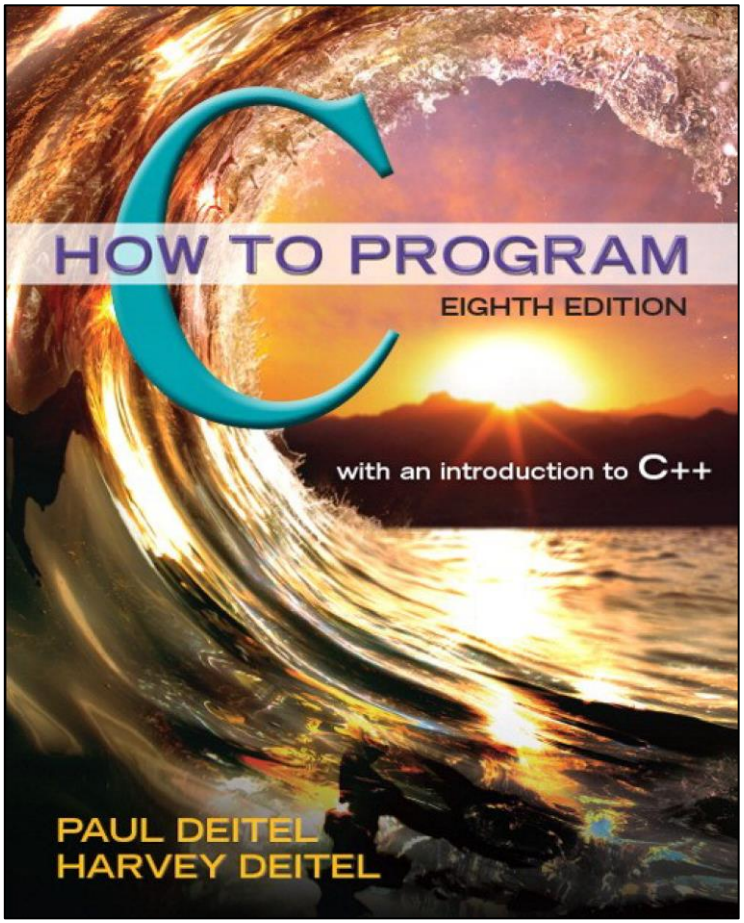
C How to Program

Eighth Edition



Chapter 4

C Program Control

Learning Objectives

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

Outline (1 of 2)

Outline (2 of 2)

4.1 Introduction

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

4.2 Iteration Essentials (1 of 3)

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

4.2 Iteration Essentials (2 of 3)

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

4.2 Iteration Essentials (3 of 3)

- 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.
- The sentinel value indicates "end of data."
- The sentinel is entered after all regular data items have been supplied to the program.
- Sentinels must be distinct from regular data items.

4.3 Counter-Controlled Iteration (1 of 5)

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

4.3 Counter-Controlled Iteration (2 of 5)

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

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

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

Figure 4.1 Counter-Controlled Iteration

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

int main(void)
{
   unsigned int counter = 1; // initialization

   while (counter <= 10) { // iteration condition
      printf ("%u\n", counter);
      ++counter; // increment
   }
}
```

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

4.3 Counter-Controlled Iteration (3 of 5)

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

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

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

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

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

4.3 Counter-Controlled Iteration (4 of 5)

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

4.3 Counter-Controlled Iteration (5 of 5)

- You could make the program in Figure 4.1 (see slide 11) more concise by initializing `counter` to `0` and by replacing the `while` statement with

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

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

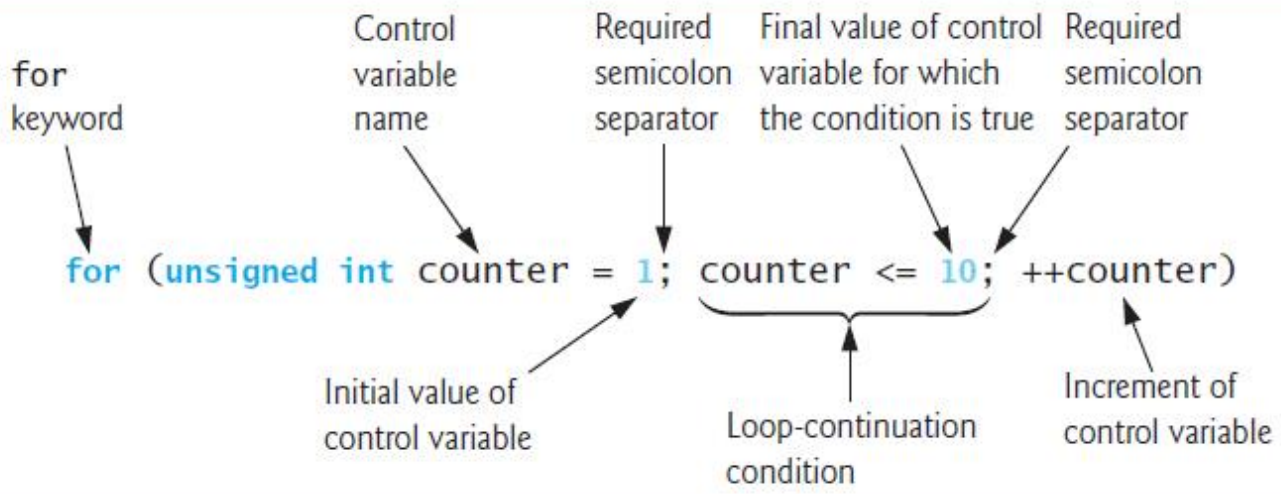
4.4 `for` Iteration Statement (1 of 10)

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

Figure 4.2 Counter-Controlled Iteration with the for Statement

```
1  // Fig. 4.2: fig04_02.c
2  // Counter-controlled iteration with the for statement.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     // initialization, iteration condition, and increment
8     //  are all included in the for statement header.
9     for (unsigned int counter = 1; counter <= 10; ++counter) {
10       printf("%u\n", counter);
11    }
12 }
```

Figure 4.3 for Statement Header Components



4.4 for Iteration Statement (6 of 10)

General Format of a for Statement

- The general format of the for statement is

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

where the **initialization** expression initializes the loop-control variable (and might define it), the **condition** expression is the loop-continuation condition and the **increment** expression increments the control variable.

4.5 for Statement: Notes and Observations (1 of 2)

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

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

  is equivalent to the statement

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

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

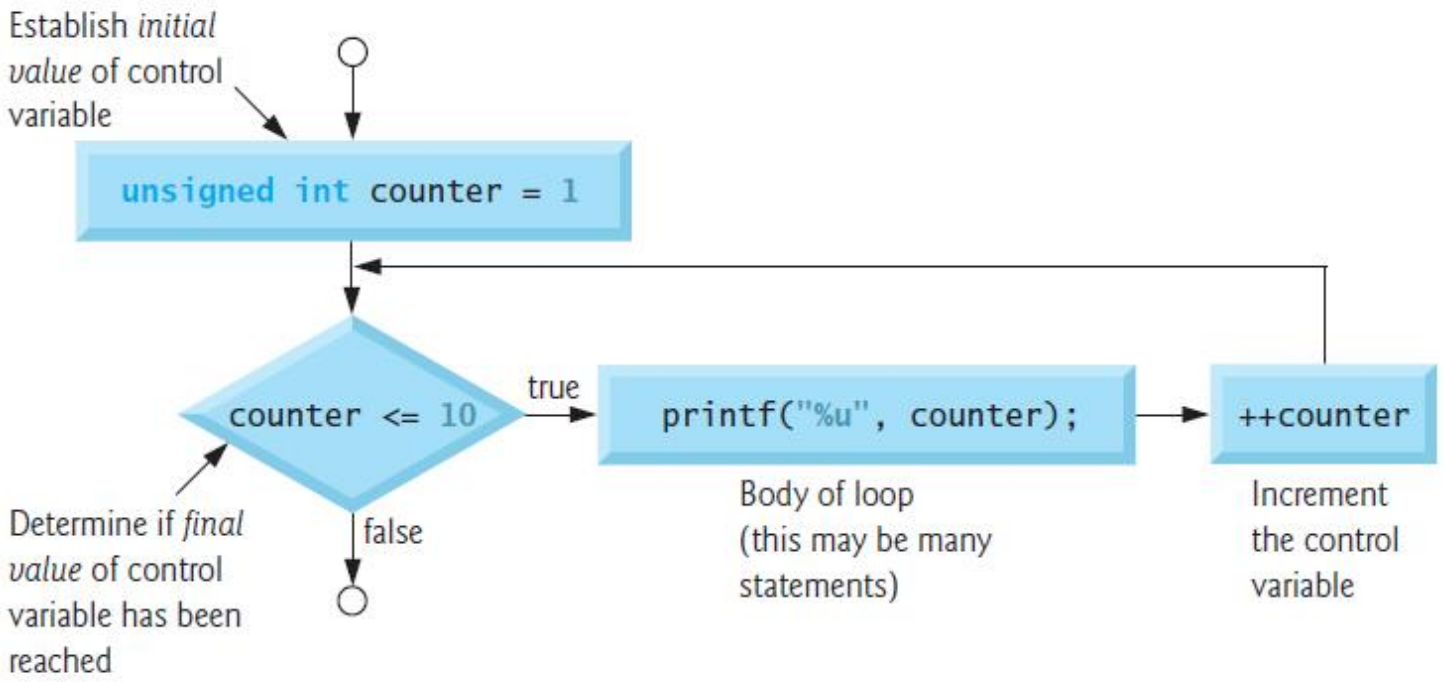
4.5 for Statement: Notes and Observations (2 of 2)

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

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

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

Figure 4.4 Flowcharting a Typical for Iteration Statement



4.6 Examples Using the `for` Statement (1 of 14)

- The following examples show methods of varying the control variable in a `for` statement.
  - Vary the control variable from `1` to `100` in increments of `1`.

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

  - Vary the control variable from `100` to `1` in increments of `−1` (decrements of `1`).

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

  - Vary the control variable from `7` to `77` in steps of `7`.

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

4.6 Examples Using the `for` Statement (2 of 14)

- Vary the control variable from 20 to 2 in steps of −2.

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

- Vary the control variable over the following sequence of values: 2, 5, 8, 11, 14, 17.

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

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

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

4.6 Examples Using the `for` Statement (3 of 14)

Application: Summing the Even Integers from 2 to 100

- Figure 4.5 uses the `for` statement to sum all the even integers from **`2`** to **`100`**.

Figure 4.5 Summation with for

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

int main(void)
{
   unsigned int sum = 0; // initialize sum

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

   printf("Sum is %u\n", sum);
}
```

```
Sum is 2550
```

4.6 Examples Using the `for` Statement (5 of 14)

Application: Compound-Interest Calculations

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

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

    where

    p is the original amount invested (i.e., the principal)
    r is the annual interest rate
    n is the number of years
    a is the amount on deposit at the end of the n^{th} year.

Figure 4.6 Calculating Compound Interest (1 of 2)

```
1  // Fig. 4.6: fig04_06.c
2  // Calculating compound interest.
3  #include <stdio.h>
4  #include <math.h>
5
6  int main(void)
7  {
8     double principal = 1000.0; // starting principal
9     double rate = .05; // annual interest rate
10
11    // output table column heads
12    printf("%4s%21s\n", "Year", "Amount on deposit");
13
14    // calculate amount on deposit for each of ten years
15    for (unsigned int year = 1; year <= 10; ++year) {
16
17       // calculate new amount for specified year
18       double amount = principal * pow(1.0 + rate, year);
19
20       // output one table row
21       printf("%4u%21.2f\n", year, amount);
22    }
23 }
```

Figure 4.6 Calculating Compound Interest (2 of 2)

```
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
```

4.6 Examples Using the `for` Statement (8 of 14)

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

4.6 Examples Using the `for` Statement (13 of 14)

Formatting Numeric Output

- The conversion specifier `%21.2f` is used to print the value of the variable `amount` in the program.
- The `21` in the conversion specifier denotes the **field width** in which the value will be printed.
- A field width of `21` specifies that the value printed will appear in `21` print positions.
- The `2` specifies the **precision** (i.e., the number of decimal positions).

4.6 Examples Using the `for` Statement (14 of 14)

- If the number of characters displayed is less than the field width, then the value will automatically be **right justified** in the field.
- This is particularly useful for aligning floating-point values with the same precision (so that their decimal points align vertically).
- To **left justify** a value in a field, place a - (minus sign) between the % and the field width.
- The minus sign may also be used to left justify integers (such as in `%-6d`) and character strings (such as in `%-8s`).

4.7 `switch` Multiple-Selection Statement (1 of 19)

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

Figure 4.7 Counting Letter Grades with `switch` (1 of 5)

```
1  // Fig. 4.7: fig04_07.c
2  // Counting letter grades with switch.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     unsigned int aCount = 0;
8     unsigned int bCount = 0;
9     unsigned int cCount = 0;
10    unsigned int dCount = 0;
11    unsigned int fCount = 0;
12
13    puts("Enter the letter grades.");
14    puts("Enter the EOF character to end input.");
15    int grade; // one grade
16
```

Figure 4.7 Counting Letter Grades with `switch` (2 of 5)

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

Figure 4.7 Counting Letter Grades with `switch` (3 of 5)

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

Figure 4.7 Counting Letter Grades with `switch` (4 of 5)

```
60      // output summary of results
61      puts("\nTotals for each letter grade are:");
62      printf("A: %u\n", aCount);
63      printf("B: %u\n", bCount);
64      printf("C: %u\n", cCount);
65      printf("D: %u\n", dCount);
66      printf("F: %u\n", fCount);
67  }
```

Figure 4.7 Counting Letter Grades with `switch` (5 of 5)

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

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

4.7 `switch` Multiple-Selection Statement (2 of 19)

Reading Character Input

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

4.7 `switch` Multiple-Selection Statement (3 of 19)

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

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

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

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

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

4.7 `switch` Multiple-Selection Statement (4 of 19)

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

4.7 `switch` Multiple-Selection Statement (5 of 19)

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

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

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

4.7 `switch` Multiple-Selection Statement (6 of 19)

- 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've chosen to represent characters in this program as `int`s because `EOF` has an integer value (normally `-1`).

4.7 `switch` Multiple-Selection Statement (7 of 19)

Entering the `EOF` Indicator

- On Linux/UNIX/MacOSX systems, the `EOF` indicator is entered by typing

  `<Ctrl>` *d*

- on a line by itself.
- This notation **`<Ctrl>`** *d* means to press the **Enter** key then simultaneously press both **Ctrl** and *d*.
- On other systems, such as Microsoft Windows, the `EOF` indicator can be entered by typing

  `<Ctrl>` *z*

- You may also need to press **Enter** on Windows**.**

4.7 `switch` Multiple-Selection Statement (8 of 19)

- The user enters grades at the keyboard.
- When the **Enter** key is pressed, the characters are read by function `getchar` one character at a time.
- If the character entered is not equal to `EOF`, the `switch` statement (line 22) is entered.

4.7 `switch` Multiple-Selection Statement (9 of 19)

`switch` Statement Details

- Keyword `switch` is followed by the variable name `grade` in parentheses.
- This is called the **controlling expression**.
- The value of this expression is compared with each of the **`case` labels**.
- Assume the user has entered the letter `C` as a grade.
- `C` is automatically compared to each `case` in the `switch`.
- If a match occurs (`case 'C':`), the statements for that `case` are executed.

4.7 `switch` Multiple-Selection Statement (10 of 19)

- In the case of the letter `C`, `cCount` is incremented by `1` (line 36), and the `switch` statement is exited immediately with the `break` statement.
- The `break` statement causes program control to continue with the first statement after the `switch` statement.
- The `break` statement is used because the `cases` in a `switch` statement would otherwise run together.

4.7 `switch` Multiple-Selection Statement (11 of 19)

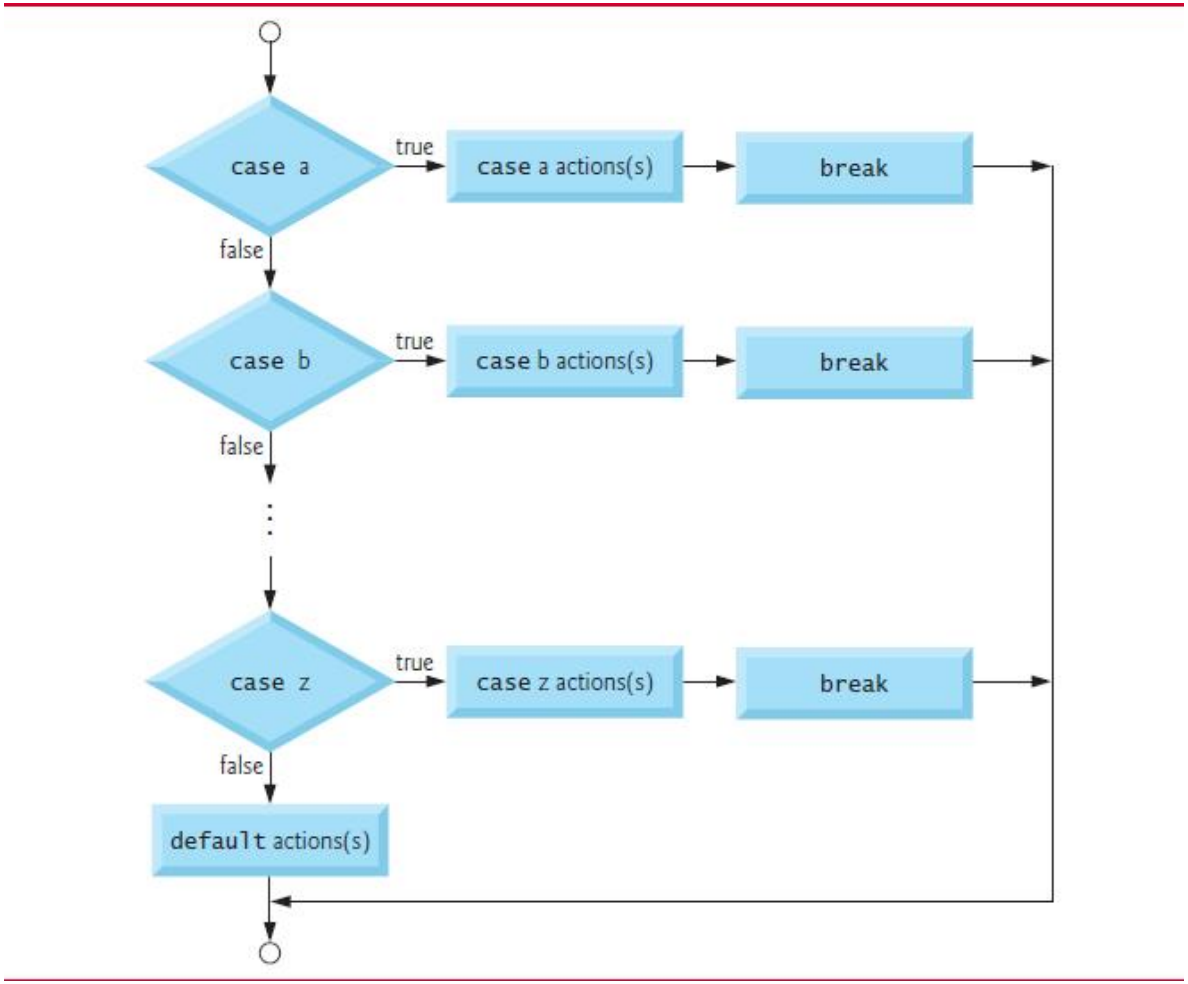
- If `break` is not used anywhere in a `switch` statement, then each time a match occurs in the statement, the statements for all the remaining `cases` will be executed—called fallthrough.
- If no match occurs, the `default` case is executed, and an error message is printed.

4.7 `switch` Multiple-Selection Statement (12 of 19)

`switch` Statement Flowchart

- Each `case` can have one or more actions.
- The `switch` statement is different from all other control statements in that braces are not required around multiple actions in a `case` of a `switch`.
- The general `switch` multiple-selection statement (using a `break` in each `case`) is flowcharted in Figure 4.8 (see slide 83).
- The flowchart makes it clear that each `break` statement at the end of a `case` causes control to immediately exit the `switch` statement.

Figure 4.8 `switch` Multiple-Selection Statement with `breaks`



4.7 switch Multiple-Selection Statement (13 of 19)

Ignoring Newline, Tab and Blank Characters in Input

- In the switch statement of Figure 4.7 (see slide 61-65), the lines

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

  cause the program to skip newline, tab and blank characters.

- Reading characters one at a time can cause some problems.
- To have the program read the characters, you must send to the computer by pressing the **Enter** key**.**
- This causes the newline character to be placed in the input after the character we wish to process.

4.7 `switch` Multiple-Selection Statement (14 of 19)

- Often, this newline character must be specially processed to make the program work correctly.
- By including the preceding cases in our `switch` statement, we prevent the error message in the `default` case from being printed each time a newline, tab or space is encountered in the input.
- So each input causes two iterations of the loop—the first for the letter grade and the second for `'\n'`.

4.7 `switch` Multiple-Selection Statement (15 of 19)

- Listing several case labels together (such as `case 'D':` `case 'd':`) simply means that the **same** set of actions is to occur for either of these cases.

4.7 `switch` Multiple-Selection Statement (16 of 19)

Constant Integral Expressions

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

4.7 `switch` Multiple-Selection Statement (17 of 19)

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

4.7 `switch` Multiple-Selection Statement (18 of 19)

Notes on Integral Types

- Portable languages like C must have flexible data type sizes.
- Different applications may need integers of different sizes.
- C provides several data types to represent integers.
- 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`), as well as `unsigned` variations of all the integral types.
- The C standard specifies the minimum range of values for each integer type, but the actual range may be greater and depends on the implementation.
- For `short int`s the minimum range is −32767 to +32767.
- For most integer calculations, `long int`s are sufficient.

4.7 `switch` Multiple-Selection Statement (19 of 19)

- The minimum range of values for `long ints` is −2147483647 to +2147483647.
- The range of values for an `int` greater than or equal to that of a `short int` and less than or equal to that of a `long int`.
- The data type `signed char` can be used to represent integers in the range −127 to +127 or any of the characters in the computer's character set.

4.8 do…while Iteration Statement (1 of 5)

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

4.8 do…while Iteration Statement (2 of 5)

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

```
while (condition)
```

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

4.8 do…while Iteration Statement (3 of 5)

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

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

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

Figure 4.9 Using the do...while Iteration Statement

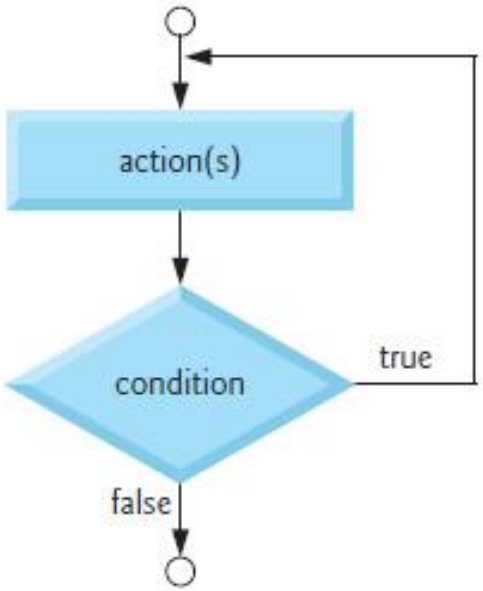
```c
// Fig. 4.9: fig04_09.c
// Using the do...while iteration statement.
#include <stdio.h>

int main(void)
{
   unsigned int counter = 1; // initialize counter

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

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

Figure 4.10 Flowcharting the do…while iteration statement



4.9 break and continue Statements (1 of 3)

- The break and continue statements are used to alter the flow of control.

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.
- Common uses of the break statement are to escape early from a loop or to skip the remainder of a switch statement (as in Figure 4.7 (see slide 61-65)).

4.9 `break` and `continue` Statements (2 of 3)

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

Figure 4.11 Using the break Statement in a for Statement

```
1  // Fig. 4.11: fig04_11.c
2  // Using the break statement in a for statement.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     unsigned int x; // declared here so it can be used after loop
8
9     // loop 10 times
10    for (x = 1; x <= 10; ++x) {
11
12       // if x is 5, terminate loop
13       if (x == 5) {
14          break; // break loop only if x is 5
15       }
16
17       printf("%u ", x);
18    }
19
20    printf("\nBroke out of loop at x == %u\n", x);
21 }
```

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

4.9 `break` and `continue` Statements (3 of 3)

`continue` Statement

- The `continue` statement, when executed in a `while`, `for` or `do…while` statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop.
- In `while` and `do…while` statements, the loop-continuation test is evaluated immediately **after** the `continue` statement is executed.
- In the `for` statement, the increment expression is executed, then the loop-continuation test is evaluated.
- Figure 4.12 uses the `continue` statement in a `for` statement to skip the `printf` statement and begin the next iteration of the loop.

Figure 4.12 Using the `continue` Statement in a `for` Statement

```
1  // Fig. 4.12: fig04_12.c
2  // Using the continue statement in a for statement.
3  #include <stdio.h>
4
5  int main(void)
6  {
7     // loop 10 times
8     for (unsigned int x = 1; x <= 10; ++x) {
9
10       // if x is 5, continue with next iteration of loop
11       if (x == 5) {
12          continue; // skip remaining code in loop body
13       }
14
15       printf("%u ", x);
16    }
17
18    puts("\nUsed continue to skip printing the value 5");
19 }
```

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

4.10 Logical Operators (1 of 12)

- C provides **logical operators** that may be used to form more complex conditions by combining simple conditions.
- The logical operators are **&&** **(logical AND)**, **||** **(logical OR)** and **!** (**logical NOT** also called **logical negation**).

4.10 Logical Operators (2 of 12)

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

4.10 Logical Operators (3 of 12)

- 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 count of `seniorFemales` is incremented by `1`.
- If **either** or **both** of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the `if`.
- Figure 4.13 (see slide 111) summarizes the **`&&` operator**.

4.10 Logical Operators (4 of 12)

- 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.

Figure 4.13 Truth Table for the Logical AND (&&) Operator

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

4.10 Logical Operators (5 of 12)

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)
   printf("Student grade is A");:
```

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

4.10 Logical Operators (6 of 12)

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

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

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

Figure 4.14 Truth Table for the Logical OR (||) Operator

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

4.10 Logical Operators (7 of 12)

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

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

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

4.10 Logical Operators (8 of 12)

Logical Negation (!) Operator

- C provides ! (logical negation) to enable you to "reverse" the meaning of a condition.
- The logical negation operator has only a single condition as an operand (and is therefore a unary operator).
- Placed before a condition when we're interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

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

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

Figure 4.15 Truth Table for Operator ! (Logical Negation)

expression	! expression
0	1
nonzero	0

4.10 Logical Operators (9 of 12)

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

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

4.10 Logical Operators (10 of 12)

Summary of Operator Precedence and Associativity

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

Figure 4.16 Operator Precedence and Associativity

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

4.10 Logical Operators (11 of 12)

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 C's convention of using zero and nonzero values to represent false and true—the value 0 in a condition evaluates to false, while any nonzero value evaluates to true.
- Assigning any non-zero 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.

4.10 Logical Operators (12 of 12)

- At preprocessor time, `bool`, `true` and `false` are replaced with `_Bool`, `1` and `0`.
- Section F.8 presents an example that uses `bool`, `true` and `false`.
- The example uses a programmer-defined function, a concept we introduce in Chapter 5.
- Microsoft Visual C++ does not implement the `_Bool` data type.

Copyright