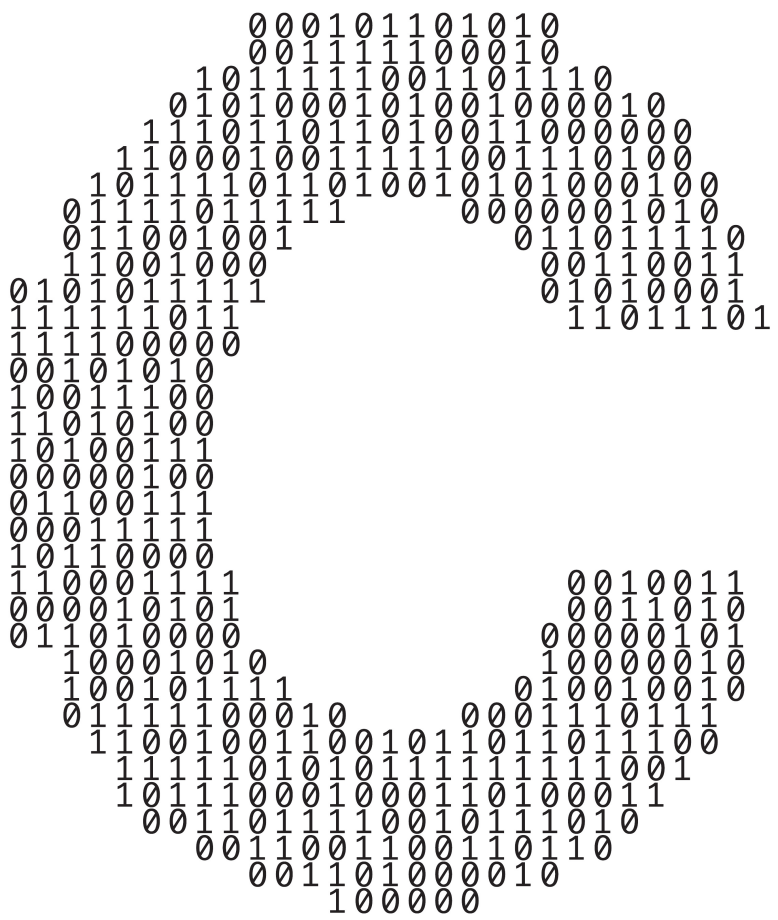
4

Program Control



Objectives

In this chapter, you'll:

- Learn the essentials of counter-controlled iteration.
- Use the `for` and `do...while` iteration statements to execute statements repeatedly.
- Understand multiple selection using the `switch` selection statement.
- Use the `break` and `continue` statements to alter the flow of control.
- Use logical operators to form complex conditions in control statements.
- Avoid the consequences of confusing the equality and assignment operators.

Outline

4.1 Introduction

You should now be comfortable with reading and writing simple C programs. Next, we consider iteration in more detail and introduce C's `for` and `do...while` iteration statements. We also introduce:

- the `switch` multiple-selection statement,
- the `break` statement for exiting immediately from certain control statements, and
- the `continue` statement for skipping the remainder of an iteration statement's body then proceeding with the next iteration of the loop.

We also discuss logical operators used for combining conditions and summarizes the principles of structured programming as presented here and in Chapter 3.

4.2 Iteration Essentials

Most programs involve iteration (or looping). A loop is a group of instructions the computer repeatedly executes while some **loop-continuation condition** remains true. We've discussed two means of iteration:

1. Counter-controlled iteration.
2. Sentinel-controlled iteration.

You saw in Chapter 3 that counter-controlled iteration uses a **control variable** to count the number of iterations for a group of instructions to perform. When the control variable's value indicates that the correct number of iterations has been completed, the loop terminates, and execution continues with the statement after the iteration statement.

You saw in Chapter 3 that we use sentinel values to control iteration if the precise number of iterations isn't known in advance, and the loop includes statements that obtain data each time the loop is performed. A 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.

Self Check

1 *(Fill-In)* A loop is a group of instructions the computer repeatedly executes while a _____ condition remains *true*.

Answer: loop-continuation.

2 *(Multiple Choice)* Which of the following statements is *false*?

a) Sentinel values are used to control iteration when the precise number of iterations isn't known in advance, and the loop includes statements that obtain data each time the loop is performed.
b) The sentinel value indicates "end of data."
c) The sentinel is entered after all regular data items have been supplied.
d) The sentinel must match a regular data item.

Answer: d) is *false*. Actually, the sentinel must be distinct from regular data items.

4.3 Counter-Controlled Iteration

Counter-controlled iteration requires:

- the **name** of a control variable,
- the **initial value** of the control variable,
- the **increment** (or **decrement**) by which the control variable is modified in each iteration, and
- the loop-continuation condition that tests for the **final value** of the control variable to determine whether looping should continue.

Consider Fig. 4.1, which displays the numbers from 1 through 5. The definition

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

names the control variable (`counter`), defines it as an integer, reserves memory space for it, and sets its initial value to `1`.

```
1  // fig04_01.c
2  // Counter-controlled iteration.
3  #include <stdio.h>
4
5  int main(void) {
6     int counter = 1; // initialization
7
8     while (counter <= 5) { // iteration condition
9        printf("%d  ", counter);
10       ++counter; // increment
11    }
12
13    puts("");
14 }
```

```
1  2  3  4  5
```

Fig. 4.1 | Counter-controlled iteration.

The statement

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

increments `counter` by 1 at the end of each loop iteration. The `while`'s condition

```
counter <= 5
```

tests whether the value of the control variable is less than or equal to `5` (the last value for which the condition is true). This `while` terminates when the control variable exceeds `5` (i.e., `counter` becomes `6`).

Use Integer Counters

Floating-point values may be approximate, so controlling counting loops with floating-point variables may result in imprecise counter values and inaccurate termination tests. For this reason, you should always control counting loops with integer values.

Self Check

1 *(Multiple Choice)* Which of the following a), b) or c) is required by counter-controlled iteration?

a) The name and initial value of a control variable (or loop counter).
b) The increment (or decrement) by which the control variable is modified each time through the loop.
c) The loop-continuation condition that tests for the final value of the control variable to determine whether looping should continue.
d) All of the above are required by counter-controlled iteration.

Answer: d.

2 *(Multiple Choice)* Based on this section's program, which of the following statements a), b) or c) is *false*?

a) Control variable `counter` increments by 1 during each iteration of the loop.
b) The loop terminates when `counter` is `5`.
c) The `while`'s body executes even when the control variable is `5`.
d) All of the above statements are *true*.

Answer: b) is *false*. Actually, the loop terminates when the control variable becomes `6`.

4.4 for Iteration Statement

The `for` iteration statement (lines 8–10 of Fig. 4.2) handles all the details of counter-controlled iteration. For readability, try to fit the `for` statement's header (line 8) on one line. The `for` statement executes as follows:

- When it begins executing, the `for` statement defines the control variable `counter` and initializes it to `1`.
- Next, it tests its loop-continuation condition `counter <= 5`. The initial value of `counter` is `1`, so the condition is *true*, and the `for` statement executes its `printf` statement (line 9) to display `counter`'s value, namely `1`.

- Next, the `for` statement increments the control variable `counter` using the expression `++counter`, then re-tests the loop-continuation condition. The control variable is now equal to `2`, so the condition is still *true*, and the `for` statement executes its `printf` statement again.
- This process continues until the control variable `counter` becomes `6`. At this point, the loop-continuation condition is *false* and iteration terminates.

The program continues executing with the first statement after the `for` (line 12).

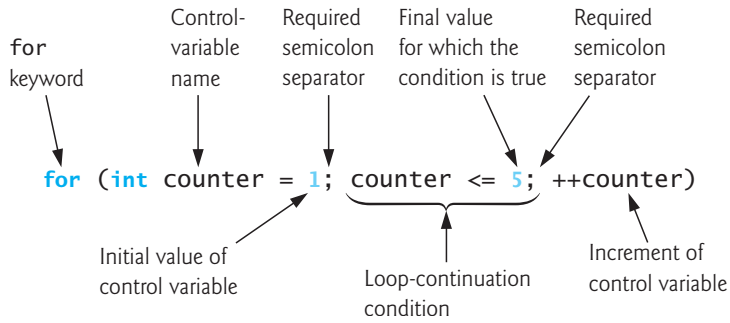
```
1  // fig04_02.c
2  // Counter-controlled iteration with the for statement.
3  #include <stdio.h>
4
5  int main(void) {
6     // initialization, iteration condition, and increment
7     // are all included in the for statement header.
8     for (int counter = 1; counter <= 5; ++counter) {
9        printf("%d  ", counter);
10    }
11
12    puts(""); // outputs a newline
13 }
```

```
1  2  3  4  5
```

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

for Statement Header Components

The following diagram takes a closer look at Fig. 4.2's `for` statement, which specifies each of the items needed for counter-controlled iteration. If there's more than one statement in the `for`'s body, braces are required. As with the other control statements, always place a `for` statement's body in braces, even if it has only one statement.



Control Variables Defined in a for Header Exist Only Until the Loop Terminates

When you define the control variable in the `for` header before the first semicolon (`;`), as in line 8 of Fig. 4.2:

```
for (int counter = 1; counter <= 5; ++counter) {
```

the control variable exists only until the loop terminates. So, attempting to access the control variable after the `for` statement's closing right brace (`}`) is a compilation error.

ERR

Off-By-One Errors

If we had written the loop-continuation condition `counter <= 5` as `counter < 5`, then the loop would be executed only four times. This is a common logic error called an **off-by-one error**. Using a control variable's final value in a `while` or `for` statement condition and using the `<=` relational operator can help avoid off-by-one errors. To print the values 1 to 5, for example, the loop-continuation condition should be `counter <= 5` rather than `counter < 6`.

ERR

General Format of a `for` Statement

The general format of the `for` statement is

```
for (initialization; loopContinuationCondition; increment) {
   statement
}
```

where

- *initialization* names the loop's control variable and provides its initial value,
- *loopContinuationCondition* determines whether the loop should continue executing, and
- *increment* modifies the control variable's value after executing the *statement* so that the loop-continuation condition eventually becomes *false*.

The two semicolons in the `for` header are required. If the loop-continuation condition is initially *false*, the program does not execute the `for` statement's body. Instead, execution proceeds with the statement following the `for`.

ERR

Infinite loops occur when the loop-continuation condition never becomes *false*. To prevent infinite loops, ensure that you do not place a semicolon immediately after a `while` statement's header. In a counter-controlled loop, ensure that you increment (or decrement) the control variable so the loop-continuation condition eventually becomes *false*. In a sentinel-controlled loop, ensure that the sentinel value is eventually input.

Expressions in the `for` Statement's Header Are Optional

All three expressions in a `for` header are optional. If you omit the *loopContinuationCondition*, the condition is always true, thus creating an infinite loop. You might omit the *initialization* expression if the program initializes the control variable before the loop. You might omit the *increment* expression if the program calculates the increment in the loop's body or if no increment is needed.

Increment Expression Acts Like a Standalone Statement

The `for` statement's *increment* acts like a standalone C statement at the end of the `for`'s body. So, the following are all equivalent in a `for` statement's increment expression:

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

The increment in a `for` statement's *increment* expression may be negative, in which case it's a *decrement*, and the loop counts *downward*.

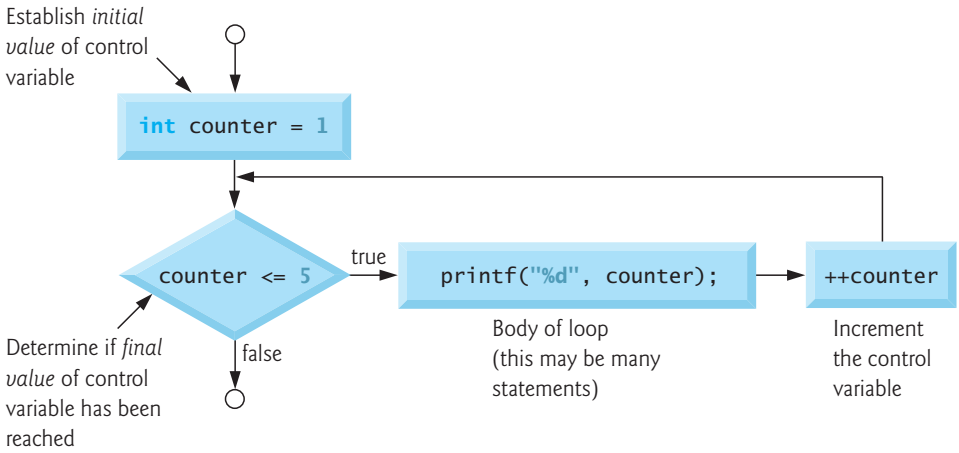
Using a `for` Statement's Control Variable in the Statement's Body

Programs frequently display the control-variable value or use it in calculations in the loop body, but this use is not required. The control variable is commonly used to control iteration without being mentioned in the `for` statement's body. Although the control variable's value can be changed in a `for` loop's body, avoid doing so, because this practice can lead to subtle errors. It's best not to change it.

ERR

`for` Statement Flowchart

Below is the flowchart for the `for` statement in Fig. 4.2:



This flowchart makes it clear that the initialization occurs once, and the increment occurs *after* the body statement each time it's performed.

Self Check

1 ***(True/False)*** When you define the control variable in the `for` header before the first semicolon (`;`), the control variable exists only until the loop terminates.
Answer: True.

2 ***(Multiple Choice)*** Which of the following statements a), b) or c) is *true*?

a) The `for` statement header specifies each of the items needed for counter-controlled iteration with a control variable.
b) If there's more than one statement in a `for`'s body, braces are required.
c) You should always place a control statement's body in braces, even if it has only one statement.
d) All of the above statements are *true*.

Answer: d.

4.5 Examples Using the for Statement

The following examples show ways to vary the control variable in a `for` statement.

1. Vary the control variable from `1` to `100` in increments of `1`.

```
for (int i = 1; i <= 100; ++i)
```

2. Vary the control variable from `100` to `1` in increments of `-1` (i.e., *decrements* of `1`).

```
for (int i = 100; i >= 1; --i)
```

3. Vary the control variable from `7` to `77` in increments of `7`.

```
for (int i = 7; i <= 77; i += 7)
```

4. Vary the control variable from `20` to `2` in increments of `-2`.

```
for (int i = 20; i >= 2; i -= 2)
```

5. Vary the control variable over the values `2`, `5`, `8`, `11`, `14` and `17`.

```
for (int j = 2; j <= 17; j += 3)
```

6. Vary the control variable over the following sequence of values: `44`, `33`, `22`, `11`, `0`.

```
for (int j = 44; j >= 0; j -= 11)
```

Application: Summing the Even Integers from 2 to 100

Figure 4.3 uses the `for` statement to sum the even integers from `2` to `100`. Each loop iteration (lines 8–10) adds the control variable `number`'s current value to the `sum`.

```
1  // fig04_03.c
2  // Summation with for.
3  #include <stdio.h>
4
5  int main(void) {
6     int sum = 0; // initialize sum
7
8     for (int number = 2; number <= 100; number += 2) {
9        sum += number; // add number to sum
10    }
11
12    printf("Sum is %d\n", sum);
13 }
```

```
Sum is 2550
```

Fig. 4.3 | Summation with `for`.

Application: Compound-Interest Calculations

The next example computes compound interest using the `for` statement. Consider the following problem statement:

> *A person invests $1000.00 in a savings account yielding 5% interest. Assuming 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, $1000.00 here),
r is the annual interest rate (for example, .05 for 5%),
n is the number of years, which is 10 here, and
a is the amount on deposit at the end of the n^{th} year.

This problem's solution (Fig. 4.4) uses a counter-controlled loop to perform the indicated calculation for each of the 10 years the money remains on deposit. The `for` statement executes the body of the loop 10 times, varying a control variable from 1 to 10 in increments of 1. C does *not* include an exponentiation operator, but we can use the Standard Library function `pow` (line 17) for this purpose. The call `pow(x, y)` calculates `x` raised to the `y`th power. The function takes two arguments of data type `double`. When it completes its calculation, `pow` returns (that is, gives back) a `double` value, which we then multiply by `principal` (line 17).

```
1  // fig04_04.c
2  // Calculating compound interest.
3  #include <stdio.h>
4  #include <math.h>
5
6  int main(void) {
7     double principal = 1000.0; // starting principal
8     double rate = 0.05; // annual interest rate
9
10    // output table column heads
11    printf("%4s%21s\n", "Year", "Amount on deposit");
12
13    // calculate amount on deposit for each of ten years
14    for (int year = 1; year <= 10; ++year) {
15
16       // calculate new amount for specified year
17       double amount = principal * pow(1.0 + rate, year);
18
19       // output one table row
20       printf("%4d%21.2f\n", year, amount);
21    }
22 }
```

```
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. 4.4 | Calculating compound interest.

You must include the header `<math.h>` (line 4) to use `pow` and C's other math functions.[1] If you did not include the header, this program would malfunction, 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 `year` value to a temporary `double` representation before calling the function. This information is contained in `pow`'s **function prototype**. We explain function prototypes in Chapter 5, where we also summarize many other math library functions.

Formatting Numeric Output

This program used the conversion specification `%21.2f` to print variable `amount`'s value. The `21` in the conversion specification denotes the **field width** in which the value will be printed. A field width of `21` specifies that the value printed will use `21` character positions. As you learned in Chapter 3, 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 be right-aligned with leading spaces. This is particularly useful for aligning the decimal points of floating-point values vertically. To left-align a value in a field, place a `-` (minus sign) between the `%` and the field width. We'll discuss the powerful formatting capabilities of `printf` and `scanf` in detail in Chapter 9.

Floating-Point Number Precision and Memory Requirements

Variables of type **`float`** typically require four bytes of memory with approximately seven significant digits. Variables of type `double` typically require eight bytes of memory with approximately 15 significant digits—about double the precision of `float`s. Most programmers use type `double`. C treats floating-point values such as 3.14159 as type `double` by default. Such values in the source code are known as **floating-point literals**.

C also has type `long double`. Such variables typically are stored in 12 or 16 bytes of memory. The C standard states the minimum sizes of each floating-point type and indicates that type `double` provides at least as much precision as `float` and that type `long double` provides at least as much precision as `double`. For a list of C's fundamental numeric types and their typical ranges, see

```
https://en.cppreference.com/w/c/language/arithmetic_types
```

Floating-Point Numbers Are Approximations

In conventional arithmetic, floating-point numbers often arise as a result of division—when we divide 10 by 3, the result is the infinitely repeating sequence 3.3333333… with the sequence of 3s repeating infinitely. The computer allocates only a fixed amount of space to hold such a value, so the stored floating-point value can be only an approximation. So, C's floating-point types suffer from what is referred to as **representational error**. Assuming that floating-point numbers are represented exactly (e.g., using them in comparisons for equality) can lead to incorrect results.

---

1. For the `gcc` compiler, you must include the `-lm` option (e.g., `gcc -lm fig04_04.c`) when compiling Fig. 4.4. This links the math library to the program.

Floating-point numbers have numerous applications, especially for measured values. For example, when we speak of a "normal" body temperature of 98.6 degrees Fahrenheit, we do not need to be precise to a large number of digits. When we read the temperature on a thermometer as 98.6, it actually might be 98.5999473210643. Calling this number 98.6 is fine for most applications involving body temperatures.

A Warning about Displaying Rounded Values

We declared variables `amount`, `principal` and `rate` to be of type `double` in this example. We're dealing with fractional parts of dollars and thus need a type that allows decimal points in its values. Unfortunately, floating-point numbers can cause trouble. Here's a simple explanation of what can go wrong when using floating-point numbers to represent dollar amounts displayed with two digits to the right of the decimal point. Two calculated dollar amounts stored in the machine could be 14.234 (rounded to 14.23 for display purposes) and 18.673 (rounded to 18.67 for display purposes). When these amounts are added, they produce the internal sum 32.907, which would typically be rounded to 32.91 for display purposes. Thus, your output could appear as

```
  14.23
+ 18.67
  32.91
```

but a person adding the individual numbers as displayed would expect the sum to be 32.90. You've been warned!

Even Common Dollar Amounts Can Have Floating-Point Representational Errors

Even simple dollar amounts, such as those you might see on a grocery or restaurant bill, can have representational errors when they're stored as `double`s. To see this, we created a simple program with the declaration

```
double d = 123.02;
```

then displayed `d`'s value with many digits of precision to the right of the decimal point. The output showed 123.02 as 123.0199999…, which is another example of a representational error. Though some dollar amounts can be represented precisely as `double`, many cannot. This is a common problem in many programming languages.

Self Check

1 *(Fill-In)* To _____ a value in a field, place a - (minus sign) between the `%` and the field width.
Answer: left-align.

2 *(Multiple Choice)* Which of the following `for` statement headers is incorrect?

a) Vary the control variable from `1` to `100` in increments of `1`.

```
for (int i = 1; i <= 100; ++i)
```

b) Vary the control variable from `7` to `77` in increments of `7`.

```
for (int i = 7; i <= 77; i += 7)
```

c) Vary the control variable over the following sequence of values: 2, 5, 8, 11, 15, 17.

```
for (int j = 2; j <= 17; j += 3)
```

d) Vary the control variable from 20 to 2 in increments of -2.

```
for (int i = 20; i >= 2; i -= 2)
```

Answer: c) is incorrect. The `for` statement actually generates the sequence 2, 5, 8, 11, 14, 17. It does not generate the value 15 in the original series.

4.6 `switch` Multiple-Selection Statement

In Chapter 3, we discussed the `if` single-selection and the `if...else` double-selection statements. Occasionally, an algorithm will contain a series of decisions that test a variable or expression separately for each of the integer values it may assume, then perform different actions. 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 4.5 uses `switch` to count the number of each different letter grade students earned on an exam.

```
1  // fig04_05.c
2  // Counting letter grades with switch.
3  #include <stdio.h>
4
5  int main(void) {
6     int aCount = 0;
7     int bCount = 0;
8     int cCount = 0;
9     int dCount = 0;
10    int fCount = 0;
11
12    puts("Enter the letter grades.");
13    puts("Enter the EOF character to end input.");
14    int grade = 0; // one grade
15
16    // loop until user types end-of-file key sequence
17    while ((grade = getchar()) != EOF) {
18
19       // determine which grade was input
20       switch (grade) { // switch nested in while
21          case 'A': // grade was uppercase A
22          case 'a': // or lowercase a
23             ++aCount;
24             break; // necessary to exit switch
25          case 'B': // grade was uppercase B
26          case 'b': // or lowercase b
27             ++bCount;
28             break;
```

Fig. 4.5 | Counting letter grades with `switch`. (Part 1 of 2.)

```
29            case 'C': // grade was uppercase C
30            case 'c': // or lowercase c
31               ++cCount;
32               break;
33            case 'D': // grade was uppercase D
34            case 'd': // or lowercase d
35               ++dCount;
36               break;
37            case 'F': // grade was uppercase F
38            case 'f': // or lowercase f
39               ++fCount;
40               break;
41            case '\n': // ignore newlines,
42            case '\t': // tabs,
43            case ' ': // and spaces in input
44               break;
45            default: // catch all other characters
46               printf("%s", "Incorrect letter grade entered.");
47               puts(" Enter a new grade.");
48               break; // optional; will exit switch anyway
49         } // end switch
50      } // end while
51
52      // output summary of results
53      puts("\nTotals for each letter grade are:");
54      printf("A: %d\n", aCount);
55      printf("B: %d\n", bCount);
56      printf("C: %d\n", cCount);
57      printf("D: %d\n", dCount);
58      printf("F: %d\n", fCount);
59   }
```

```
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. 4.5 | Counting letter grades with `switch`. (Part 2 of 2.)

Reading Character Input

In the program, the user enters students' letter grades. In the `while` header (line 17),

```
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 **`char`** variables. However, C can store characters in variables of any integer type because characters are usually represented as one-byte integers in the computer. Function `getchar` returns as an `int` the character that the user entered. 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 specifications `%c` and `%d` to print the character `'a'` and its integer value. The result is

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

Characters can be read with `scanf` by using the conversion specification `%c`. The integer 97 is the numerical representation of the character `'a'` in the computer. Many computers today use the Unicode® character set. Appendix B contains the **ASCII (American Standard Code for Information Interchange) character set** and its numeric values. ASCII is a subset of Unicode.

Assignments Have Values

Assignments as a whole actually have a value. The value of the assignment expression `grade = getchar()` is the character that's returned by `getchar` and assigned to the variable `grade`. 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 groups 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).

The value of the assignment `grade = getchar()` is compared with the value of `EOF` (a symbol whose acronym stands for "end of file"). 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 in Chapter 6 how symbolic constants are defined). If the value assigned to `grade` is equal to `EOF`, the program terminates.

We represent characters in this program as `int`s because `EOF` has an integer value (again, normally `-1`). 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.

Entering the EOF Indicator

The keystroke combinations for entering EOF (end of file) are system dependent. On Linux/UNIX/macOS systems, the `EOF` indicator is entered by typing on a line by itself

Ctrl + *d*

This notation means to simultaneously press both the *Ctrl* key and the *d* key. On other systems, such as Microsoft Windows, the `EOF` indicator can be entered by typing

Ctrl + *z*

You also need to press *Enter* on Windows.

The user enters grades at the keyboard. When the *Enter* key is pressed, the characters are read by function `getchar` one at a time. If the character entered is not equal to `EOF`, the `switch` statement (lines 20–49) executes.

switch Statement Details

Keyword `switch` is followed by the variable name `grade` in parentheses. This is called the **controlling expression**. The `switch` compares this expression's value with each of the **case labels**. Each `case` can have one or more actions, but braces are not required around multiple actions in a given `case`.

Assume the user has entered the letter `C` as a grade. When the `switch` compares `C` to each `case`, if a match occurs (`case 'C':`), the statements for that `case` execute. For the letter `C`, the `switch` increments `cCount` by `1` (line 31), then the `break` statement (line 32) exits the `switch` immediately, causing program control to continue with the first statement after the `switch` statement.

We use a `break` statement here because the `case`s in a `switch` statement would otherwise run together. Without `break` statements, each time a match occurs, all the remaining `case`s' statements will execute. (This feature—called fallthrough—is rarely useful, although it's perfect for compactly programming Exercise 4.38—the iterative song "The Twelve Days of Christmas"!) Forgetting a `break` statement when one is needed in a `switch` statement is a logic error. ERR

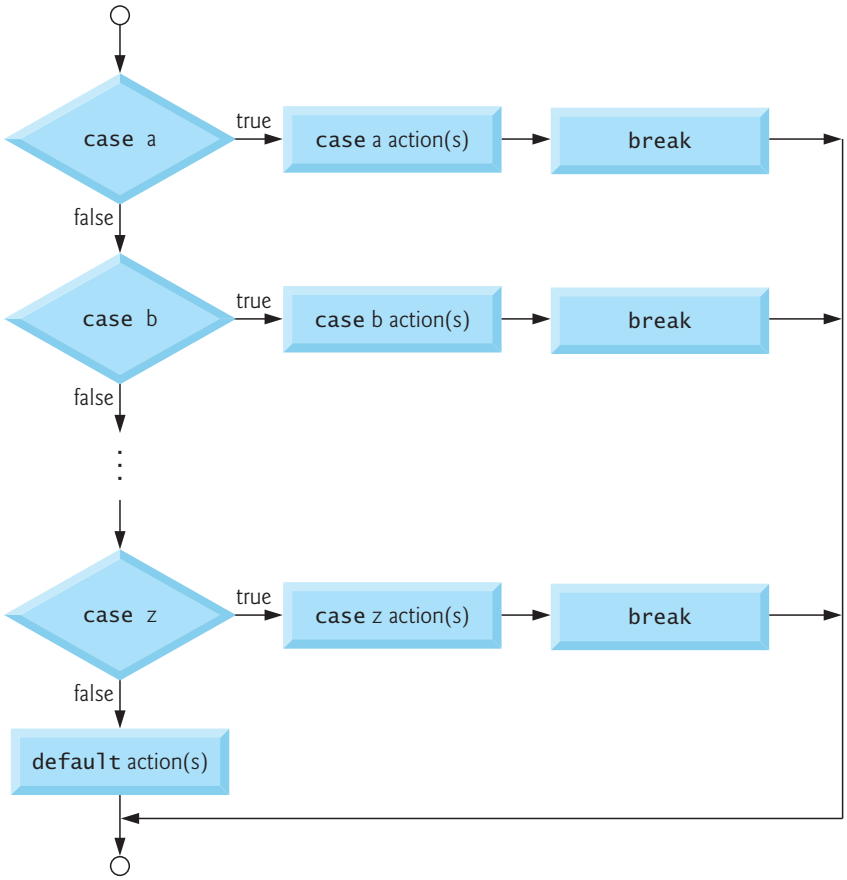
default Case

If no match occurs, the `default` case executes. In this program, it displays an error message. You should always include a `default` case; otherwise, values not explicitly tested in a `switch` will be ignored. The `default` case helps prevent this by focusing you on the need to process exceptional conditions. Sometimes no `default` processing is needed.

Although the `case` clauses and the `default` case clause in a `switch` statement can occur in any order, it's common to place the `default` clause last. When the `default` clause is last, the `break` statement isn't required. But many programmers include this `break` for clarity and symmetry with other cases.

switch Statement Flowchart

The following `switch` multiple-selection-statement flowchart makes it clear that each `case`'s `break` statement immediately exits the `switch` statement.



Ignoring Newline, Tab and Blank Characters in Input

In the `switch` statement of Fig. 4.5, the lines

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

cause the program to skip newline, tab and blank characters. Reading characters one at a time can cause problems. To have the program read the characters, you must send them to the computer by pressing *Enter*. This places the newline character in the input after the character we wish to process.

Often, this newline (and other whitespace characters) must be specifically ignored to make the program work correctly. The preceding cases in our `switch` statement prevent the error message in the `default` case from being printed each time a newline, tab or space is encountered in the input. Each input in this example causes two iterations of the loop—the first for a letter grade and the second for `'\n'`. Listing several `case` labels with no intervening statements means that the same actions occur for each of the cases.

Constant Integral Expressions

When using the `switch` statement, remember that each `case` can test only a **constant integral expression**. The expression can be any combination of character constants and integer constants that evaluates to a constant integer value. A character constant can be represented as the specific character in single quotes, such as `'A'`. 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 used character constants.

Notes on Integral Types

Portable languages like C must have flexible data-type sizes. Applications may need integers of various sizes. C provides several data types to represent integers. In addition to `int` and `char`, C provides types `short int` (which can be abbreviated as `short`) and `long int` (which can be abbreviated as `long`). There also are `unsigned` variations of all the integral types that represent non-negative integer values. In Section 5.14, we'll see that C also provides type `long long int` (which can be abbreviated as `long long`).

The C standard specifies the minimum range of values for each integer type. The actual range may be greater, depending on the implementation. For `short ints`, the minimum range is –32767 to +32767. For most integer calculations, `long ints` are sufficient. The minimum range of values for `long ints` is –2147483647 to +2147483647. An `int`'s range is greater than or equal to that of a `short int` and less than or equal to that of a `long int`. On many of today's platforms, `ints` and `long ints` represent the same range of values. The data type `signed char` can represent integers in the range –127 to +127 or any of the ASCII character set. See Section 5.2.4.2 of the C standard document for the complete list of `signed` and `unsigned` integer-type minimum ranges.

Self Check

1 *(Fill-In)* 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 _____.
Answer: multiple selection.

2 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) The value of an assignment is the value assigned to the variable on the left of the `=`.

b) The value of the assignment expression `grade = getchar()` is the character that's returned by `getchar` and assigned to the variable `grade`.

c) The following statement sets variables `a`, `b` and `c` to `0`:

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

d) All of the above statements are *true*.

Answer: c) is *false*. The correct statement is:

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

4.7 do...while Iteration Statement

The do...while iteration statement is similar to the while statement. The while statement tests its loop-continuation condition before executing the loop body. The do...while statement tests its loop-continuation condition after executing the loop body, so the loop body always executes at least once. When a do...while terminates, execution continues with the statement after the while clause. Figure 4.6 uses a do...while statement to display the numbers from 1 through 5. We chose to preincrement the control variable counter in the loop-continuation test (line 10).

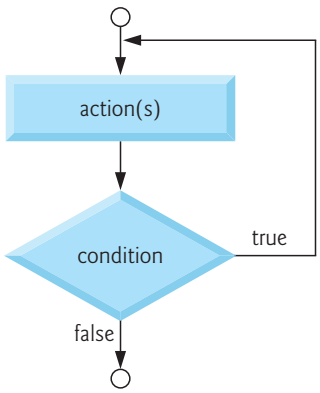
```
1  // fig04_06.c
2  // Using the do...while iteration statement.
3  #include <stdio.h>
4
5  int main(void) {
6     int counter = 1; // initialize counter
7
8     do {
9        printf("%d  ", counter);
10    } while (++counter <= 5);
11 }
```

```
1  2  3  4  5
```

Fig. 4.6 | Using the do...while iteration statement.

do...while Statement Flowchart

The following do...while statement flowchart makes it clear that the loop-continuation condition does not execute until after the loop's action is performed the first time:



Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) The while statement tests its loop-continuation condition before executing its body, so the loop body will always execute at least once.

b) The do...while statement tests its loop-continuation condition after executing its loop body.

c) When a do...while terminates, execution continues with the statement after the while clause.
d) All of the above statements are *true*.

Answer: a) is *false*. Actually, if the while statement's loop-continuation test fails upon entering the loop, the loop's body will not execute.

2 ***(True/False)*** Assuming counter is initialized to 1, the following loop displays the numbers 1 through 10:

```
do {
   printf("%d  ", counter);
} while (++counter < 10);
```

Answer: *False*. This loop displays the numbers 1 through 9. To display the numbers 1 through 10, change the < in the loop-continuation condition to <=.

4.8 break and continue Statements

The break and continue statements are used to alter the flow of control. Section 4.6 showed that a break encountered in a switch statement terminates the switch's execution. This section discusses how to use break in an iteration statement.

break Statement

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 after that while, for, do...while or switch. Common uses of break are to escape early from a loop or skip the remainder of a switch (as in Fig. 4.5). Figure 4.7 demonstrates the break statement (line 12) in a for iteration statement.

```
1  // fig04_07.c
2  // Using the break statement in a for statement.
3  #include <stdio.h>
4
5  int main(void) {
6     int x = 1; // declared here so it can be used after loop
7
8     // loop 10 times
9     for (; x <= 10; ++x) {
10       // if x is 5, terminate loop
11       if (x == 5) {
12          break; // break loop only if x is 5
13       }
14
15       printf("%d ", x);
16    }
17
18    printf("\nBroke out of loop at x == %d\n", x);
19 }
```

Fig. 4.7 | Using the break statement in a for statement. (Part 1 of 2.)

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

Fig. 4.7 | Using the `break` statement in a `for` statement. (Part 2 of 2.)

When the `if` statement detects that `x` has become `5`, `break` executes. This terminates the `for` statement, and the program continues with the `printf` after the `for`. The loop fully executes only four times. Recall that when you declare the control variable in a `for` loop's *initialization* expression, the variable no longer exists after the loop terminates. We declared and initialized `x` before the loop in this example, so that we could use its final value after the loop terminates. So, the initialization section of the `for`'s header (before the first semicolon) is empty.

`continue` Statement

The `continue` statement, when executed in a `while`, `for` or `do...while` statement, skips the remaining statements in that control statement's body 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 executes. In the `for` statement, the increment expression executes, then the loop-continuation test is evaluated. Figure 4.8 uses `continue` (line 10) in the `for` statement to skip the `printf` statement when `x` is `5` and begin the next iteration of the loop.

```
1  // fig04_08.c
2  // Using the continue statement in a for statement.
3  #include <stdio.h>
4
5  int main(void) {
6     // loop 10 times
7     for (int x = 1; x <= 10; ++x) {
8        // if x is 5, continue with next iteration of loop
9        if (x == 5) {
10          continue; // skip remaining code in loop body
11       }
12
13       printf("%d ", x);
14    }
15
16    puts("\nUsed continue to skip printing the value 5");
17 }
```

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

Fig. 4.8 | Using the `continue` statement in a `for` statement.

`break` and `continue` Notes

Some programmers feel `break` and `continue` violate the norms of structured programming, so they do not use them. The effects of these statements can be achieved by

structured programming techniques we'll soon discuss, but the `break` and `continue` statements perform faster. PERF

There's a tension between achieving quality software engineering and achieving the best-performing software—one is achieved at the expense of the other. For all but the most performance-intensive situations, apply the following guidelines: First, make your code simple and correct; then make it fast and small, but only if necessary. SE

Self Check

1 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) The `break` statement terminates a `switch` statement's execution.

b) The `break` statement, when executed in a `while`, `for` or `do...while` statement, causes an immediate exit from that statement.

c) Common uses of `break` are to escape early from a loop or to skip the remainder of an `if...else`.

d) All of the above statements are *true*.

Answer: c) is *false*. Actually, `break` skips the remainder of a `switch`, not an `if...else`.

2 *(Multiple Choice)* Which of the following statements a), b) or c) is *false*?

a) The `continue` statement, when executed in a `while`, `for` or `do...while` statement, skips the remaining statements in that control statement's body and performs the next iteration of the loop.

b) In `while` and `do...while` statements, the loop-continuation test is evaluated immediately after the `continue` statement executes.

c) In the `for` statement, after the `continue` statement executes, the loop-continuation test is evaluated, then the increment expression executes.

d) All of the above statements are *true*.

Answer: c) is *false*. Actually, in the `for` statement, after the `continue` statement executes, the increment expression executes, then the loop-continuation test is evaluated.

4.9 Logical Operators

So far we've used simple conditions, such as `counter <= 10`, `total > 1000`, and `grade != -1`. We've expressed these conditions in terms of the relational operators (`>`, `<`, `>=` and `<=`) and equality operators (`==` and `!=`). Each decision tested precisely one condition. To test multiple conditions in the process of making a decision, we had to perform these tests in separate statements or in nested `if` or `if...else` statements. 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**, which is also called **logical negation**). We'll consider examples of each of these operators.

Logical AND (&&) Operator

Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution. 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, for example, determine whether a person is a female. The condition `age >= 65` determines whether a person is a senior citizen. The two simple conditions are evaluated first because `==` and `>=` each have higher precedence than `&&`. The `if` statement then considers the combined condition `gender == 1 && age >= 65`, which is *true* if and only if both of the simple conditions are *true*. Finally, if this combined condition is *true*, then the preceding `if` statement increments `seniorFemales` by `1`. If either or both simple conditions are *false*, the program skips the `if`'s body and proceeds to the next statement in sequence.

The following table summarizes the **&& operator**:

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

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*.

Logical OR (||) Operator

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) {
    puts("Student grade is A");
};
```

This statement contains two simple conditions. The condition `semesterAverage >= 90` determines whether the student deserves an "A" because of a solid performance throughout the semester. The condition `finalExam >= 90` determines whether the student deserves an "A" because of an outstanding performance on the final exam. The `if` statement then considers the combined condition and awards the student an "A" if either or both of the simple conditions are *true*. The message "`Student grade is A`" prints unless both simple conditions are *false* (zero). The following 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

Short-Circuit Evaluation

The && operator has higher precedence than ||. Both operators associate from left-to-right. An expression containing && or || operators evaluates only until it's known whether the condition is *true* or *false*. Thus, the condition

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

stops evaluating if `gender` is not equal to `1`—the entire expression is guaranteed to be *false*. The condition continues evaluating if `gender` is equal to `1`—the entire expression could be *true* if `age` is greater than or equal to `65`. This performance feature for evaluating logical AND and logical OR expressions is called **short-circuit evaluation**.

PERF

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

PERF

Logical Negation (!) Operator

C provides the unary ! (logical negation) operator to enable you to "reverse" the meaning of a condition. The logical negation operator has a single condition as an operand. You use it when you're interested in choosing a path of execution if the operand condition 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 higher precedence than the equality operator. The following is a truth table for the logical negation operator:

expression	!expression
0	1
nonzero	0

In most cases, you can avoid using logical negation by expressing the condition differently. For example, the preceding statement may also be written as:

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

Summary of Operator Precedence and Grouping

The following table shows the precedence and grouping of the operators introduced to this point. The operators are shown from top to bottom in decreasing order of precedence.

Operators	Grouping	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

The _Bool Data Type

The C standard includes a boolean type—represented by the keyword `_Bool`—which can hold only the values 0 or 1. Recall that the value 0 in a condition is *false*, while any nonzero value is *true*. Assigning any nonzero value to a `_Bool` sets it to 1. The standard also includes the `<stdbool.h>` header, which defines `bool` as a shorthand for the type `_Bool`, and `true` and `false` as named representations of `1` and `0`, respectively. During preprocessing, the identifiers `bool`, `true` and `false` are replaced with `_Bool`, `1` and `0`, respectively.

Self Check

1 *(Multiple Choice)* When the following `if` statement executes, which pair of variable values would cause `seniorFemales` to be incremented?

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

a) `gender` is `2` and `age` is `60`.
b) `gender` is `2` and `age` is `73`.
c) `gender` is `1` and `age` is `19`.
d) `gender` is `1` and `age` is `65`.

Answer: d.

2 *(Multiple Choice)* When the following `if` statement executes, which pair of variable values would not cause `"Student grade is A"` to print?

```
if (semesterAverage >= 90 || finalExam >= 90) {
    puts("Student grade is A");
}
```

a) `semesterAverage` is `75` and `finalExam` is `80`.
b) `semesterAverage` is `85` and `finalExam` is `91`.
c) `semesterAverage` is `93` and `finalExam` is `67`.
d) `semesterAverage` is `94` and `finalExam` is `90`.

Answer: a.

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

There's one type of error that C programmers, no matter how experienced, tend to make so frequently that it's 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.

Two aspects of C cause these problems. One is that any expression 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 an assignment has a value—whatever value is assigned to the variable on the left of the = operator.

For example, suppose we intend to write

```
if (payCode == 4) {
   printf("%s", "You get a bonus!");
}
```

but we accidentally write

```
if (payCode = 4) {
   printf("%s", "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. The value in the condition after the assignment is 4. Because any nonzero value is *true*, the condition in this `if` statement is always *true*. Not only is `payCode` inadvertently set to 4, but the person always receives a bonus regardless of what the actual `payCode` is! Accidentally using operator == for assignment and accidentally using operator = for equality are both logic errors.

lvalues and *rvalues*

You'll probably be inclined to 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`, if you accidentally

replace the == operator with =, you'll be 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. This will prevent the potential devastation of a run-time logic error.

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 only on the right side of an assignment operator. An *lvalue* can also be used as an *rvalue*, but not vice versa.

Confusing == and = in Standalone Statements

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

```
x = 1;
```

but instead write

```
x == 1;
```

Here, too, this is not a syntax error. The compiler evaluates the conditional expression. 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's no assignment operator, so the value is simply lost. 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.

Self Check

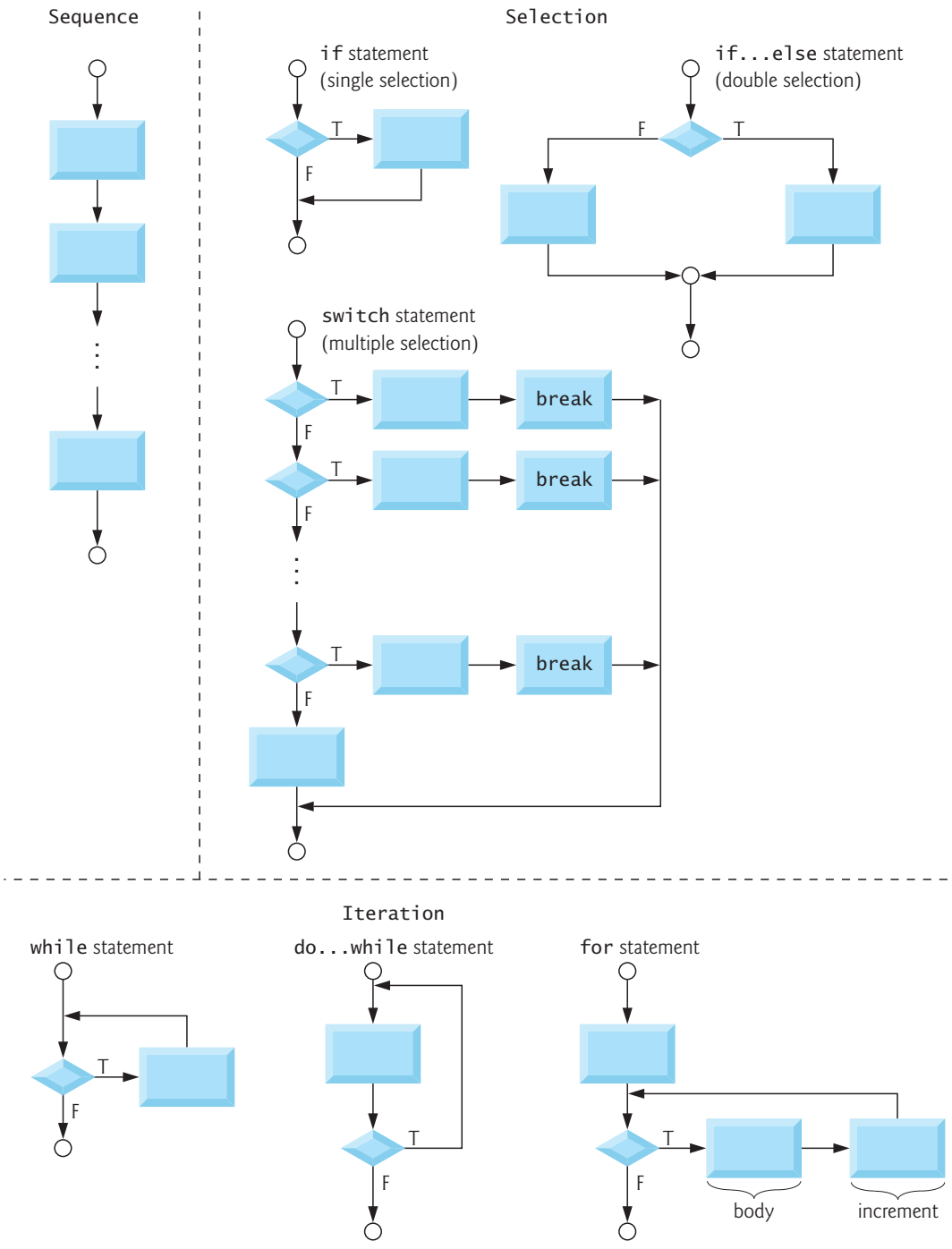
1 ***(True/False)*** Accidentally swapping the operators == (equality) and = (assignment) is damaging because these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results.
Answer: *True*.

2 ***(True/False)*** An *rvalue* can also be used as an *lvalue*, but not vice versa.
Answer: *False*. Actually, an *lvalue* can also be used as an *rvalue*, but not vice versa.

4.11 Structured-Programming Summary

Just as architects design buildings by employing the collective wisdom of their profession, programmers should design programs by employing the collective wisdom of their profession. Our field is younger than architecture, and our collective wisdom is considerably sparser. We've learned a great deal in a mere nine decades. Perhaps most important, we've learned that structured programs are easier (than unstructured programs) to understand, test, debug, modify, and even prove correct in a mathematical sense.

Chapters 3 and 4 discussed C's control statements. Now, we summarize these capabilities and introduce a simple set of rules for forming structured programs. The following diagram summarizes the control statements' flowcharts:



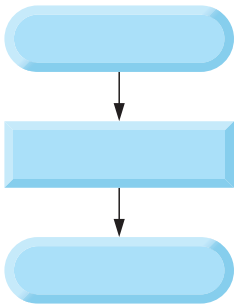
In the diagram, small circles indicate each statement's single entry point and single exit point. Connecting individual flowchart symbols arbitrarily can lead to unstructured programs. Therefore, the programming profession has chosen to combine flowchart symbols to form a limited set of control statements and to build only properly structured programs by combining control statements in two straightforward ways. For simplicity, only single-entry/single-exit control statements are used, and you may combine them only by stacking control statements in sequence or by nesting them.

Rules for Forming Structured Programs

The following table summarizes the rules for forming structured programs. We assume that the rectangle flowchart symbol indicates any action, including input/output:

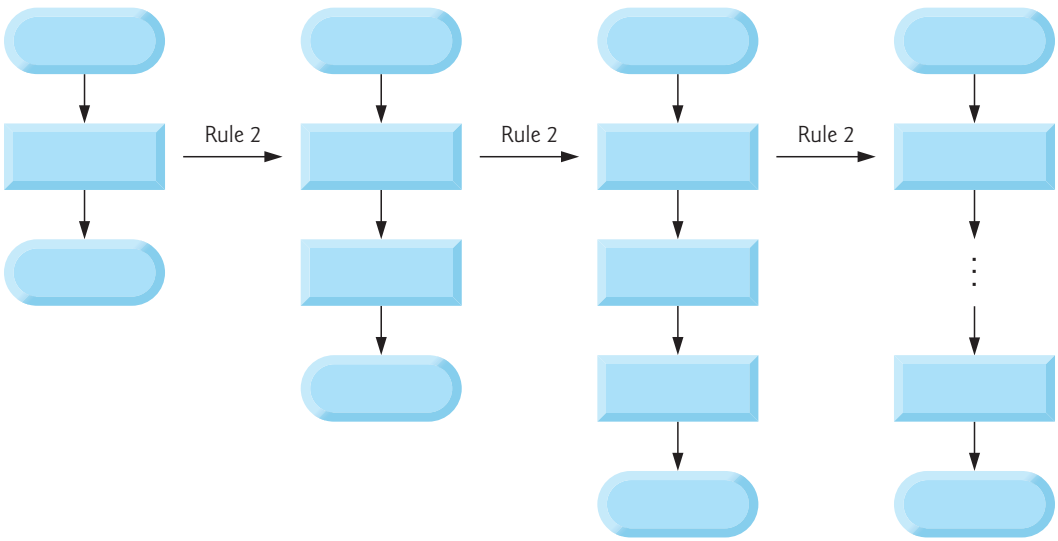
Rules for forming structured programs

1. Begin with the "simplest flowchart" shown in the next diagram.
2. "Stacking" rule—Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3. "Nesting" rule—Any rectangle (action) can be replaced 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.



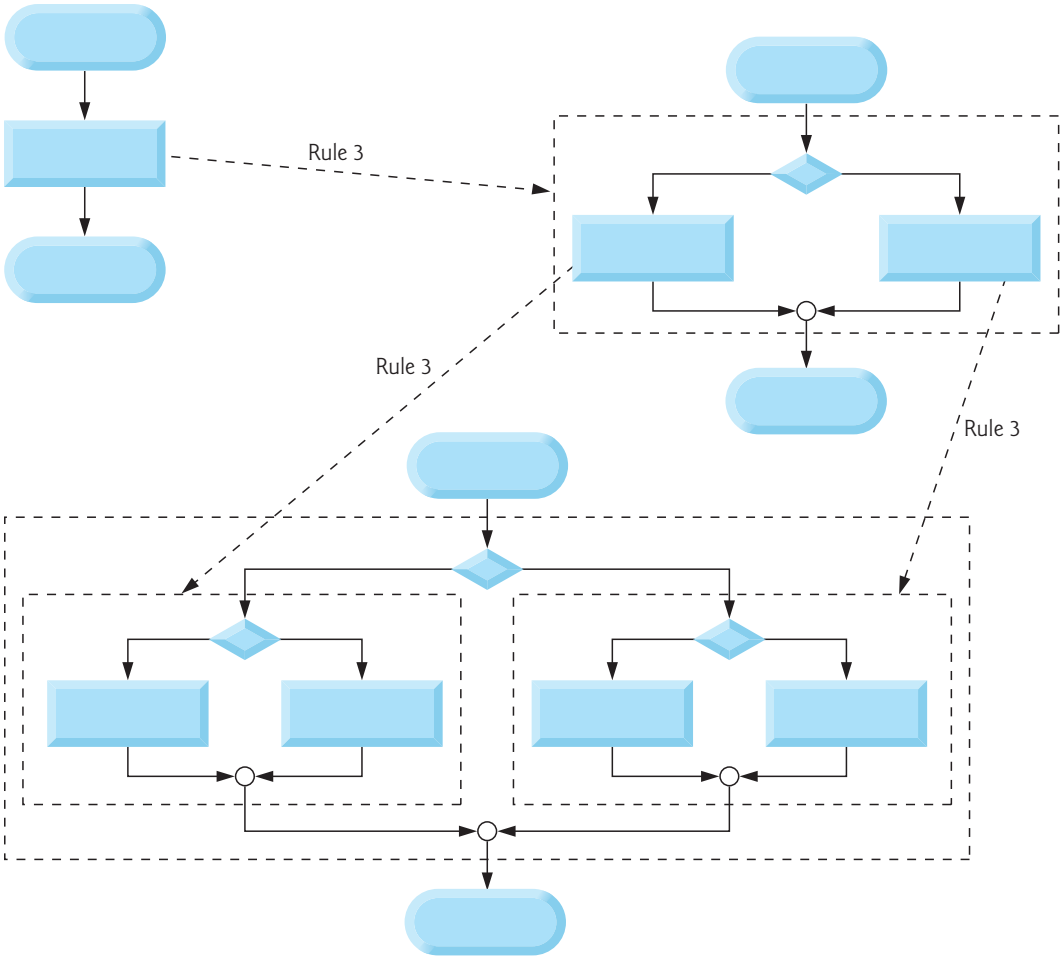
Rules for Forming Structured Programs—Stacking Rule

Applying the rules for forming structured programs always results in a structured flowchart with a neat, building-block appearance. Repeatedly applying Rule 2 to the simplest flowchart results in a structured flowchart containing many rectangles in sequence, as in the following diagram. Rule 2 generates a stack of control statements, so we call Rule 2 the **stacking rule**.



Rules for Forming Structured Programs—Nesting Rule

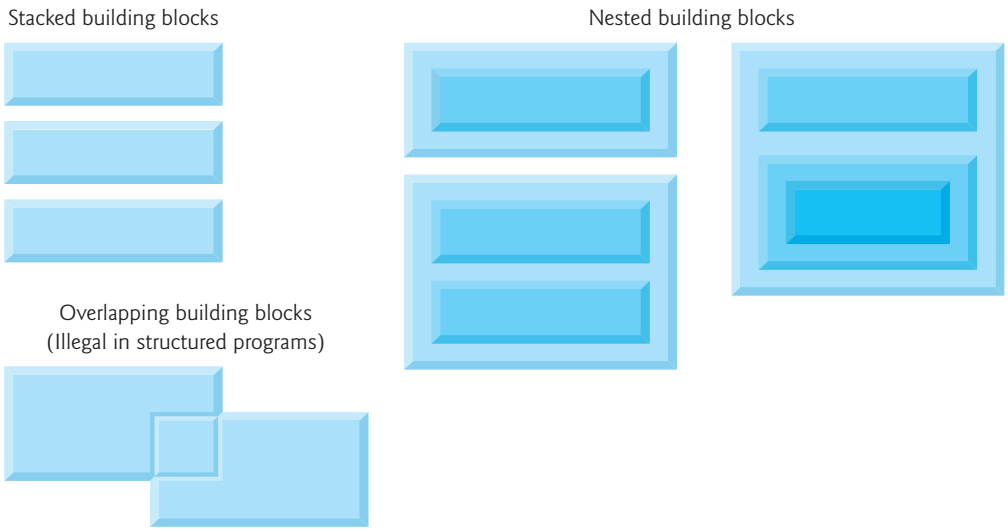
Rule 3 is called the **nesting rule**. Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control statements. For example, in the following diagram, the rectangle in the simplest flowchart is replaced with a double-selection (`if...else`) statement. Then Rule 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 we replaced in the original flowchart.



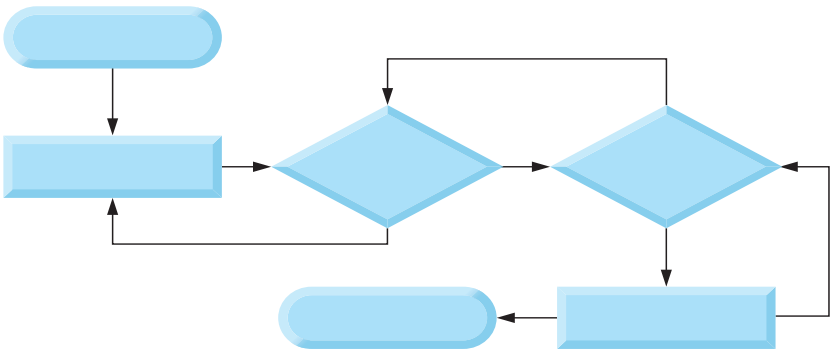
Rule 4 generates larger, more involved, and more deeply nested structures. The flowcharts that emerge from applying the rules for forming structured programs constitute the set of all possible structured flowcharts and hence the set of all possible structured programs.

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. The following diagram 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).



If the rules for forming structured programs are followed, an unstructured flowchart, such as the one in the following diagram, *cannot* be created:



If you're uncertain whether a particular flowchart is structured, apply the rules for forming structured programs in reverse to try to reduce the flowchart to the simplest flowchart. If you succeed, the original flowchart is structured; otherwise, it's not.

Three Forms of Control

Structured programming promotes simplicity. Böhm and Jacopini showed that only three forms of control are needed:

- Sequence.
- Selection.
- Iteration.

Sequence is straightforward. Selection is implemented in one of three ways:

- `if` statement (single selection).
- `if...else` statement (double selection).
- `switch` statement (multiple selection).

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.

Iteration is implemented in one of three ways:

- `while` statement.
- `do...while` statement.
- `for` statement.

It's also straightforward to prove that the `while` statement is sufficient to provide any form of iteration. Everything that can be done with the `do...while` statement and the `for` statement can be done with the `while` statement.

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 (iteration).

And these control statements can be combined in only *two* ways—*stacking* and *nesting*. Indeed, structured programming promotes simplicity.

In Chapters 3 and 4, we've discussed how to compose programs from control statements containing only 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, can be composed of control statements. We'll also discuss how using functions promotes software reusability.

4.12 Secure C Programming

SEC

Checking Function `scanf`'s Return Value

Figure 4.4 used the math library function `pow`, which calculates the value of its first argument raised to the power of its second argument and *returns* the result as a `double` value. The calculation's result was then used in the statement that called `pow`.

Many functions return values indicating whether they executed successfully. For example, function `scanf` returns an `int` indicating whether the input operation was successful. If an input failure occurs before `scanf` can input a value, `scanf` returns the value `EOF` (defined in `<stdio.h>`); otherwise, it returns the number of items that were read into variables. If this value does *not* match the number you intended to input, then `scanf` was unable to complete the input operation.

Consider the following statement that expects to read one `int` value into `grade`:

```
scanf("%d", &grade); // read grade from user
```

If the user enters an integer, `scanf` returns `1`, indicating that one value was indeed read. If the user enters a string, such as `"hello"`, `scanf` returns `0`, indicating that it was unable to convert the input to an integer. In this case, the variable `grade` does *not* receive a value.

Function `scanf` can input multiple values, as in

```
scanf("%d%d", &number1, &number2); // read two integers
```

If the input into both variables is successful, `scanf` will return `2`. If the user enters a string for the first value, `scanf` will return `0`, and neither `number1` nor `number2` will receive a value. If the user enters an integer followed by a string, `scanf` will return `1`, and only `number1` will receive a value.

ERR ⊗ To make your input processing more robust, check `scanf`'s return value to ensure that the number of inputs read matches the number of inputs expected. Otherwise, your program will use the values of the variables as if `scanf` had completed successfully. This could lead to logic errors, program crashes or even attacks.

Range Checking

Even if a `scanf` operates successfully, the values read might still be invalid. For example, grades are typically integers in the range 0–100. In a program that inputs grades, you should **validate** each grade to ensure that it's in the range 0–100 by using **range checking**. You can then ask the user to reenter any value that's out of range. If a program requires inputs from a specific set of values (such as non-sequential product codes), you can ensure that each input matches a value in the set.[2]

✓ Self Check

1 *(Fill-In)* If an input failure occurs, `scanf` returns the value _____; otherwise, it returns the number of items that were read.

Answer: `EOF`.

2 *(Multiple Choice)* Given the following code:

```
scanf("%d%d", &grade1, &grade2); // read two integers
```

which of the following statements a), b) or c) is *false*?

a) If the input is successful, `scanf` will return `0`, indicating that integer values for both variables were input.
b) If the user enters a string for the first value, `scanf` will return `0`, and neither `grade1` nor `grade2` will receive a value.
c) If the user enters an integer followed by a string, `scanf` will return `1`, and only `grade1` will receive a value.
d) All of the above statements are *true*.

Answer: a) is *false*. Actually, if both values are input correctly, `scanf` will return `2`.

Summary

Section 4.2 Iteration Essentials

- Most programs involve iteration (or looping). A loop is a group of instructions the computer repeatedly executes while some **loop-continuation condition** (p. 186) remains *true*.

2. For more information, see Chapter 5, "Integer Security," of Robert Seacord's book *Secure Coding in C and C++, 2/e*.

- Counter-controlled iteration uses a **control variable** (p. 186) to count the number of iterations. When the correct number of iterations completes, the loop terminates, and the program resumes execution with the statement after the iteration statement.
- In sentinel-controlled iteration, a **sentinel value** is entered after all regular data items to indicate "end of data." Sentinels must be distinct from regular data items.

Section 4.3 Counter-Controlled Iteration

- Counter-controlled iteration requires the control variable's **name** (p. 187), its **initial value** (p. 187), the **increment** (or **decrement**) by which it's modified each time through the loop, and the condition that tests for the control variable's **final value** (p. 187).
- The control variable **increments** (or **decrements**) each time the group of instructions is performed (p. 187).

Section 4.4 `for` Iteration Statement

- The **`for` iteration statement** handles all the details of counter-controlled iteration.
- When a `for` statement begins executing, its control variable is initialized. Then, the loop-continuation condition is checked. If the condition is true, the loop's body executes. The control variable is then incremented, and the loop-continuation condition is tested. This continues until the loop-continuation condition fails.
- The general format of the `for` statement is

```
for (initialization; condition; increment) {
   statements
}
```

  where the *initialization* expression initializes (and possibly defines) the control variable, the *condition* expression is the loop-continuation condition, and the *increment* expression increments the control variable.
- The three expressions in the `for` header are optional. If the *condition* is omitted, C assumes the condition is true, creating an infinite loop. One might omit the *initialization* expression if the control variable is initialized before the loop. One might omit the *increment* expression if it's calculated by statements in the `for` statement's body or if no increment is needed.
- The two semicolons in the `for` header are required.
- The "increment" may be negative to create a loop that counts downward.
- If the loop-continuation condition is initially false, the body portion of the loop isn't performed. Instead, execution proceeds with the statement following the `for` statement.

Section 4.5 Examples Using the `for` Statement

- **Function `pow`** (p. 194) performs exponentiation. The call `pow(x, y)` calculates the value of `x` raised to the `y`th power. The function receives two `double` arguments and returns a `double`.
- Include the **header `<math.h>`** (p. 194) whenever you need a math function such as `pow`.
- The conversion specification `%21.2f` denotes that a floating-point value will be displayed **right-aligned** in a field of 21 characters with two digits to the right of the decimal point.
- To **left-align** a value in a field, place a `-` (minus sign) between the `%` and the field width.

Section 4.6 `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` statement** to handle this.

- Characters are normally stored in variables of type **char** (p. 198). Characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer. Thus, we can treat a character as either an integer or a character, depending on its use.
- The `switch` statement consists of a series of **`case` labels** (p. 199), an optional **`default` case** and statements to execute for each case.
- The **`getchar` function** (header `<stdio.h>`) reads and returns as an `int` one character from the keyboard.
- Many computers today use the Unicode character set. ASCII is a subset of Unicode.
- Characters can be read with `scanf` by using the conversion specification **`%c`**.
- **Assignment expressions** as a whole actually **have a value**. This value is assigned to the variable on the left side of the `=`.
- **`EOF`** is often used as a sentinel value. `EOF` is a symbolic integer constant defined in `<stdio.h>`.
- On macOS/Linux systems, the `EOF` indicator is entered by typing *Ctrl* + *d*. On Windows, the `EOF` indicator can be entered by typing *Ctrl* + *z*.
- Keyword `switch` is followed by the **controlling expression** (p. 199) in parentheses. The value of this expression is compared with each of the `case` labels. If a match occurs, the statements for that `case` execute. If no match occurs, the `default` case executes.
- The **`break` statement** causes program control to continue with the statement after the `switch`. The `break` statement prevents the `case`s in a `switch` statement from running together.
- Each `case` can have one or more actions. Braces are not required around multiple actions in a `case` of a `switch`.
- **Listing several `case` labels together** executes the same set of actions for any of these `case`s.
- Each case in a `switch` statement can test only a **constant integral expression** (p. 201)—i.e., any combination of character constants and integer constants that evaluates to a constant integer value. A character constant can be represented as the specific character in single quotes, such as `'A'`. Characters must be enclosed within single quotes to be recognized as character constants. Integer constants are simply integer values.
- In addition to integer types `int` and `char`, C provides types **`short int`** (which can be abbreviated as **`short`**) and **`long int`** (which can be abbreviated as **`long`**), as well as **`unsigned`** versions of all the integral types. The C standard specifies the minimum value range for each type. The actual range may be greater, depending on the implementation. For `short` `int`s, the minimum range is –32767 to +32767. The minimum range of values for `long` `int`s is –2147483647 to +2147483647. The range of values for an `int` is greater than or equal to that of a `short` `int` and less than or equal to that of a `long` `int`. On many of today's platforms, `int`s and `long` `int`s represent the same range of values. The data type `signed` `char` can be used to represent integers in the range –127 to +127 or any of the characters in the ASCII character set.

Section 4.7 do…while Iteration Statement

- The **`do…while` statement** tests the loop-continuation condition *after* the loop body is performed. Therefore, the loop body executes at least once. When a `do…while` terminates, execution continues with the statement after the `while` clause.

Section 4.8 break and continue Statements

- The **`break` statement**, when executed in a `while`, `for`, `do…while` or `switch` statement, immediately exits that statement. Program execution continues with the next statement.
- The **`continue` statement**, when executed in a `while`, `for` or `do…while` statement, skips the remaining statements in the body and performs the next loop iteration. The `while` and

`do...while` evaluate the loop-continuation test immediately. A `for` statement executes its increment expression, then tests the loop-continuation condition.

Section 4.9 Logical Operators

- **Logical operators** `&&` (logical AND), `||` (logical OR) and `!` (logical NOT, or logical negation) may be used to form complex conditions by combining simple conditions.
- The **`&&`** (**logical AND**; p. 205) operator evaluates to *true* if and only if both of its operands are *true*.
- 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***.
- The **`||`** (**logical OR**; p. 205) operator evaluates to *true* if either or both its operands are true.
- The `&&` operator has higher precedence than `||`. Both operators group from left-to-right.
- Operators `&&` or `||` use short-circuit evaluation, terminating as soon as the condition is known to be false or true.
- C provides the **`!`** (**logical negation**; p. 205) operator to enable you to "reverse" the meaning of a condition. Unlike the binary operators `&&` and `||`, which combine two conditions, the unary logical negation operator has only a single condition as an operand.
- 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*.

Section 4.10 Confusing Equality (==) and Assignment (=) Operators

- Programmers often accidentally swap the operators `==` and `=`. Statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through runtime logic errors.
- In a condition like `7 == x`, if you accidentally replace `==` with `=`, the compiler will report a syntax error. Only a variable name can be placed on the left-hand side of an assignment.
- Variable names are said to be ***lvalues*** (for "**left values**"; p. 210) because they can be used on the left side of an assignment operator.
- Constants are said to be ***rvalues*** (for "**right values**"; p. 210) because they can be used only on the right side of an assignment operator. *lvalues* can also be used as *rvalues*, but not vice versa.

Self-Review Exercises

4.1 Fill-In the blanks in each of the following statements.

a) In counter-controlled iteration, a(n) _____ is used to count the number of times a group of instructions should be repeated.

b) The _____ statement, when executed in an iteration statement, causes the next iteration of the loop to be performed immediately.

c) The _____ statement, when executed in an iteration statement or a `switch`, causes an immediate exit from the statement.

d) The _____ is used to test a particular variable or expression for each of the constant integral values it may assume.

4.2 State whether the following are *true* or *false*. If the answer is *false*, explain why.

a) The `default` case is required in the `switch` selection statement.

b) The `break` statement is required in a `switch` statement's `default` case.

c) The expression `(x > y && a < b)` is *true* if either `x > y` is *true* or `a < b` is *true*.

d) An expression containing the `||` operator is *true* if either or both of its operands are *true*.

4.3 Write a statement or a set of statements to accomplish each of the following tasks:

a) Sum the odd integers between 1 and 99 using a `for` statement. Use the integer variables `sum` and `count`.

b) Print the value `333.546372` in a field width of `15` characters with precisions of `1`, `2`, `3`, `4` and `5`. Left-align the output. What are the five values that print?

c) Calculate the value of `2.5` raised to the power of `3` using the `pow` function. Print the result with a precision of `2` in a field width of `10` positions. What is the value that prints?

d) Print the integers from 1 to 20 using a `while` loop and the counter variable `x`. Print only five integers per line. [*Hint*: Use the calculation `x % 5`. When this is 0, print a newline character, otherwise print a tab character.]

e) Repeat Exercise 4.3(d) using a `for` statement.

4.4 Find the error in each of the following code segments and explain how to correct it:

a)
```
x = 1;
while (x <= 10);
   ++x;
}
```

b)
```
for (double y = .1; y != 1.0; y += .1) {
   printf("%f\n", y);
}
```

c)
```
switch (n) {
   case 1:
      puts("The number is 1");
   case 2:
      puts("The number is 2");
      break;
   default:
      puts("The number is not 1 or 2");
      break;
}
```

d) The following code should print the values 1 to 10.
```
n = 1;
while (n < 10) {
   printf("%d ", n++);
}
```

Answers to Self-Review Exercises

4.1 a) control variable or counter. b) `continue`. c) `break`. d) `switch` selection statement.

4.2 See the answers below:

a) *False*. The `default` case is optional. If no default action is needed, then there's no need for a `default` case.

b) *False*. The `break` statement is used to exit the `switch` statement. The `break` statement is not required in *any* `case`.

c) *False*. Both of the relational expressions must be *true* for the entire expression to be *true* when using the `&&` operator.

d) *True*.

4.3 See the answers below:

a)
```
int sum = 0;
for (int count = 1; count <= 99; count += 2) {
   sum += count;
}
```

b)
```
printf("%-15.1f\n", 333.546372); // prints 333.5
printf("%-15.2f\n", 333.546372); // prints 333.55
printf("%-15.3f\n", 333.546372); // prints 333.546
printf("%-15.4f\n", 333.546372); // prints 333.5464
printf("%-15.5f\n", 333.546372); // prints 333.54637
```

c)
```
printf("%10.2f\n", pow(2.5, 3)); // prints 15.63
```

d)
```
int x = 1;
while (x <= 20) {
   printf("%d", x);
   if (x % 5 == 0) {
      puts("");
   }
   else {
      printf("%s", "\t");
   }
   ++x;
}
```

or

```
int x = 1;
while (x <= 20) {
   if (x % 5 == 0) {
      printf("%d\n", x++);
   }
   else {
      printf("%d\t", x++);
   }
}
```

or

```
int x = 0;
while (++x <= 20) {
   if (x % 5 == 0) {
      printf("%d\n", x);
   }
   else {
      printf("%d\t", x);
   }
}
```

e)
```
for (int x = 1; x <= 20; ++x) {
   printf("%d", x);
   if (x % 5 == 0) {
      puts("");
   }
   else {
      printf("%s", "\t");
   }
}
```

or

```
for (int x = 1; x <= 20; ++x) {
   if (x % 5 == 0) {
      printf("%d\n", x);
   }
   else {
      printf("%d\t", x);
   }
}
```

4.4 a) Error: The semicolon after the `while` header causes an infinite loop.
Correction: Replace the semicolon with a `{` or remove both the `;` and the `}`.

b) Error: Using a floating-point number to control a `for` iteration statement.
Correction: Use an integer, and perform the proper calculation to get the values you desire.

```
for (int y = 1; y != 10; ++y) {
   printf("%f\n", (float) y / 10);
}
```

c) Error: Missing `break` statement in the statements for the first `case`.
Correction: Add a `break` statement at the end of the statements for the first `case`. This is not necessarily an error if you want the statement of `case 2:` to execute every time the `case 1:` statement executes.

d) Error: Improper relational operator used in the `while` iteration-continuation condition.
Correction: Use <= rather than <.

Exercises

4.5 Find the error in each of the following: (*Note*: There may be more than one error.)

a)
```
for (a = 25, a <= 1, a--); {
   printf("%d\n", a);
}
```

b) The following code should print whether a given integer is odd or even:
```
switch (value) {
   case (value % 2 == 0):
     puts("Even integer");
   case (value % 2 != 0):
     puts("Odd integer");
}
```

c) The following code should calculate incremented salary after 10 years:
```
for (int year = 1; year <= 10; ++year) {
double salary += salary * 0.05;
}
   printf("%4u%21.2f\n", year, salary);
```

d)
```
for (double y = 7.11; y != 7.20; y += .01)
   printf("%7.2f\n", y);
```

e) The following code should output all multiples of 3 from 1 to 100:
```
for (int x=3; x <=100, x%3==0; x++) {
   printf("%d\n", x);
}
```

f)
```
x = 1;
While (x <= 10) {
   printf("%d\n", x);
}
```

g) The following code should sum the squares of all numbers from 1 to 50 (assume sum is initialized to 0):
```
for (x = 1; x == 50; ++x); {
   sum =+ x * x;
}
```

4.6 State which values of the control variable x are printed by each of the following `for` statements:

a)
```
for (x = 20; x >= 3; x -= 3) {
   printf("%u\n", x);
}
```

```
b) for (x = 7; x <= 27; x += 5) {
      printf("%u\n", x);
   }
c) for (x = 2; x <= 20; x += 4) {
      printf("%u\n", x);
   }
d) for (x = 30; x >= 15; x -= 6) {
      printf("%u\n", x);
   }
e) for (x = 22; x >= 2; x -= 5) {
      printf("%d\n", x);
   }
```

4.7 Write `for` statements that print the following sequences of values:
a) 1, 3, 5, 7, 9, 11, 13
b) 2, 5, 8, 11, 14, 17
c) 30, 20, 10, 0, –10, –20, –30
d) 15, 23, 31, 39, 47, 55

4.8 What does the following program do?

```
1  #include <stdio.h>
2  int main()
3  {
4     int x, i, j;
5     // prompt user for input
6     printf("%s", "Enter an integer in the range 1-20:");
7     scanf("%d", &x); // read values for x
8     for (i = 1; i <= x; i++) { // count from 1 to x
9        for (j = 1; j <= x; j++) { // count from 1 to x
10          if (j==i)
11              printf("%c",'@'); // output @
12          else
13              printf(" ");
14       } // end inner for
15       printf("\n");
16    } // end outer for
17 } // end of main
```

4.9 ***(Sum and Average of Integers)*** Write a program to sum a sequence of integers, and calculate their average. Assume that the first integer read with `scanf` specifies the number of values to be entered. Your program should read only one value each time `scanf` is executed. A typical input sequence might be

```
7 678 234 315 489 536 456 367
```

where the 7 indicates that the subsequent 7 values are to be summed.

4.10 ***(Converting Celsius to Fahrenheit)*** Write a program that converts temperatures ranging between 30°C and 50°C to the Fahrenheit scale. The program should print a table displaying temperatures in the two scales side by side. [*Hint*: $°F = \frac{9}{5}C + 32$]

4.11 ***(Calculating the Sum of Multiples)*** Write a program to calculate and print the sum of all multiples of 7 from 1 to 100.

4.12 ***(Prime Numbers)*** Write a program to calculate and print a list of all prime numbers from 1 to 100.

4.13 ***(Natural Numbers Arithmetic)*** Write a program that prints the sum, the sum of the squares, and the sum of the cubes of all natural numbers from 1 till any number entered by the user.

4.14 ***(Factorials)*** The *factorial* function is used frequently in probability problems. The factorial of a positive integer *n* (written *n*! and pronounced "*n* factorial") is equal to the product of the positive integers from 1 to *n*. Write a program that evaluates the factorials of the integers from 1 to 5. Print the results in tabular format. What difficulty might prevent you from calculating the factorial of 20?

4.15 ***(Modified Compound-Interest Program)*** Modify the compound-interest program of Section 4.5 to repeat its steps for interest rates of 5%, 6%, 7%, 8%, 9%, and 10%. Use a `for` loop to vary the interest rate.

4.16 ***(Triangle-Printing Program)*** Write a program that prints the following patterns separately, one below the other. Use `for` loops to generate the patterns. All asterisks (`*`) should be printed by a single `printf` statement of the form `printf("%s", "*");` (this causes the asterisks to print side-by-side). [*Hint*: The last two patterns require that each line begin with an appropriate number of blanks.]

```
(A)          (B)          (C)          (D)
*            **********    **********           *
**           *********      *********          **
***          ********        ********         ***
****         *******          *******        ****
*****        ******            ******       *****
******       *****              *****      ******
*******      ****                ****     *******
********     ***                  ***    ********
*********    **                    **   *********
**********   *                      *  **********
```

4.17 ***(Calculating Credit Limits)*** Collecting money becomes increasingly difficult during periods of recession, so companies may tighten their credit limits to prevent their accounts receivable (money owed to them) from becoming too large. In response to a prolonged recession, one company has cut its customers' credit limits in half. Thus, if a particular customer had a credit limit of $2000, it's now $1000. If a customer had a credit limit of $5000, it's now $2500. Write a program that analyzes the credit status of three customers of this company. For each customer you're given:

a) The customer's account number.
b) The customer's credit limit before the recession.
c) The customer's current balance (i.e., the amount the customer owes).

Your program should calculate and print the new credit limit for each customer and determine (and print) which customers have balances that exceed their new credit limits.

4.18 ***(Bar-Chart Printing Program)*** One interesting application of computers is drawing graphs and bar charts. Write a program that reads five numbers (each between 1 and 30). For each number read, your program should print a line containing that number of adjacent asterisks. For example, if your program reads the number seven, it should print *******.

4.19 ***(Calculating Sales)*** An online retailer sells five different products whose retail prices are shown in the following table:

Product number	Retail price
1	$ 2.98
2	$ 4.50
3	$ 9.98
4	$ 4.49
5	$ 6.87

Write a program that reads a series of pairs of numbers as follows:

a) Product number.
b) Quantity sold for one day.

Your program should use a `switch` statement to help determine the retail price for each product. Your program should calculate and display the total retail value of all products sold last week.

4.20 ***(Truth Tables)*** Complete the following truth tables by filling in each blank with 0 or 1.

Condition1	Condition2	Condition1 && Condition2
0	0	0
0	nonzero	0
nonzero	0	_____
nonzero	nonzero	_____

Condition1	Condition2	Condition1 || Condition2
0	0	0
0	nonzero	1
nonzero	0	_____
nonzero	nonzero	_____

Condition	! Condition
0	1
nonzero	———

4.21 *(ASCII Values)* Write a program to convert and print the characters for the ASCII values 0 to 127. The program should print 10 characters per line.

4.22 *(Average Grade)* Modify the program of Fig. 4.5 so that it calculates the average grade for the class.

4.23 *(Calculating the Compound Interest with Integers)* Modify the program of Fig. 4.4 so that it uses only integers to calculate the compound interest. [*Hint*: Treat all monetary amounts as integral numbers of pennies. Then "break" the result into its dollar portion and cents portion by using the division and remainder operations, respectively. Insert a period.]

4.24 Assume `i = 5`, `j = 7`, `k = 4`, and `m = -2`. What does each of the following statements print?

a) `printf("%d", i == 5);`
b) `printf("%d", j != 3);`
c) `printf("%d", i >= 5 && j < 4);`
d) `printf("%d", !m && k > m);`
e) `printf("%d", !k || m);`
f) `printf("%d", k - m < j || 5 - j >= k);`
g) `printf("%d", j + m <= i && !0);`
h) `printf("%d", !(j - m));`
i) `printf("%d", !(k > m));`
j) `printf("%d", !(j > k));`

4.25 *(Table of Decimal, Binary, Octal and Hexadecimal Equivalents)* Write a program that prints a table of the binary, octal and hexadecimal equivalents of the decimal numbers 1—256. If you're not familiar with these number systems, read online Appendix E before you attempt this exercise. [*Note*: You can display an integer as an octal or hexadecimal value with the conversion specifications `%o` and `%X`, respectively.]

4.26 *(Calculating the Value of π)* Calculate the value of π from the infinite series

$$\pi = 4 - \frac{4}{3} + \frac{4}{5} - \frac{4}{7} + \frac{4}{9} - \frac{4}{11} + \cdots$$

Print a table that shows the value of π approximated by one term of this series, by two terms, by three terms, and so on. How many terms of this series do you have to use before you first get 3.14? 3.141? 3.1415? 3.14159?

4.27 *(Pythagorean Triples)* A right triangle can have sides that are all integers. The set of three integer values for a right triangle's sides is a Pythagorean triple. These three sides must satisfy the relationship that the sum of the sides' squares is equal to the

hypotenuse's square. Find all Pythagorean triples for side1, side2, and the hypotenuse, all no larger than 500. Use a triple-nested `for` loop that tries all possibilities. This is an example of "brute-force" computing. It's not aesthetically pleasing to many people. But there are many reasons why this technique is important. First, with computing power increasing at such a phenomenal pace, solutions that would have taken years or even centuries of computer time to produce with the technology of just a few years ago can now be produced in hours, minutes, seconds or even less. Second, there are large numbers of interesting problems for which there's no known algorithmic approach other than sheer brute force. We investigate many problem-solving methodologies in this book. We'll consider brute-force approaches to various interesting problems.

4.28 ***(Calculating Weekly Pay)*** A company pays its employees as managers (who receive a fixed weekly salary), hourly workers (who receive a fixed hourly wage for up to the first 40 hours they work and "time-and-a-half" for overtime hours worked), commission workers (who receive $250 plus 5.7% of their gross weekly sales), or piecework workers (who receive a fixed amount of money for each of the items they produce—each pieceworker in this company works on only one type of item). Write a program to compute each employee's weekly pay. You do not know the number of employees in advance. Each type of employee has a pay code: Managers have paycode 1, hourly workers have code 2, commission workers have code 3 and pieceworkers have code 4. Use a `switch` to compute each employee's pay based on the paycode. Within the `switch`, prompt the user to enter the appropriate facts your program needs to calculate each employee's pay based on that employee's paycode. [*Note*: You can input values of type `double` using the conversion specification `%lf` with `scanf`.]

4.29 ***(De Morgan's Laws)*** We discussed the logical operators `&&`, `||`, and `!`. De Morgan's Laws help express logical expressions more conveniently. These laws state that the expression `!(`*condition1* `&&` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `||` `!`*condition2*`)`. Also, the expression `!(`*condition1* `||` *condition2*`)` is logically equivalent to the expression `(!`*condition1* `&&` `!`*condition2*`)`. Use De Morgan's Laws to write equivalent expressions for each of the following, and then write a program to show that both the original expression and the new expression in each case are equivalent.

a) `!(x < 5) && !(y >= 7)`
b) `!(a == b) || !(g != 5)`
c) `!((x <= 8) && (y > 4))`
d) `!((i > 4) || (j <= 6))`

4.30 ***(Replacing `switch` with `if...else`)*** Rewrite Fig. 4.5 by replacing the `switch` with a nested `if...else` statement. Be careful to deal with the `default` case properly. Next, rewrite this new version by replacing the nested `if...else` statement with a series of `if` statements. Here, too, be careful to deal with the `default` case properly. This exercise demonstrates that `switch` is a convenience and that any `switch` statement can be written with only single-selection statements.

4.31 ***(Diamond-Printing Program)*** Write a program that prints the following diamond shape. Your `printf` statements may print either one asterisk (`*`) or one blank. Use nested `for` statements and minimize the number of `printf` statements.

```
    *
   ***
  *****
 *******
*********
 *******
  *****
   ***
    *
```

4.32 ***(Modified Diamond-Printing Program)*** Modify the program you wrote in Exercise 4.31 to read an odd number in the range 1 to 19 to specify the number of rows in the diamond. Your program should then display a diamond of the appropriate size.

4.33 ***(Roman-Numeral Equivalent of Decimal Values)*** Write a program that prints a table of the Roman-numeral equivalents for the decimal numbers in the range 1 to 100.

4.34 Describe how you'd replace a `do...while` loop with an equivalent `while`. What problem occurs when you try to replace a `while` loop with an equivalent `do...while` loop? Suppose you've been told that you must remove a `while` loop and replace it with a `do...while`. What additional control statement would you need to use? How would you use it to ensure that the resulting program behaves exactly like the original?

4.35 A criticism of the `break` and `continue` statements is that each is unstructured. Actually, `break` and `continue` statements can always be replaced by structured statements, though doing so can be awkward. Describe how you'd remove any `break` statement from a loop and replace that statement with some structured equivalent. [*Hint*: The `break` statement terminates a loop from the loop body. The other way to leave is by failing the loop-continuation test. Consider using in the loop-continuation test a second test that indicates "early exit because of a 'break' condition."] Use the technique you developed here to remove the `break` statement from the program of Fig. 4.7.

4.36 What does the following program segment do?

```
1  int n = 4, a = 1;
2     int i, c;
3        for (i = 1; i <= n; i++)
4           {
5           for (c = 1; c <= i; c++)
6              {
7               printf("%d", a);
8               a++
9              }
10       printf("\n");
11 }
```

4.37 Describe in general how you would remove any `continue` statement from a loop in a program and replace that statement with some structured equivalent.

Use the technique you developed here to remove the `continue` statement from the program of Fig. 4.8.

4.38 ***("The Twelve Days of Christmas" Song)*** Write a program that uses iteration and `switch` statements to print the song "The Twelve Days of Christmas." One `switch` statement should be used to print the day (i.e., "first," "second," etc.). A separate `switch` statement should be used to print the remainder of each verse.

4.39 ***(Limitations of Floating-Point Numbers for Monetary Amounts)*** Section 4.5 cautioned about using floating-point values for monetary calculations. Try this experiment: Create a `float` variable with the value `1000000.00`. Next, add to that variable the literal `float` value `0.12f`. Display the result using `printf` and the conversion specification `"%.2f"`. What do you get?