Outline

4.1 Introduction

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

4.2 Repetition Essentials

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

1. Counter-controlled repetition
2. Sentinel-controlled repetition

Counter-controlled repetition is sometimes called *definite repetition* because we know *in advance* exactly how many times the loop will be executed. Sentinel-controlled repetition is sometimes called *indefinite repetition* because it's *not known* in advance how many times the loop will be executed.

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

Sentinel values are used to control repetition when:

1. The precise number of repetitions isn't known in advance, and
2. The loop includes statements that obtain data each time the loop is performed.

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

4.3 Counter-Controlled Repetition

Counter-controlled repetition requires:

1. The name of a control variable (or loop counter).
2. The initial value of the control variable.
3. The increment (or decrement) by which the control variable is modified each time through the loop.
4. The condition that tests for the final value of the control variable (i.e., whether looping should continue).

Consider the simple program shown in Fig. 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`.

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

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

   while ( counter <= 10 ) { // repetition condition
      printf ( "%u\n", counter ); // display counter
      ++counter; // increment
   } // end while
} // end function main
```

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

Fig. 4.1 | Counter-controlled repetition.

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. 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`).

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

```
while ( ++counter <= 10 )
   printf( "%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. Coding in such a condensed fashion takes some practice. Some programmers feel that this makes the code too cryptic and error prone.

Common Programming Error 4.1

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

Error-Prevention Tip 4.1

Control counting loops with integer values.

Good Programming Practice 4.1

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

Good Programming Practice 4.2

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

4.4 for Repetition Statement

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

The program operates as follows. When the `for` statement begins executing, the control variable `counter` is initialized to `1`. Then, the loop-continuation condition `counter <= 10` is checked. Because the initial value of `counter` is `1`, the condition is satisfied, so the `printf` statement (line 13) prints the value of `counter`, namely `1`. The control variable `counter` is then incremented by the expression `++counter`, and the loop begins again with the loop-continuation test. Because the control variable is now equal to `2`, the final value is not exceeded, so the program performs the `printf` statement again. This process continues until the control variable `counter` is incremented to its final value of 11—this causes the

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

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

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

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

loop-continuation test to fail, and repetition terminates. The program continues by performing the first statement after the `for` statement (in this case, the end of the program).

`for` Statement Header Components

Figure 4.3 takes a closer look at the `for` statement of Fig. 4.2. Notice that the `for` statement "does it all"—it specifies each of the items needed for counter-controlled repetition with a control variable. If there's more than one statement in the body of the `for`, braces are required to define the body of the loop.

The C standard allows you to declare the control variable in the initialization section of the `for` header (as in `int counter = 1`). We show a complete code example of this in Appendix F. This feature is not supported in Microsoft Visual C++.

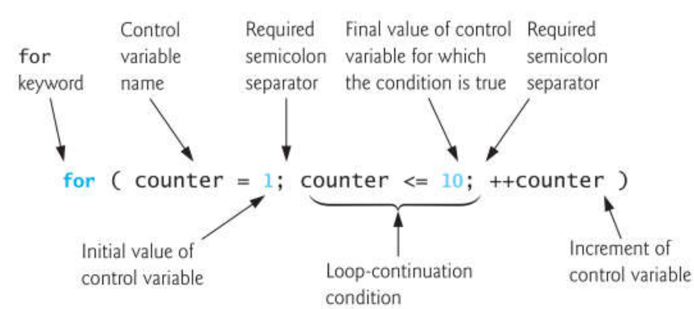

Fig. 4.3 | `for` statement header components.

Off-By-One Errors

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

Error-Prevention Tip 4.2

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

General Format of a `for` *Statement*

The general format of the `for` statement is

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

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

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

There's an exception to this rule, which we discuss in Section 4.9.

Comma-Separated Lists of Expressions

Often, *expression1* and *expression3* are comma-separated lists of expressions. The commas as used here are actually comma operators that guarantee that lists of expressions evaluate from left to right. The value and type of a comma-separated list of expressions are the value and type of the rightmost expression in the list. The comma operator is most often used in the `for` statement. Its primary use is to enable you to use multiple initialization and/or multiple increment expressions. For example, there may be two control variables in a single `for` statement that must be initialized and incremented.

Software Engineering Observation 4.1

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

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

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

Increment Expression Acts Like a Standalone Statement

The increment expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`. Therefore, the expressions

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

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

Common Programming Error 4.2

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

Common Programming Error 4.3

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

4.5 for Statement: Notes and Observations

1. 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 )
```

2. The "increment" may be negative (in which case it's really a *decrement* and the loop actually counts *downward*).
3. 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. The control variable is frequently printed or used in calculations in the body of a loop, but it need not be. It's common to use the control variable for controlling repetition while never mentioning it in the body of the loop.
5. The `for` statement is flowcharted much like the `while` statement. For example, Fig. 4.4 shows the flowchart of the `for` statement

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

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

Error-Prevention Tip 4.3

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

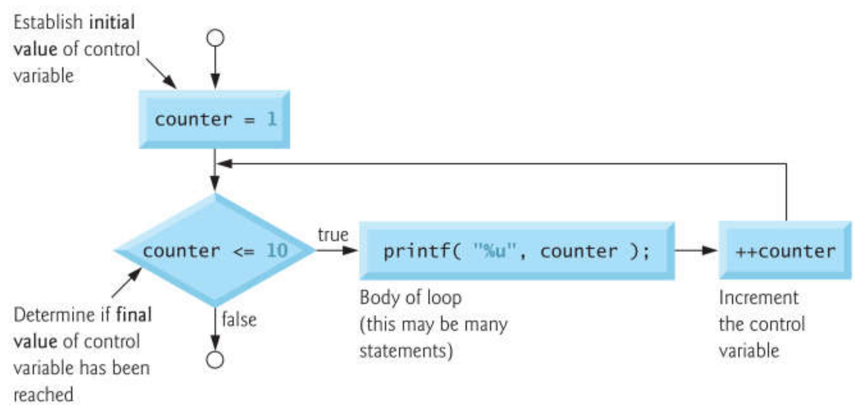

Fig. 4.4 | Flowcharting a typical `for` repetition statement.

4.6 Examples Using the for Statement

The following examples show methods of varying the control variable in a `for` statement.

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

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

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

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

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

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

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

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

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

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

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

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

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`. Each iteration of the loop (lines 11–13) adds control variable `number`'s value to variable `sum`.

```
1  // Fig. 4.5: fig04_05.c
2  // Summation with for.
3  #include <stdio.h>
4
```

Fig. 4.5 | Summation with `for`. (Part 1 of 2.)

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

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

   printf( "Sum is %u\n", sum ); // output sum
} // end function main
```

```
Sum is 2550
```

Fig. 4.5 | Summation with `for`. (Part 2 of 2.)

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

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

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

Good Programming Practice 4.3

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

Good Programming Practice 4.4

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

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 that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:*
>
> $a = p(1 + r)^n$
>
> *where*
>
> *p is the original amount invested (i.e., the principal)*
> *r is the annual interest rate*
> *n is the number of years*
> *a is the amount on deposit at the end of the n^{th} year.*

This problem involves a loop that performs the indicated calculation for each of the 10 years the money remains on deposit. The solution is shown in Fig. 4.6.

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

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

Fig. 4.6 | Calculating compound interest.

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

1. On many Linux/UNIX C compilers, you must include the `-lm` option (e.g., `gcc -lm fig04_06.c`) when compiling Fig. 4.6. This links the math library to the program.

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. This information is contained in something called `pow`'s function prototype. Function prototypes are explained in Chapter 5. We also provide a summary of the `pow` function and other math library functions in Chapter 5.

A Caution about Using Type `float` or `double` for Monetary Amounts

Notice that we defined the variables `amount`, `principal` and `rate` to be of type `double`. We did this for simplicity because we're dealing with fractional parts of dollars.

> **Error-Prevention Tip 4.4**
>
> *Do not use variables of type `float` or `double` to perform monetary calculations. The impreciseness of floating-point numbers can cause errors that will result in incorrect monetary values. [In this chapter's exercises, we explore the use of integers to perform precise monetary calculations.]*

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

```
  14.23
+ 18.67
-------
  32.91
```

Clearly the sum of the individual numbers as printed should be 32.90! You've been warned!

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). 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`). We'll discuss the powerful formatting capabilities of `printf` and `scanf` in detail in Chapter 9.

4.7 `switch` Multiple-Selection Statement

In Chapter 3, we discussed the `if` single-selection statement and the `if…else` double-selection statement. Occasionally, an algorithm will contain a *series of decisions* in which a variable or expression is tested separately for each of the constant integral values it may assume, and different actions are taken. This is called *multiple selection*. C provides the `switch` multiple-selection statement to handle such decision making.

The `switch` statement consists of a series of `case` labels, an optional `default` case and statements to execute for each case. Figure 4.7 uses `switch` to count the number of each different letter grade students earned on an exam.

```
// Fig. 4.7: fig04_07.c
// Counting letter grades with switch.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int grade; // one grade
   unsigned int aCount = 0; // number of As
   unsigned int bCount = 0; // number of Bs
   unsigned int cCount = 0; // number of Cs
   unsigned int dCount = 0; // number of Ds
   unsigned int fCount = 0; // number of Fs

   puts( "Enter the letter grades."  );
   puts( "Enter the EOF character to end input."  );

   // loop until user types end-of-file key sequence
   while ( ( grade = getchar() ) != EOF ) {

      // determine which grade was input
      switch ( grade ) { // switch nested in while

         case 'A': // grade was uppercase A
         case 'a': // or lowercase a
            ++aCount; // increment aCount
            break; // necessary to exit switch

         case 'B': // grade was uppercase B
         case 'b': // or lowercase b
            ++bCount; // increment bCount
            break; // exit switch

         case 'C': // grade was uppercase C
         case 'c': // or lowercase c
            ++cCount; // increment cCount
            break; // exit switch

         case 'D': // grade was uppercase D
         case 'd': // or lowercase d
            ++dCount; // increment dCount
            break; // exit switch

         case 'F': // grade was uppercase F
         case 'f': // or lowercase f
            ++fCount; // increment fCount
            break; // exit switch

```

Fig. 4.7 | Counting letter grades with `switch`. (Part 1 of 2.)

```
49          case '\n': // ignore newlines,
50          case '\t': // tabs,
51          case ' ': // and spaces in input
52             break; // exit switch
53
54          default: // catch all other characters
55             printf( "%s", "Incorrect letter grade entered." );
56             puts( " Enter a new grade." );
57             break; // optional; will exit switch anyway
58       } // end switch
59    } // end while
60
61    // output summary of results
62    puts( "\nTotals for each letter grade are:" );
63    printf( "A: %u\n", aCount ); // display number of A grades
64    printf( "B: %u\n", bCount ); // display number of B grades
65    printf( "C: %u\n", cCount ); // display number of C grades
66    printf( "D: %u\n", dCount ); // display number of D grades
67    printf( "F: %u\n", fCount ); // display number of F grades
68 } // end function main
```

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

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

Fig. 4.7 | Counting letter grades with `switch`. (Part 2 of 2.)

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 stores that character in the integer variable `grade`. Characters are normally stored in variables of type **`char`**. However, an important feature of C is that characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer. 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. 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 `=`. 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 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"). 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 (again, normally `-1`).

Portability Tip 4.1

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

Portability Tip 4.2

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

Entering the `EOF` Indicator

On Linux/UNIX/Mac OS X systems, the `EOF` indicator is entered by typing

```
<Ctrl> d
```

on a line by itself. This notation *<Ctrl> d* means to press the *Enter* key and then 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 may 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 character at a time. If the character entered is not equal to `EOF`, the `switch` statement (line 22–58) is entered.

`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. In the case of the letter `C`, `cCount` is incremented by `1` (line 36), and the `switch` statement is exited immediately with the `break` statement.

The `break` statement causes program control to continue with the first statement after the `switch` statement. The `break` statement is used because the `case`s in a `switch` statement would otherwise run together. If `break` is *not* used anywhere in a `switch` statement, then each time a match occurs in the statement, the statements for *all* the remaining `case`s will be executed. (This feature is rarely useful, although it's perfect for programming Exercise 4.38—the iterative song *The Twelve Days of Christmas*!) If no match occurs, the `default` case is executed, and an error message is printed.

`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 Fig. 4.8. The flowchart makes it clear that each `break` statement at the end of a `case` causes control to immediately exit the `switch` statement.

Common Programming Error 4.4

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

Software Engineering Observation 4.2

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

Good Programming Practice 4.5

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.

Good Programming Practice 4.6

In a `switch` statement when the `default` clause is last, the `break` statement isn't required. You may prefer to include this `break` for clarity and symmetry with other `case`s.

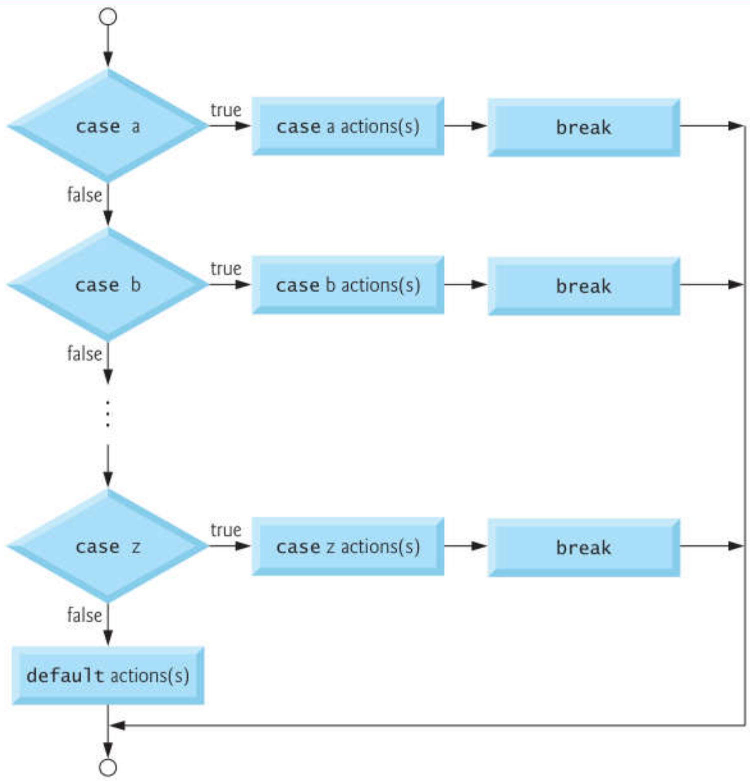

Fig. 4.8 | switch multiple-selection statement with breaks.

Ignoring Newline, Tab and Blank Characters in Input

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

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

cause the program to skip newline, tab and blank characters. Reading characters one at a time can cause some problems. To have the program read the characters, you must send them to the computer by pressing the *Enter* key. This causes the newline character to be placed in the input after the character we wish to process. Often, this newline character must be specially processed to make the program work correctly. By including the preceding cases in our `switch` statement, we prevent the error message in the `default` case from being printed each time a newline, tab or space is encountered in the input.

Error-Prevention Tip 4.5

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

Listing several case labels together (such as `case 'D': case 'd':` in Fig. 4.7) simply means that the *same* set of actions is to occur for either of these cases.

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'`. 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've used character constants. Remember that characters are represented as small integer values.

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`). 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. The minimum range of values for `long int`s 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`. 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 computer's character set. See section 5.2.4.2 of the C standard document for the complete list of `signed` and `unsigned` integer-type ranges.

4.8 do…while Repetition Statement

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

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

Good Programming Practice 4.7

To eliminate the potential for ambiguity, you may want to include braces in `do…while` *statements, even if they're not necessary.*

Common Programming Error 4.5

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

Figure 4.9 uses a `do…while` statement to print the numbers from 1 to 10. The control variable `counter` is preincremented in the loop-continuation test.

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

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

   do {
      printf( "%u  ", counter ); // display counter
   } while ( ++counter <= 10 ); // end do...while
} // end function main
```

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

Fig. 4.9 | Using the `do…while` repetition statement.

`do…while` *Statement Flowchart*

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

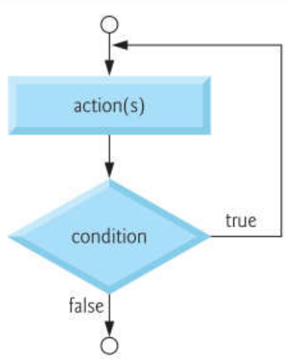

Fig. 4.10 | Flowcharting the `do…while` repetition statement.

4.9 break and continue Statements

The `break` and `continue` statements are used to alter the flow of control. Section 4.7 showed how `break` can be used to terminate a `switch` statement's execution. This section discusses how to use `break` in a repetition 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. Common uses of the `break` statement are to escape early from a loop or to skip the remainder of a `switch` statement (as in Fig. 4.7). Figure 4.11 demonstrates the break statement in a `for` repetition statement. When the `if` statement detects that `x` has become `5`, `break` is executed. This terminates the `for` statement, and the program continues with the `printf` after the `for`. The loop fully executes only four times.

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

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

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

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

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

   printf( "\nBroke out of loop at x == %u\n", x );
} // end function main
```

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

Fig. 4.11 | Using the `break` statement in a `for` statement.

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. Earlier, we said that the `while` statement could be used in most cases to represent the `for` statement.

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

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

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

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

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

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

   puts( "\nUsed continue to skip printing the value 5" );
} // end function main
```

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

Fig. 4.12 | Using the `continue` statement in a `for` statement.

Software Engineering Observation 4.3

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

Performance Tip 4.1

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

Software Engineering Observation 4.4

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