the logical negation operator.

expression	!expression
0	1
nonzero	0

Fig. 4.15 | Truth table for operator ! (logical negation).

4.10 Logical Operators (Cont.)

- In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational operator.
- For example, the preceding statement may also be written as follows:

```
if ( grade != sentinelValue )
   printf( "The next grade is %f\n", grade );
```

- Figure 4.16 shows the precedence and associativity of the operators introduced to this point.
- The operators are shown from top to bottom in decreasing order of precedence.

Operators	Associativity	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - ! ++ *(prefix)* -- *(prefix)* (*type*)	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
&&	left to right	logical AND
||	left to right	logical OR
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment
,	left to right	comma

Fig. 4.16 | Operator precedence and associativity,

4.11 Confusing Equality (==) and Assignment (=) Operators

- There is one type of error that C programmers, no matter how experienced, tend to make so frequently that we felt it was worth a separate section.
- That error is accidentally swapping the operators == (equality) and = (assignment).
- What makes these swaps so damaging is the fact that they do not ordinarily cause compilation errors.
- Rather, statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through runtime logic errors.

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- Two aspects of C cause these problems.
- One is that any expression in C that produces a value can be used in the decision portion of any control statement.
- If the value is 0, it's treated as false, and if the value is nonzero, it's treated as true.
- The second is that assignments in C produce a value, namely the value that is assigned to the variable on the left side of the assignment operator.

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- For example, suppose we intend to write

```
if ( payCode == 4 )
   printf( "You get a bonus!" );
```

  but we accidentally write

```
if ( payCode = 4 )
   printf( "You get a bonus!" );
```

- The first `if` statement properly awards a bonus to the person whose paycode is equal to 4.
- The second `if` statement—the one with the error—evaluates the assignment expression in the `if` condition.

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- This expression is a simple assignment whose value is the constant 4.
- Because any nonzero value is interpreted as "true," the condition in this `if` statement is always true, and not only is the value of `payCode` inadvertantly set to 4, but the person always receives a bonus regardless of what the actual paycode is!

Common Programming Error 4.8

Using operator == for assignment or using operator = for equality is a logic error.

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- Programmers normally write conditions such as x == 7 with the variable name on the left and the constant on the right.
- By reversing these terms so that the constant is on the left and the variable name is on the right, as in 7 == x, the programmer who accidentally replaces the == operator with = is protected by the compiler.
- The compiler will treat this as a syntax error, because only a variable name can be placed on the left-hand side of an assignment expression.
- At least this will prevent the potential devastation of a runtime logic error.

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- Variable names are said to be *lvalues* (for "left values") because they can be used on the left side of an assignment operator.
- Constants are said to be *rvalues* (for "right values") because they can be used on only the right side of an assignment operator.
- *Lvalues* can also be used as *rvalues*, but not vice versa.

Good Programming Practice 4.9

When an equality expression has a variable and a constant, as in `x == 1`*, some programmers prefer to write the expression with the constant on the left and the variable name on the right (e.g.* `1 == x` *as protection against the logic error that occurs when you accidentally replace operator == with =).*

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- The other side of the coin can be equally unpleasant.
- Suppose you want to assign a value to a variable with a simple statement like

```
x = 1;
```

  but instead write

```
x == 1;
```

- Here, too, this is not a syntax error.
- Rather the compiler simply evaluates the conditional expression.

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- If x is equal to 1, the condition is true and the expression returns the value 1.
- If x is not equal to 1, the condition is false and the expression returns the value 0.
- Regardless of what value is returned, there is no assignment operator, so the value is simply lost, and the value of x remains unaltered, probably causing an execution-time logic error.
- Unfortunately, we do not have a handy trick available to help you with this problem! Many compilers, however, will issue a warning on such a statement.

Error-Prevention Tip 4.6

After you write a program, text search it for every = and check that it's used properly.

4.12 Structured Programming Summary

- Figure 4.17 summarizes the control statements discussed in Chapters 3 and 4.
- Small circles are used in the figure to indicate the single entry point and the single exit point of each statement.
- Connecting individual flowchart symbols arbitrarily can lead to unstructured programs.
- Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control statements, and to build only structured programs by properly combining control statements in two simple ways.

4.12 Structured Programming Summary (Cont.)

- For simplicity, only single-entry/single-exit control statements are used—there is only one way to enter and only one way to exit each control statement.
- Connecting control statements in sequence to form structured programs is simple—the exit point of one control statement is connected directly to the entry point of the next, i.e., the control statements are simply placed one after another in a program—we have called this "control-statement stacking."
- The rules for forming structured programs also allow for control statements to be nested.

4.12 Structured Programming Summary (Cont.)

- Figure 4.18 shows the rules for forming structured programs.
- The rules assume that the rectangle flowchart symbol may be used to indicate any action including input/output.
- Figure 4.19 shows the simplest flowchart.
- Applying the rules of Fig. 4.18 always results in a structured flowchart with a neat, building-block appearance.
- Repeatedly applying Rule 2 to the simplest flowchart (Fig. 4.19) results in a structured flowchart containing many rectangles in sequence (Fig. 4.20).
- Notice that Rule 2 generates a stack of control statements; so we call Rule 2 the stacking rule.

4.12 Structured Programming Summary (Cont.)

- Rule 3 is called the nesting rule.
- Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control statements.
- For example, in Fig. 4.21, the rectangle in the simplest flowchart is first replaced with a double-selection (`if...else`) statement.
- Then Rule 3 is applied again to both of the rectangles in the double-selection statement, replacing each of these rectangles with double-selection statements.
- The dashed box around each of the double-selection statements represents the rectangle that was replaced in the original flowchart.

4.12 Structured Programming Summary (Cont.)

- Rule 4 generates larger, more involved, and more deeply nested structures.
- The flowcharts that emerge from applying the rules in Fig. 4.18 constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.
- It's because of the elimination of the `goto` statement that these building blocks never overlap one another.
- The beauty of the structured approach is that we use only a small number of simple single-entry/single-exit pieces, and we assemble them in only two simple ways.

4.12 Structured Programming Summary (Cont.)

- Figure 4.22 shows the kinds of stacked building blocks that emerge from applying Rule 2 and the kinds of nested building blocks that emerge from applying Rule 3.
- The figure also shows the kind of overlapped building blocks that cannot appear in structured flowcharts (because of the elimination of the `goto` statement).

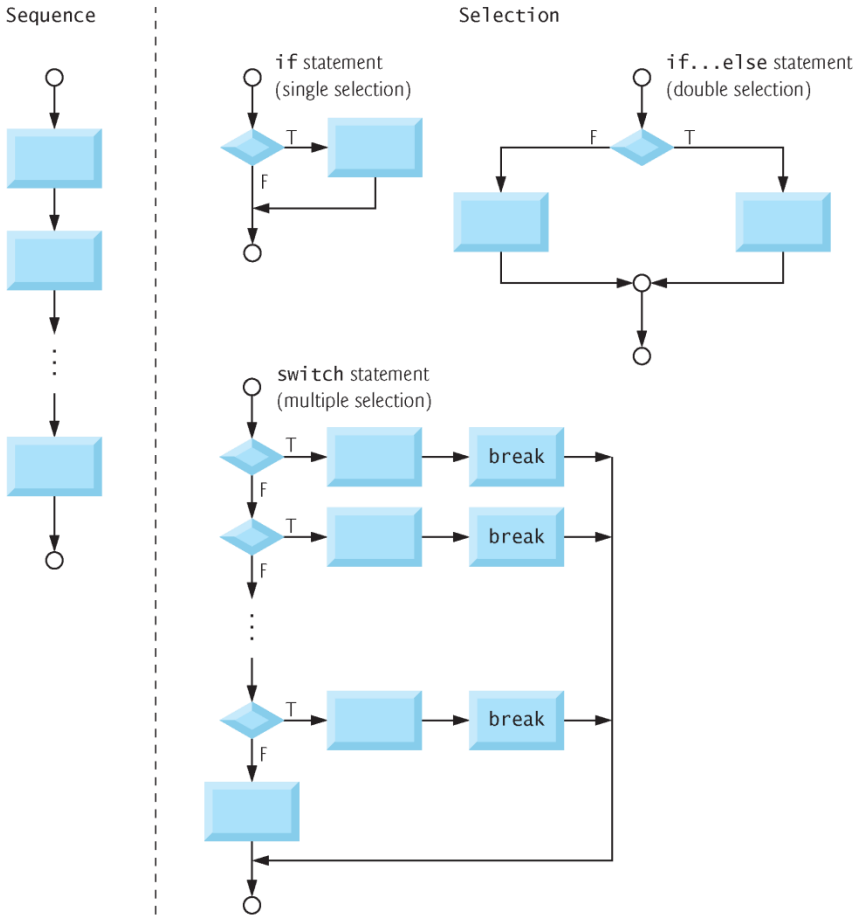

Fig. 4.17 | C's single-entry/single-exit sequence, selection and repetition statements. (Part 1 of 2.)

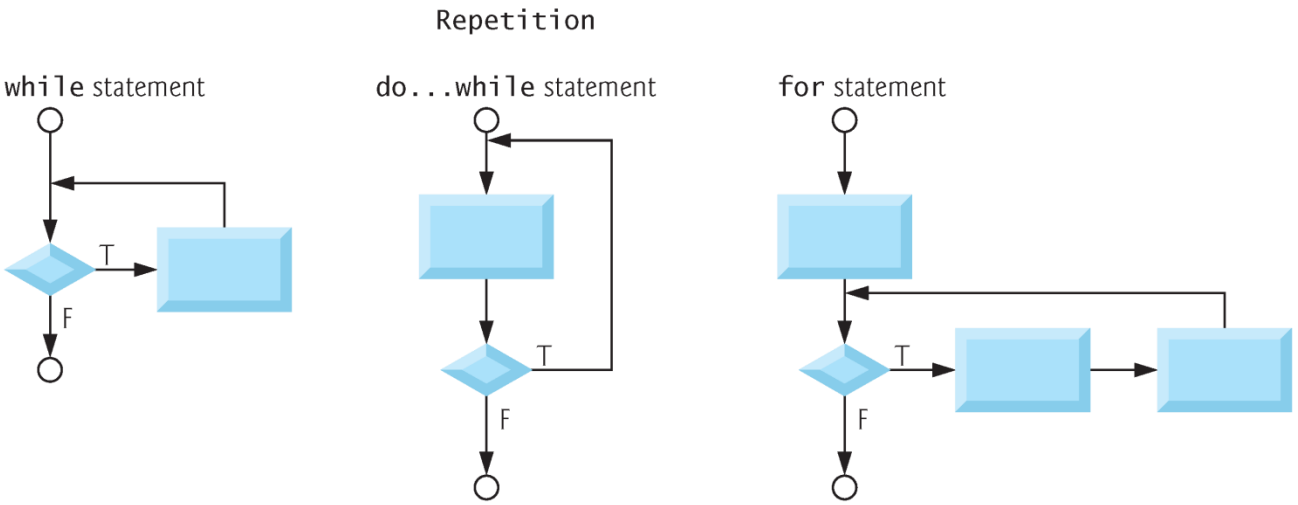

Fig. 4.17 | C's single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

Rules for Forming Structured Programs

1)Begin with the "simplest flowchart" (Fig. 4.19).

2)Any rectangle (action) can be replaced by two rectangles (actions) in sequence.

3)Any rectangle (action) can be rep'laced by any control statement (sequence, `if`, `if...else`, `switch`, `while`, `do...while` or `for`).

4)Rules 2 and 3 may be applied as often as you like and in any order.

Fig. 4.18 | Rules for forming structured programs.

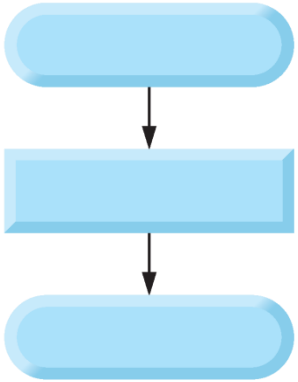

Fig. 4.19 | Simplest flowchart.

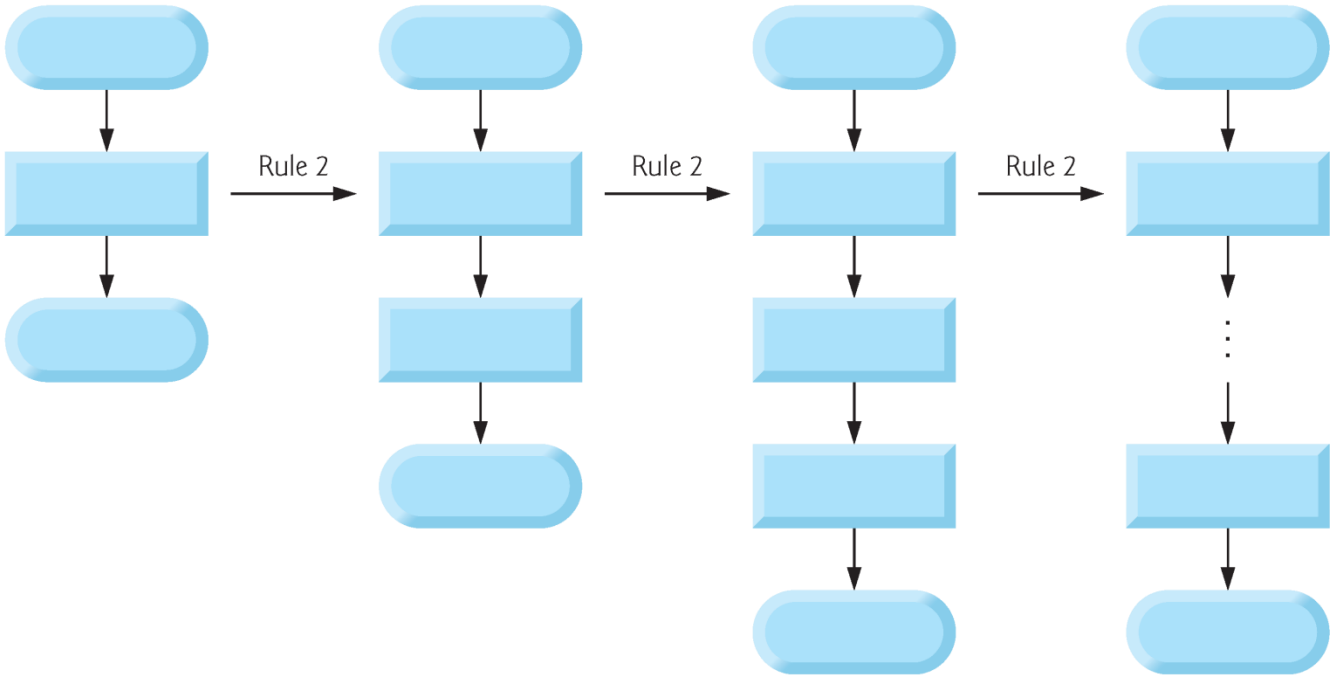

Fig. 4.20 | Repeatedly applying Rule 2 of Fig. 4.18 to the simplest flowchart.

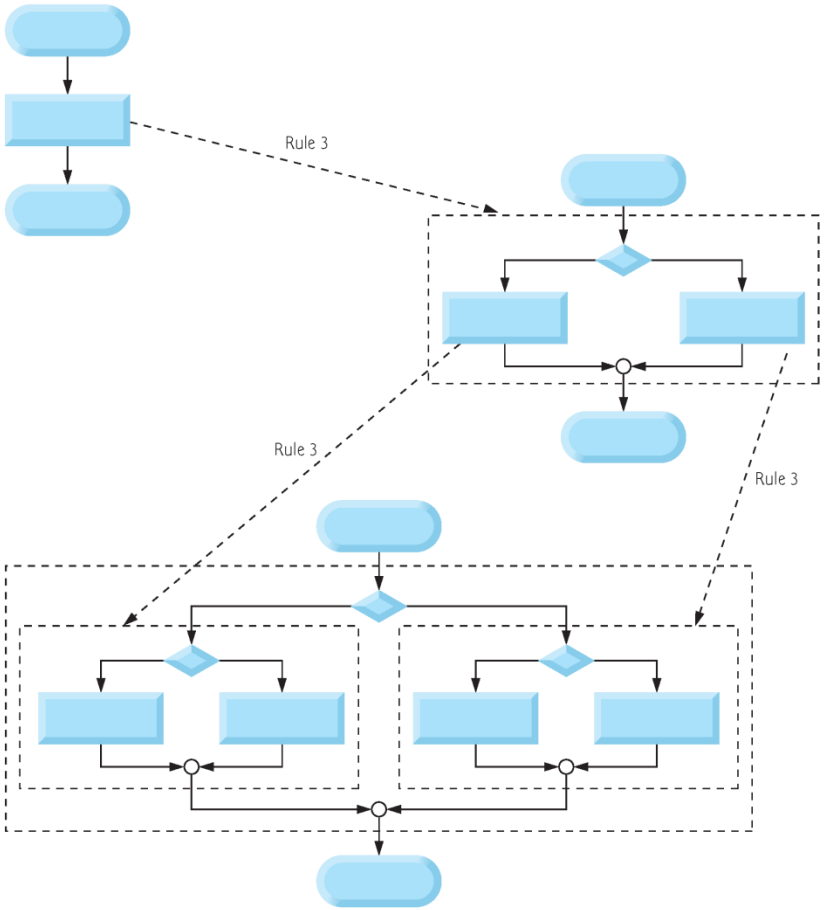

Fig. 4.21 | Applying Rule 3 of Fig. 4.18 to the simplest flowchart.

4.12 Structured Programming Summary (Cont.)

- If the rules in Fig. 4.18 are followed, an unstructured flowchart (such as that in Fig. 4.23) cannot be created.
- If you're uncertain whether a particular flowchart is structured, apply the rules of Fig. 4.18 in reverse to try to reduce the flowchart to the simplest flowchart.
- If you succeed, the original flowchart is structured; otherwise, it's not.
- Structured programming promotes simplicity.
- Bohm and Jacopini showed that only three forms of control are needed:
  - Sequence
  - Selection
  - Repetition

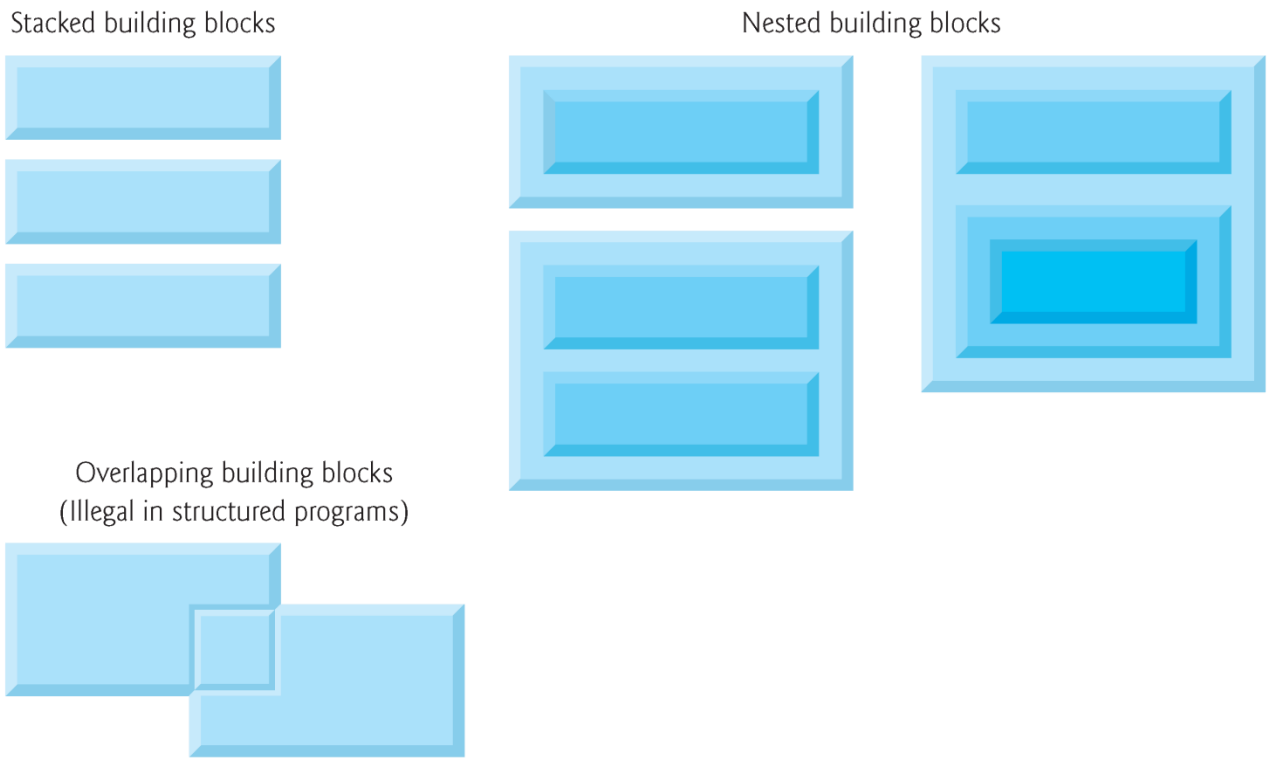

Fig. 4.22 | Stacked, nested and overlapped building blocks.

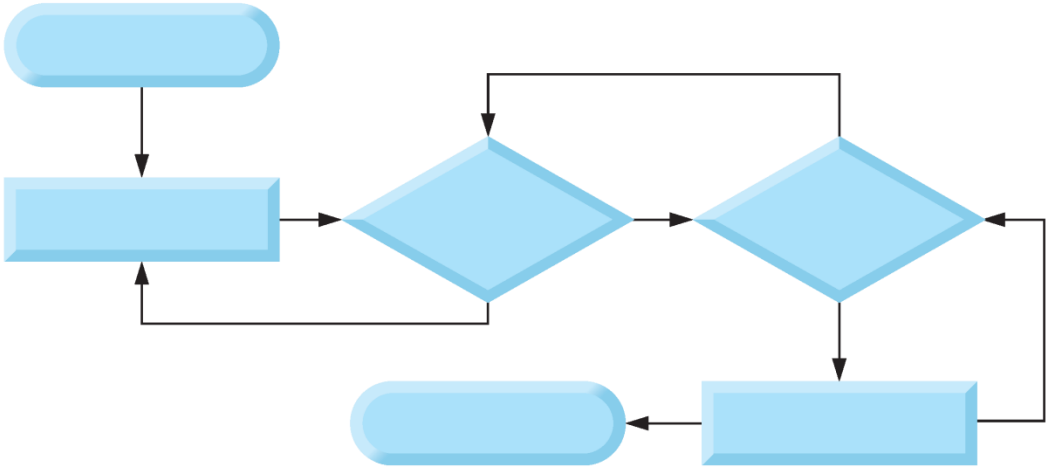

Fig. 4.23 | An unstructured flowchart.

4.12 Structured Programming Summary (Cont.)

- Sequence is straighforward.
- Selection is implemented in one of three ways:
  - `if` statement (single selection)
  - `if...else` statement (double selection)
  - `switch` statement (multiple selection)
- In fact, it's straightforward to prove that the simple `if` statement is sufficient to provide any form of selection—everything that can be done with the `if...else` statement and the `switch` statement can be implemented with one or more `if` statements.

4.12 Structured Programming Summary (Cont.)

- Repetition is implemented in one of three ways:
  - `while` statement
  - `do…while` statement
  - `for` statement
- It's straightforward to prove that the `while` statement is sufficient to provide any form of repetition.
- Everything that can be done with the `do…while` statement and the `for` statement can be done with the `while` statement.

4.12 Structured Programming Summary (Cont.)

- Combining these results illustrates that any form of control ever needed in a C program can be expressed in terms of only three forms of control:
  - sequence
  - `if` statement (selection)
  - `while` statement (repetition)
- And these control statements can be combined in only two ways—stacking and nesting.
- Indeed, structured programming promotes simplicity.

4.12 Structured Programming Summary (Cont.)

- In Chapters 3 and 4, we discussed how to compose programs from control statements containing actions and decisions.
- In Chapter 5, we introduce another program structuring unit called the function.
- We'll learn to compose large programs by combining functions, which, in turn, are composed of control statements.
- We'll also discuss how using functions promotes software reusability.