Chapter 3 Structured Program Development in C

C How to Program, 7/e

OBJECTIVES

In this chapter, you'll:

- Use basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Use the `if` selection statement and the `if...else` selection statement to select actions.
- Use the `while` repetition statement to execute statements in a program repeatedly.
- Use counter-controlled repetition and sentinel-controlled repetition.
- Learn structured programming.
- Use increment, decrement and assignment operators.

3.1 Introduction
3.2 Algorithms
3.3 Pseudocode
3.4 Control Structures
3.5 The `if` Selection Statement
3.6 The `if...else` Selection Statement
3.7 The `while` Repetition Statement
3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition
3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition
3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements
3.11 Assignment Operators
3.12 Increment and Decrement Operators
3.13 Secure C Programming

3.1 Introduction

- Before writing a program to solve a particular problem, we must have a thorough understanding of the problem and a carefully planned solution approach.
- The next two chapters discuss techniques that facilitate the development of structured computer programs.

3.2 Algorithms

- The solution to any computing problem involves executing a series of actions in a specific order.
- A procedure for solving a problem in terms of
  - the actions to be executed, and
  - the order in which these actions are to be executed
- is called an algorithm.
- Correctly specifying the order in which the actions are to be executed is important.

3.2 Algorithms (Cont.)

- Consider the “rise-and-shine algorithm” followed by one junior executive for getting out of bed and going to work: (1) Get out of bed, (2) take off pajamas, (3) take a shower, (4) get dressed, (5) eat breakfast, (6) carpool to work.
- This routine gets the executive to work well prepared to make critical decisions.

3.2 Algorithms (Cont.)

- Suppose that the same steps are performed in a slightly different order: (1) Get out of bed, (2) take off pajamas, (3) get dressed, (4) take a shower, (5) eat breakfast, (6) carpool to work.
- In this case, our junior executive shows up for work soaking wet.
- Specifying the order in which statements are to be executed in a computer program is called program control.

3.3 Pseudocode

- Pseudocode is an artificial and informal language that helps you develop algorithms.
- Pseudocode is similar to everyday English; it's convenient and user friendly although it's not an actual computer programming language.
- Pseudocode programs are *not* executed on computers.
- Rather, they merely help you "think out" a program before attempting to write it in a programming language like C.
- Pseudocode consists purely of characters, so you may conveniently type pseudocode programs into a computer using an editor program.

3.3 Pseudocode (Cont.)

- A carefully prepared pseudocode program may be converted easily to a corresponding C program.
- Pseudocode consists only of action statements—those that are executed when the program has been converted from pseudocode to C and is run in C.
- Definitions are not executable statements.
- They’re simply messages to the compiler.

3.3 Pseudocode (Cont.)

- For example, the definition
  - `int i;`

  tells the compiler the type of variable `i` and instructs the compiler to reserve space in memory for the variable.
- But this definition does *not* cause any action—such as input, output, a calculation or a comparison—to occur when the program is executed.
- Some programmers choose to list each variable and briefly mention the purpose of each at the beginning of a pseudocode program.

3.4 Control Structures

- Normally, statements in a program are executed one after the other in the order in which they're written.
- This is called sequential execution.
- Various C statements we'll soon discuss enable you to specify that the next statement to be executed may be other than the next one in sequence.
- This is called transfer of control.
- During the 1960s, it became clear that the indiscriminate use of transfers of control was the root of a great deal of difficulty experienced by software development groups.

3.4 Control Structures (Cont.)

- The finger of blame was pointed at the `goto` statement that allows you to specify a transfer of control to one of many possible destinations in a program.
- The notion of so-called structured programming became almost synonymous with "`goto` elimination."
- Research had demonstrated that programs could be written without any `goto` statements.
- The challenge of the era was for programmers to shift their styles to "`goto`-less programming."

3.4 Control Structures (Cont.)

- The results were impressive, as software development groups reported reduced development times, more frequent on-time delivery of systems and more frequent within-budget completion of software projects.
- Programs produced with structured techniques were clearer, easier to debug and modify and more likely to be bug free in the first place.
- Research had demonstrated that all programs could be written in terms of only three control structures, namely the sequence structure, the selection structure and the repetition structure.

3.4 Control Structures (Cont.)

- The sequence structure is simple—unless directed otherwise, the computer executes C statements one after the other in the order in which they're written.
- The flowchart segment of Fig. 3.1 illustrates C's sequence structure.

Flowcharts

- A flowchart is a graphical representation of an algorithm or of a portion of an algorithm.
- Flowcharts are drawn using certain special-purpose symbols such as rectangles, diamonds, rounded rectangles, and small circles; these symbols are connected by arrows called flowlines.

3.4 Control Structures (Cont.)

- Like pseudocode, flowcharts are useful for developing and representing algorithms, although pseudocode is preferred by most programmers.
- Consider the flowchart for the sequence structure in Fig. 3.1.
- We use the rectangle symbol, also called the action symbol, to indicate any type of action including a calculation or an input/output operation.
- The flowlines in the figure indicate the order in which the actions are performed—first, `grade` is added to `total`, then `1` is added to `counter`.
- C allows us to have as many actions as we want in a sequence structure.

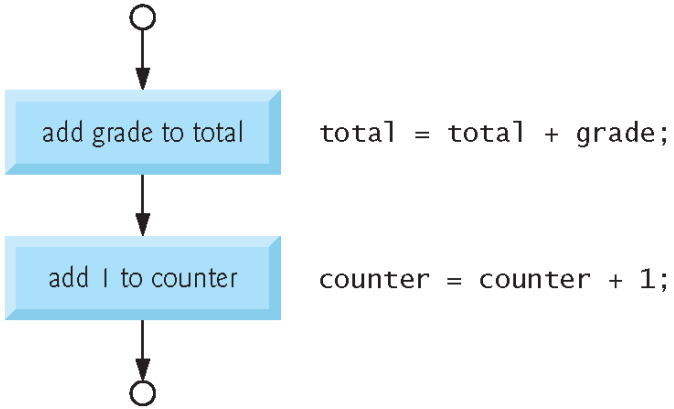

Fig. 3.1 | Flowcharting C's sequence structure.

3.4 Control Structures (Cont.)

- When drawing a flowchart that represents a complete algorithm, a rounded rectangle symbol containing the word “Begin” is the first symbol used in the flowchart; an oval symbol containing the word “End” is the last symbol used.
- When drawing only a portion of an algorithm as in Fig. 3.1, the rounded rectangle symbols are omitted in favor of using small circle symbols, also called connector symbols.
- Perhaps the most important flowcharting symbol is the diamond symbol, also called the decision symbol, which indicates that a decision is to be made.

3.4 Control Structures (Cont.)

Selection Statements in C

- C provides three types of selection structures in the form of statements.
- The `if` selection statement (Section 3.5) either selects (performs) an action if a condition is true or skips the action if the condition is false.
- The `if...else` selection statement (Section 3.6) performs an action if a condition is true and performs a different action if the condition is false.
- The `switch` selection statement (discussed in Chapter 4) performs one of many different actions depending on the value of an expression.

3.4 Control Structures (Cont.)

- The `if` statement is called a single-selection statement because it selects or ignores a single action.
- The `if...else` statement is called a double-selection statement because it selects between two different actions.
- The `switch` statement is called a multiple-selection statement because it selects among many different actions.

Repetition Statements in C

- C provides three types of repetition structures in the form of statements, namely `while` (Section 3.7), `do...while`, and `for` (both discussed in Chapter 4).
- That's all there is.

3.4 Control Structures (Cont.)

- C has only seven control statements: sequence, three types of selection and three types of repetition.
- Each C program is formed by combining as many of each type of control statement as is appropriate for the algorithm the program implements.
- As with the sequence structure of Fig. 3.1, we'll see that the flowchart representation of each control statement has two small circle symbols, one at the entry point to the control statement and one at the *exit point*.
- These single-entry/single-exit control statements make it easy to build clear programs.

3.4 Control Structures (Cont.)

- The control-statement flowchart segments can be attached to one another by connecting the exit point of one control statement to the entry point of the next.
- This is much like the way in which a child stacks building blocks, so we call this control-statement stacking.
- We'll learn that there's only one other way control statements may be connected—a method called control-statement nesting.
- Thus, any C program we'll ever need to build can be constructed from only seven different types of control statements combined in only two ways.
- This is the essence of simplicity.

3.5 The if Selection Statement

- Selection statements are used to choose among alternative courses of action.
- For example, suppose the passing grade on an exam is 60.
- The pseudocode statement
  - *If student's grade is greater than or equal to 60*
    Print "Passed"

  determines whether the condition "student's grade is greater than or equal to 60" is true or false.
- If the condition is true, then "Passed" is printed, and the next pseudocode statement in order is "performed" (remember that pseudocode isn't a real programming language).

3.5 The if Selection Statement (Cont.)

- If the condition is false, the printing is ignored, and the next pseudocode statement in order is performed.
- The second line of this selection structure is indented.
- Such indentation is optional, but it's highly recommended as it helps emphasize the inherent structure of structured programs.
- The C compiler ignores white-space characters such as blanks, tabs and newlines used for indentation and vertical spacing.

3.5 The `if` Selection Statement (Cont.)

- The preceding pseudocode *If statement may be written in C as*

```
if ( grade >= 60 ) {
   printf( "Passed\n" );
} /* end if */
```

- Notice that the C code corresponds closely to the pseudocode (of course you'll also need to declare the `int` variable `grade`).

3.5 The `if` Selection Statement (Cont.)

- The flowchart of Fig. 3.2 illustrates the single-selection `if` statement.
- This flowchart contains what is perhaps the most important flowcharting symbol—the diamond symbol, also called the decision symbol, which indicates that a decision is to be made.
- The decision symbol contains an expression, such as a condition, that can be either true or false.

3.5 The `if` Selection Statement (Cont.)

- The decision symbol has *two* flowlines emerging from it.
- One indicates the direction to take when the expression in the symbol is true and the other the direction to take when the expression is false.
- Decisions can be based on conditions containing relational or equality operators.
- In fact, a decision can be based on *any* expression—if the expression evaluates to *zero*, it's treated as false, and if it evaluates to *nonzero*, it's treated as true.

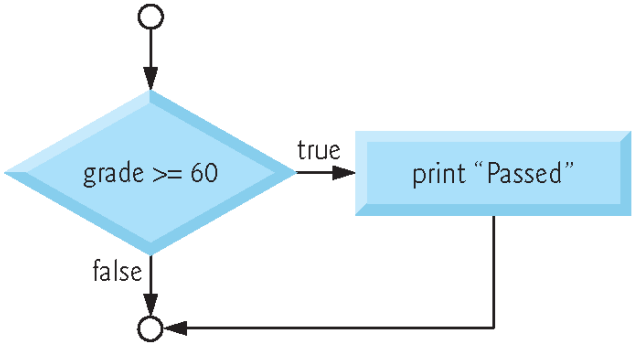

Fig. 3.2 | Flowcharting the single-selection `if` statement.

3.5 The `if` Selection Statement (Cont.)

- The `if` statement, too, is a single-entry/single-exit statement.
- We can envision seven bins, each containing only control-statement flowcharts of one of the seven types.
- These flowchart segments are empty—nothing is written in the rectangles and nothing is written in the diamonds.
- Your task, then, is assembling a program from as many of each type of control statement as the algorithm demands, combining them in only *two* possible ways (*stacking* or *nesting*), and then filling in the actions and decisions in a manner appropriate for the algorithm.

3.6 The `if...else` Selection Statement

- The `if...else` selection statement allows you to specify that *different* actions are to be performed when the condition is true and when it's false.
- For example, the pseudocode statement
  - *If student's grade is greater than or equal to 60*
    Print "Passed"
    else
    Print "Failed"

  prints *Passed* if the student's grade is greater than or equal to 60 and *Failed* if the student's grade is less than 60.
- In either case, after printing occurs, the next pseudocode statement in sequence is "performed." The body of the *else* is also indented.

Good Programming Practice 3.1

Indent both body statements of an `if...else` statement.

Good Programming Practice 3.2

If there are several levels of indentation, each level should be indented the same additional amount of space.

3.6 The `if...else` Selection Statement (Cont.)

- The preceding pseudocode *If…else* statement may be written in C as

```
if ( grade >= 60 ) {
   printf( "Passed\n" );
} /* end if */
else {
   printf( "Failed\n" );
} /* end else */
```

- The flowchart of Fig. 3.3 illustrates the flow of control in the `if...else` statement.
- Once again, besides small circles and arrows, the only symbols in the flowchart are rectangles for actions and a diamond for a decision.

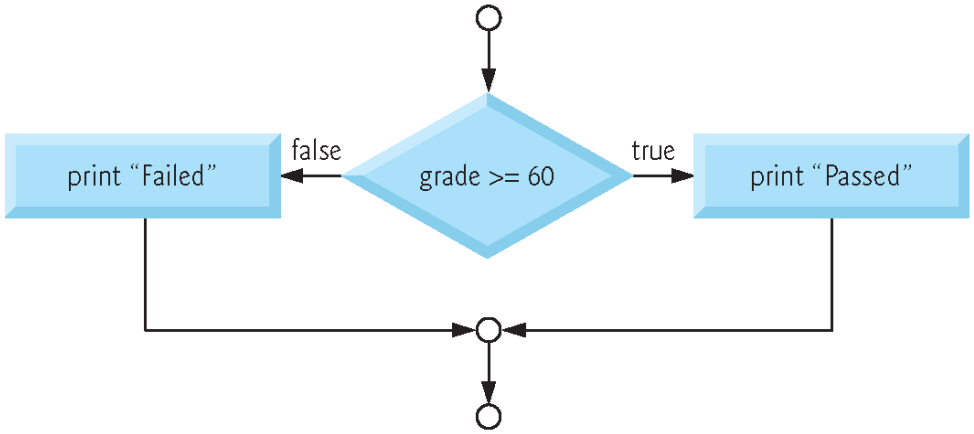

Fig. 3.3 | Flowcharting the double-selection `if...else` statement.

3.6 The `if...else` Selection Statement (Cont.)

- C provides the conditional operator (`?:`) which is closely related to the `if...else` statement.
- The conditional operator is C's only ternary operator—it takes *three* operands.
- These together with the conditional operator form a conditional expression.
- The first operand is a *condition*.
- The second operand is the value for the entire conditional expression if the condition is *true* and the operand is the value for the entire conditional expression if the condition is *false*.

3.6 The `if...else` Selection Statement (Cont.)

- For example, the `puts` statement
  - `puts( grade >= 60 ? "Passed" : "Failed" );`

  contains as its second argument a conditional expression that evaluates to the string `"Passed"` if the condition `grade >= 60` is true and to the string `"Failed"` if the condition is false.
- The `puts` statement performs in essentially the same way as the preceding `if...else` statement.

3.6 The `if...else` Selection Statement (Cont.)

- The second and third operands in a conditional expression can also be actions to be executed.
- For example, the conditional expression

```
grade >= 60 ? puts( "Passed" ) : puts( "Failed" );
```

- is read, "If `grade` is greater than or equal to `60` then `puts("Passed")`, otherwise `puts( "Failed" )`." This, too, is comparable to the preceding `if...else` statement.
- We'll see that conditional operators can be used in some places where `if...else` statements cannot.

3.6 The `if...else` Selection Statement (Cont.)

Nested if...else Statements

- Nested `if...else` statements test for multiple cases by placing `if...else` statements inside `if...else` statements.
- For example, the following pseudocode statement will print `A` for exam grades greater than or equal to `90`, `B` for grades greater than or equal to `80` (but less than `90`), `C` for grades greater than or equal to `70` (but less than `80`), `D` for grades greater than or equal to `60` (but less than `70`) and `F` for all other grades.

3.6 The `if...else` Selection Statement (Cont.)

```
If student's grade is greater than or equal to 90
   Print "A"
else
   If student's grade is greater than or equal to 80
      Print "B"
   else
      If student's grade is greater than or equal to 70
         Print "C"
      else
         If student's grade is greater than or equal to 60
            Print "D"
         else
            Print "F"
```

3.6 The if...else Selection Statement (Cont.)

- This pseudocode may be written in C as

```
if ( grade >= 90 )
   puts( "A" );
else
   if ( grade >= 80 )
      puts("B");
   else
      if ( grade >= 70 )
         puts("C");
      else
         if ( grade >= 60 )
            puts( "D" );
         else
            puts( "F" );
```

3.6 The `if...else` Selection Statement (Cont.)

- If the variable `grade` is greater than or equal to 90, all four conditions will be true, but only the `puts` statement after the first test will be executed.
- After that `puts` is executed, the `else` part of the "outer" `if...else` statement is skipped.

3.6 The if...else Selection Statement (Cont.)

- You may prefer to write the preceding `if` statement as

```
if ( grade >= 90 )
   puts( "A" );
else if ( grade >= 80 )
   puts( "B" );
else if ( grade >= 70 )
   puts( "C" );
else if ( grade >= 60 )
   puts( "D" );
else
   puts( "F" );
```

3.6 The `if...else` Selection Statement (Cont.)

- As far as the C compiler is concerned, both forms are equivalent.
- The latter form is popular because it avoids the deep indentation of the code to the right.
- The `if` selection statement expects only one statement in its body—if you have only one statement in the if's body, you do not need the enclose it in braces.
- To include several statements in the body of an `if`, you must enclose the set of statements in braces (`{` and `}`).
- A set of statements contained within a pair of braces is called a compound statement or a block.

Software Engineering Observation 3.1

A compound statement can be placed anywhere in a program that a single statement can be placed.

3.6 The `if...else` Selection Statement (Cont.)

- The following example includes a compound statement in the `else` part of an `if...else` statement.

```
if ( grade >= 60 ) {
   puts( "Passed. " );
} /* end if */
else {
   puts( "Failed. " );
   puts( "You must take this course again.
" );
} /* end else */
```

3.6 The `if...else` Selection Statement (Cont.)

- In this case, if grade is less than `60`, the program executes both `puts` statements in the body of the `else` and prints
  - ```
    Failed.
    You must take this course again.
    ```
- The braces surrounding the two statements in the `else` are important. Without them, the statement

  ```
  puts( "You must take this course again." );
  ```

  would be outside the body of the `else` part of the `if` and would execute regardless of whether the grade was less than 60.
  ```

3.6 The `if...else` Selection Statement (Cont.)

- A syntax error is caught by the compiler.
- A logic error has its effect at execution time.
- A fatal logic error causes a program to fail and terminate prematurely.
- A nonfatal logic error allows a program to continue executing but to produce incorrect results.
- Just as a compound statement can be placed anywhere a single statement can be placed, it's also possible to have no statement at all, i.e., the empty statement.
  - The empty statement is represented by placing a semicolon (;) where a statement would normally be.

Common Programming Error 3.1

Placing a semicolon after the condition in an `if` statement as in `if ( grade >= 60 );` leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if` statements.

Error-Prevention Tip 3.1

Typing the beginning and ending braces of compound statements before typing the individual statements within the braces helps avoid omitting one or both of the braces, preventing syntax errors and logic errors (where both braces are indeed required).

3.7 The `while` Repetition Statement

- A repetition statement (also called an iteration statement) allows you to specify that an action is to be repeated while some condition remains true.
- The pseudocode statement
  - *While there are more items on my shopping list*
    Purchase next item and cross it off my list

  describes the repetition that occurs during a shopping trip.
- The condition, "there are more items on my shopping list" may be true or false.
- If it's true, then the action, "Purchase next item and cross it off my list" is performed.
- This action will be performed repeatedly while the condition remains true.

3.7 The while Repetition Statement (Cont.)

- The statement(s) contained in the *while* repetition statement constitute the body of the while.
- The *while* statement body may be a single statement or a compound statement.
- Eventually, the condition will become false (when the last item on the shopping list has been purchased and crossed off the list).
- At this point, the repetition terminates, and the first pseudocode statement *after* the repetition structure is executed.

Common Programming Error 3.2

Not providing in the body of a `while` statement an action that eventually causes the condition in the `while` to become false. Normally, such a repetition structure will never terminate—an error called an "infinite loop."

Common Programming Error 3.3

Spelling the keyword `while` with an uppercase `W`, as in `While` (remember that C is a case-sensitive language).

3.7 The `while` Repetition Statement (Cont.)

- As an example of a `while` statement, consider a program segment designed to find the first power of 3 larger than 100.
- Suppose the integer variable `product` has been initialized to 3.
- When the following `while` repetition statement finishes executing, `product` will contain the desired answer:

```
product = 3;
while ( product <= 100 ) {
   product = 3 * product;
} /* end while */
```

- The flowchart of Fig. 3.4 illustrates the flow of control in the `while` repetition statement.

3.7 The `while` Repetition Statement (Cont.)

- Once again, note that (besides small circles and arrows) the flowchart contains only a rectangle symbol and a diamond symbol.
- The flowchart clearly shows the repetition.
- The flowline emerging from the rectangle wraps back to the decision, which is tested each time through the loop until the decision eventually becomes false.
- At this point, the `while` statement is exited and control passes to the next statement in the program.

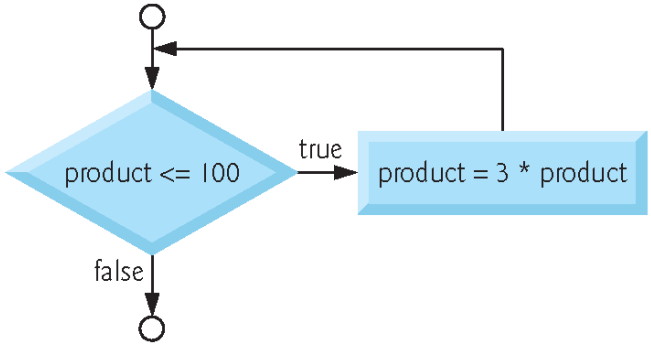

Fig. 3.4 | Flowcharting the `while` repetition statement.

3.7 The `while` Repetition Statement (Cont.)

- When the `while` statement is entered, the value of `product` is 3.
- The variable `product` is repeatedly multiplied by 3, taking on the values 9, 27 and 81 successively.
- When `product` becomes 243, the condition in the `while` statement, `product <= 100`, becomes false.
- This terminates the repetition, and the final value of `product` is 243.
- Program execution continues with the next statement after the `while`.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition

- To illustrate how algorithms are developed, we solve several variations of a class-averaging problem.
- Consider the following problem statement:
  - *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*
- The class average is equal to the sum of the grades divided by the number of students.
- The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation, and print the result.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition (Cont.)

- Let's use pseudocode to list the actions to execute and specify the order in which these actions should execute.
- We use counter-controlled repetition to input the grades one at a time.
- This technique uses a variable called a counter to specify the number of times a set of statements should execute.
- In this example, repetition terminates when the counter exceeds 10.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition (Cont.)

- In this section we simply present the pseudocode algorithm (Fig. 3.5) and the corresponding C program (Fig. 3.6).
- In the next section we show how pseudocode algorithms are *developed*.
- Counter-controlled repetition is often called definite repetition because the number of repetitions is known *before* the loop begins executing.

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5      Input the next grade
6      Add the grade into the total
7      Add one to the grade counter
8
9  Set the class average to the total divided by ten
10 Print the class average
```

Fig. 3.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class-average problem.

```c
// Fig. 3.6: fig03_06.c
// Class average program with counter-controlled repetition.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   unsigned int counter; // number of grade to be entered next
   int grade; // grade value
   int total; // sum of grades entered by user
   int average; // average of grades

   // initialization phase
   total = 0; // initialize total
   counter = 1; // initialize loop counter

   // processing phase
   while ( counter <= 10 ) { // loop 10 times
      printf( "%s", "Enter grade: " ); // prompt for input
      scanf( "%d", &grade ); // read grade from user
      total = total + grade; // add grade to total
      counter = counter + 1; // increment counter
   } // end while
```

Fig. 3.6 | Class-average problem with counter-controlled repetition. (Part 1 of 2.)

```
24
25      // termination phase
26      average = total / 10; // integer division
27
28      printf( "Class average is %d\n", average ); // display result
29   } // end function main
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

Fig. 3.6 | Class-average problem with counter-controlled repetition. (Part 2 of 2.)

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition (Cont.)

- The algorithm mentions a total and a counter.
- A total is a variable used to accumulate the sum of a series of values.
- A counter is a variable (line 8) used to count—in this case, to count the number of grades entered.
- Because the counter variable is used to count from 1 to 10 in this program (all positive values), we declared the variable as an `unsigned int`, which can store only non-negative values (that is, 0 and higher).

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition (Cont.)

- Variables used to store totals should normally be initialized to zero before being used in a program; otherwise the sum would include the previous value stored in the total's memory location.
- Counter variables are normally initialized to zero or one, depending on their use (we'll present examples of each).
- An uninitialized variable contains a "garbage" value—the value last stored in the memory location reserved for that variable.

Common Programming Error 3.4

If a counter or total isn't initialized, the results of your program will probably be incorrect. This is an example of a logic error.

Error-Prevention Tip 3.2

Initialize all counters and totals.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition (Cont.)

- The averaging calculation in the program produced an integer result of 81.
- Actually, the sum of the grades in this example is 817, which when divided by 10 should yield 81.7, i.e., a number with a *decimal point*.
- We'll see how to deal with such numbers (called *floating-point numbers*) in the next section.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition

- Let's generalize the class-average problem.
- Consider the following problem:
  - *Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.*
- In the first class-average example, the number of grades (10) was known in advance.
- In this example, the program must process an arbitrary number of grades.
- How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- One way to solve this problem is to use a special value called a sentinel value (also called a signal value, a dummy value, or a flag value) to indicate "end of data entry."
- The user types in grades until all legitimate grades have been entered.
- The user then types the sentinel value to indicate "the last grade has been entered."
- Sentinel-controlled repetition is often called indefinite repetition because the number of repetitions isn't known before the loop begins executing.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value.
- Because grades on a quiz are normally nonnegative integers, –1 is an acceptable sentinel value for this problem.
- Thus, a run of the class-average program might process a stream of inputs such as 95, 96, 75, 74, 89 and –1.
- The program would then compute and print the class average for the grades 95, 96, 75, 74, and 89 (–1 is the sentinel value, so it should not enter into the averaging calculation).

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

Top-Down, Stepwise Refinement

- We approach the class-average program with a technique called top-down, stepwise refinement, a technique that is essential to the development of well-structured programs.
- We begin with a pseudocode representation of the top:
  - *Determine the class average for the quiz*
- The top is a single statement that conveys the program's overall function.
- As such, the top is, in effect, a complete representation of a program.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Unfortunately, the top rarely conveys a sufficient amount of detail for writing the C program.
- So we now begin the *refinement* process.
- We divide the top into a series of smaller tasks and list these in the order in which they need to be performed.
- This results in the following first refinement.
  - *Initialize variables*
    Input, sum, and count the quiz grades
    Calculate and print the class average
- Here, only the sequence structure has been used—the steps listed are to be executed in order, one after the other.

Software Engineering Observation 3.2

Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

Second Refinement

- To proceed to the next level of refinement, i.e., the second refinement, we commit to specific variables.
- We need a running total of the numbers, a count of how many numbers have been processed, a variable to receive the value of each grade as it's input and a variable to hold the calculated average.
- The pseudocode statement
  - *Initialize variables*
- may be refined as follows:
  - *Initialize total to zero*
    Initialize counter to zero

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Only the total and counter need to be initialized; the variables average and grade (for the calculated average and the user input, respectively) need not be initialized because their values will be written over by the process of destructive read-in discussed in Chapter 2.
- The pseudocode statement
  - *Input, sum, and count the quiz grades*

  requires a *repetition structure* that successively inputs each grade.
- Because we do not know in advance how many grades are to be processed, we'll use sentinel-controlled repetition.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- The user will enter legitimate grades in one at a time.
- After the last legitimate grade is typed, the user will type the sentinel value.
- The program will test for this value after each grade is input and will terminate the loop when the sentinel is entered.
- The refinement of the preceding pseudocode statement is then
  - *Input the first grade*
    While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Notice that in pseudocode, we do not use braces around the set of statements that form the body of the *while statement*.
- We simply indent all these statements under the *while to show that they all belong to the while*.
- Again, pseudocode is an informal program development aid.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- The pseudocode statement
  - *Calculate and print the class average*

  may be refined as follows:
  - *If the counter is not equal to zero*
    Set the average to the total divided by the counter
    Print the average
    else
    Print "No grades were entered"
- Notice that we're being careful here to test for the possibility of *division by zero*—a fatal error that if undetected would cause the program to fail (often called "crashing").
- The complete second refinement is shown in Fig. 3.7.

Common Programming Error 3.5

An attempt to divide by zero causes a fatal error.

Good Programming Practice 3.3

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as printing an error message) rather than allowing the fatal error to occur.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- In Fig. 3.5 and Fig. 3.7, we include some completely blank lines in the pseudocode for readability.
- Actually, the blank lines separate these programs into their various phases.

```
1   Initialize total to zero
2   Initialize counter to zero
3
4   Input the first grade
5   While the user has not as yet entered the sentinel
6       Add this grade into the running total
7       Add one to the grade counter
8       Input the next grade (possibly the sentinel)
9
10  If the counter is not equal to zero
11      Set the average to the total divided by the counter
12      Print the average
13  else
14      Print "No grades were entered"
```

Fig. 3.7 | Pseudocode algorithm that uses sentinel-controlled repetition to solve the class-average problem.

Software Engineering Observation 3.3

Many programs can be divided logically into three phases: an initialization phase that initializes the program variables; a processing phase that inputs data values and adjusts program variables accordingly; and a termination phase that calculates and prints the final results.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- The pseudocode algorithm in Fig. 3.7 solves the more general class-averaging problem.
- This algorithm was developed after only two levels of refinement.
- Sometimes more levels are necessary.

Software Engineering Observation 3.4

You terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C. Implementing the C program is then normally straightforward.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- The C program and a sample execution are shown in Fig. 3.8.
- Although only integer grades are entered, the averaging calculation is likely to produce a number with a decimal point.
- The type `int` cannot represent such a number.
- The program introduces the data type `float` to handle numbers with decimal points (called floating-point numbers) and introduces a special operator called a cast operator to handle the averaging calculation.
- These features are explained after the program is presented.

```
1  // Fig. 3.8: fig03_08.c
2  // Class-average program with sentinel-controlled repetition.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     unsigned int counter; // number of grades entered
9     int grade; // grade value
10    int total; // sum of grades
11
12    float average; // number with decimal point for average
13
14    // initialization phase
15    total = 0; // initialize total
16    counter = 0; // initialize loop counter
17
18    // processing phase
19    // get first grade from user
20    printf( "%s", "Enter grade, -1 to end: " ); // prompt for input
21    scanf( "%d", &grade ); // read grade from user
22
```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 1 of 3.)

```
23      // loop while sentinel value not yet read from user
24      while ( grade != -1 ) {
25         total = total + grade; // add grade to total
26         counter = counter + 1; // increment counter
27
28         // get next grade from user
29         printf( "%s", "Enter grade, -1 to end: " ); // prompt for input
30         scanf("%d", &grade); // read next grade
31      } // end while
32
33      // termination phase
34      // if user entered at least one grade
35      if ( counter != 0 ) {
36
37         // calculate average of all grades entered
38         average = ( float ) total / counter; // avoid truncation
39
40         // display average with two digits of precision
41         printf( "Class average is %.2f\n", average );
42      } // end if
43      else { // if no grades were entered, output message
44         puts( "No grades were entered" );
45      } // end else
46   } // end function main
```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 2 of 3.)

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

Fig. 3.8 | Class-average program with sentinel-controlled repetition. (Part 3 of 3.)

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Notice the compound statement in the `while` loop (line 24) in Fig. 3.8 Once again, the braces are *necessary* to ensure that all four statements are executed within the loop.
- Without the braces, the last three statements in the body of the loop would fall outside the loop, causing the computer to interpret this code incorrectly as follows.

```
while ( grade != -1 )
   total = total + grade; /* add grade to total */
counter = counter + 1; /* increment counter */
printf( "Enter grade, -1 to end: " ); /* prompt for input */
scanf( "%d", &grade ); /* read next grade */
```

- This would cause an *infinite loop* if the user did not input `-1` for the first grade.

Good Programming Practice 3.4

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

Converting Between Types Explicitly and Implicitly

- Averages do not always evaluate to integer values.
- Often, an average is a value such as 7.2 or –93.5 that contains a fractional part.
- These values are referred to as floating-point numbers and can be represented by the data type `float`.
- The variable `average` is defined to be of type `float` (line 12) to capture the fractional result of our calculation.
- However, the result of the calculation `total / counter` is an integer because `total` and `counter` are both integer variables.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Dividing two integers results in integer division in which any fractional part of the calculation is truncated (i.e., lost).
- Because the calculation is performed *first*, the fractional part is lost *before* the result is assigned to `average`.
- To produce a floating-point calculation with integer values, we must create temporary values that are floating-point numbers.
- C provides the unary cast operator to accomplish this task.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Line 38
  - `average = ( float ) total / counter;`
- includes the cast operator `(float)`, which creates a temporary floating-point copy of its operand, `total`.
- The value stored in `total` is still an integer.
- Using a cast operator in this manner is called explicit conversion.
- The calculation now consists of a floating-point value (the temporary `float` version of `total`) divided by the `unsigned int` value stored in `counter`.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- C evaluates arithmetic expressions only in which the data types of the operands are *identical*.
- To ensure that the operands are of the *same* type, the compiler performs an operation called implicit conversion on selected operands.
- For example, in an expression containing the data types `unsigned int` and `float`, copies of `unsigned int` operands are made and converted to `float`.
- In our example, after a copy of `counter` is made and converted to `float`, the calculation is performed and the result of the floating-point division is assigned to `average`.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- C provides a set of rules for convertion of operands of different types.
- We discuss this further in Chapter 5.
- Cast operators are available for *most* data types—they're formed by placing parentheses around a type name.
- Each cast operator is a unary operator, i.e., an operator that takes only one operand.
- C also supports unary versions of the plus (+) and minus (-) operators, so you can write expressions such as `-7` or `+5`.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Cast operators associate from right to left and have the same precedence as other unary operators such as unary `+` and unary `-`.
- This precedence is one level higher than that of the multiplicative operators `*`, `/` and `%`.

Formatting Floating-Point Numbers

- Figure 3.8 uses the `printf` conversion specifier `%.2f` (line 41) to print the value of `average`.
- The `f` specifies that a floating-point value will be printed.
- The `.2` is the precision with which the value will be displayed—with 2 digits to the right of the decimal point.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- If the `%f` conversion specifier is used (without specifying the precision), the default precision of 6 is used—exactly as if the conversion specifier `%.6f` had been used.
- When floating-point values are printed with precision, the printed value is rounded to the indicated number of decimal positions.
- The value in memory is unaltered.
- When the following statements are executed, the values 3.45 and 3.4 are printed.

```
printf( "%.2f\n", 3.446 ); /* prints 3.45 */
printf( "%.1f\n", 3.446 ); /* prints 3.4  */
```

Common Programming Error 3.6

Using precision in a conversion specification in the format control string of a `scanf` statement is wrong. Precisions are used only in `printf` conversion specifications.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

Notes on Floating-Point Numbers

- Although floating-point numbers are not always "100% precise," they have numerous applications.
- For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits.
- When we view the temperature on a thermometer and read it as 98.599947321064 3, it may actually be 98.599947321064 3.
- The point here is that calling this number simply 98.6 is fine for most applications.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Another way floating-point numbers develop is through division.
- When we divide 10 by 3, the result is 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*.

Common Programming Error 3.7

Using floating-point numbers in a manner that assumes they're represented precisely can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.

Error-Prevention Tip 3.3

Do not compare floating-point values for equality.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements

- Let's work another complete problem.
- We'll once again formulate the algorithm using pseudocode and top-down, stepwise refinement, and write a corresponding C program.
- We've seen that control statements may be stacked on top of one another (in sequence) just as a child stacks building blocks.
- In this case study we'll see the only other structured way control statements may be connected in C, namely through nesting of one control statement within another.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- Consider the following problem statement:
  - A college offers a course that prepares students for the state licensing exam for real estate brokers. Last year, 10 of the students who completed this course took the licensing examination. Naturally, the college wants to know how well its students did on the exam. You've been asked to write a program to summarize the results. You've been given a list of these 10 students. Next to each name a 1 is written if the student passed the exam and a 2 if the student failed.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- Your program should analyze the results of the exam as follows:
  - Input each test result (i.e., a 1 or a 2). Display the prompting message “Enter result” each time the program requests another test result.
  - Count the number of test results of each type.
  - Display a summary of the test results indicating the number of students who passed and the number who failed.
  - If more than eight students passed the exam, print the message “Bonus to instructor!”

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- After reading the problem statement carefully, we make the following observations:
  - The program must process 10 test results. A counter-controlled loop will be used.
  - Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine whether the number is a 1 or a 2. We test for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2. (An exercise at the end of the chapter considers the consequences of this assumption.)
  - Two counters are used—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
  - After the program has processed all the results, it must decide whether more than 8 students passed the exam.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- Let's proceed with top-down, stepwise refinement.
- We begin with a pseudocode representation of the top:
  - *Analyze exam results and decide if instructor should receive a bonus*
- Once again, it's important to emphasize that the top is a complete representation of the program, but several refinements are likely to be needed before the pseudocode can be naturally evolved into a C program.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- Our first refinement is
  - *Initialize variables*
    Input the ten quiz grades and count passes and failures
    Print a summary of the exam results and decide if instructor should receive a bonus
- Here, too, even though we have a complete representation of the entire program, further refinement is necessary.
- We now commit to specific variables.
- Counters are needed to record the passes and failures, a counter will be used to control the looping process, and a variable is needed to store the user input.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- The pseudocode statement
  - *Initialize variables*

  may be refined as follows:
  - *Initialize passes to zero*
    Initialize failures to zero
    Initialize student to one
- Notice that only the counters and totals are initialized.
- The pseudocode statement
  - *Input the ten quiz grades and count passes and failures*

  requires a loop that successively inputs the result of each exam.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- Here it's known in advance that there are precisely ten exam results, so counter-controlled looping is appropriate.
- Inside the loop (i.e., nested within the loop) a double-selection statement will determine whether each exam result is a pass or a failure and will increment the appropriate counters accordingly.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- The refinement of the preceding pseudocode statement is then
  - *While student counter is less than or equal to ten*
    Input the next exam result

    If the student passed
        Add one to passes
    else
        Add one to failures

    Add one to student counter
- Notice the use blank lines to set off the *If…else* to improve program readability.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- The pseudocode statement
  - *Print a summary of the exam results and decide if instructor should receive a bonus*
- may be refined as follows:
  - *Print the number of passes*
    Print the number of failures
    If more than eight students passed
        Print "Bonus to instructor!"
- The complete second refinement appears in Fig. 3.9.
- We use blank lines to set off the `while` statement for program readability.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (Cont.)

- This pseudocode is now sufficiently refined for conversion to C.
- The C program and two sample executions are shown in Fig. 3.10.
- We've taken advantage of a feature of C that allows initialization to be incorporated into definitions.
- Such initialization occurs at compile time.
- Also, notice that when you output an `unsigned int` you use the `%u` conversion specifier.

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student to one
4
5  While student counter is less than or equal to ten
6          Input the next exam result
7
8          If the student passed
9              Add one to passes
10         else
11             Add one to failures
12
13         Add one to student counter
14
15 Print the number of passes
16 Print the number of failures
17 If more than eight students passed
18     Print "Bonus to instructor!"
```

Fig. 3.9 | Pseudocode for examination-results problem.

Software Engineering Observation 3.5

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C program is normally straightforward.

Software Engineering Observation 3.6

Many programmers write programs without ever using program development tools such as pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs.

```
1  // Fig. 3.10: fig03_10.c
2  // Analysis of examination results.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     // initialize variables in definitions
9     unsigned int passes = 0; // number of passes
10    unsigned int failures = 0; // number of failures
11    unsigned int student = 1; // student counter
12    int result; // one exam result
13
14    // process 10 students using counter-controlled loop
15    while ( student <= 10 ) {
16
17       // prompt user for input and obtain value from user
18       printf( "%s", "Enter result ( 1=pass,2=fail ): " );
19       scanf( "%d", &result );
20
21       // if result 1, increment passes
22       if ( result == 1 ) {
23          passes = passes + 1;
24       } // end if
```

Fig. 3.10 | Analysis of examination results. (Part 1 of 4.)

```
25          else { // otherwise, increment failures
26             failures = failures + 1;
27          } // end else
28
29          student = student + 1; // increment student counter
30       } // end while
31
32       // termination phase; display number of passes and failures
33       printf( "Passed %u\n", passes );
34       printf( "Failed %u\n", failures );
35
36       // if more than eight students passed, print "Bonus to instructor!"
37       if ( passes > 8 ) {
38          puts( "Bonus to instructor!" );
39       } // end if
40    } // end function main
```

Fig. 3.10 | Analysis of examination results. (Part 2 of 4.)

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Passed 6
Failed 4
```

Fig. 3.10 | Analysis of examination results. (Part 3 of 4.)

```
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 2
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Enter Result (1=pass,2=fail): 1
Passed 9
Failed 1
Bonus to instructor!
```

Fig. 3.10 | Analysis of examination results. (Part 4 of 4.)

3.11 Assignment Operators

- C provides several assignment operators for abbreviating assignment expressions.
- For example, the statement
  - `c = c + 3;`
- can be abbreviated with the addition assignment operator **+=** as
  - `c += 3;`
- The **+=** operator adds the value of the expression on the right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator.

3.11 Assignment Operators (Cont.)

- Any statement of the form
  - *variable = variable operator expression;*
- where *operator is one of the binary operators* `+`, `-`, `*`, `/` *or* `%` *(or others we'll discuss in Chapter 10), can be written in the form*
  - *variable operator= expression;*
- Thus the assignment `c += 3` adds `3` to `c`.
- Figure 3.11 shows the arithmetic assignment operators, sample expressions using these operators and explanations.

Assignment operator	Sample expression	Explanation	Assigns
Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`			
`+=`	`c += 7`	`c = c + 7`	10 to c
`-=`	`d -= 4`	`d = d - 4`	1 to d
`*=`	`e *= 5`	`e = e * 5`	20 to e
`/=`	`f /= 3`	`f = f / 3`	2 to f
`%=`	`g %= 9`	`g = g % 9`	3 to g

Fig. 3.11 | Arithmetic assignment operators.

3.12 Increment and Decrement Operators

- C also provides the unary increment operator, `++`, and the unary decrement operator, `--`, which are summarized in Fig. 3.12.
- If a variable `c` is to be incremented by 1, the increment operator `++` can be used rather than the expressions `c = c + 1` or `c += 1`.
- If increment or decrement operators are placed before a variable (i.e., prefixed), they're referred to as the preincrement or predecrement operators, respectively.
- If increment or decrement operators are placed after a variable (i.e., postfixed), they're referred to as the postincrement or postdecrement operators, respectively.

3.12 Increment and Decrement Operators (Cont.)

- Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears.
- Postincrementing (postdecrementing) the variable causes the current value of the variable to be used in the expression in which it appears, then the variable value is incremented (decremented) by 1.

Operator	Sample expression	Explanation
++	++a	Increment a by 1, then use the new value of a in the expression in which a resides.
++	a++	Use the current value of a in the expression in which a resides, then increment a by 1.
--	--b	Decrement b by 1, then use the new value of b in the expression in which b resides.
--	b--	Use the current value of b in the expression in which b resides, then decrement b by 1.

Fig. 3.12 | Increment and decrement operators

3.12 Increment and Decrement Operators (Cont.)

- Figure 3.13 demonstrates the difference between the preincrementing and the postincrementing versions of the ++ operator.
- Postincrementing the variable `c` causes it to be incremented after it's used in the `printf` statement.
- Preincrementing the variable `c` causes it to be incremented *before* it's used in the `printf` statement.

```c
// Fig. 3.13: fig03_13.c
// Preincrementing and postincrementing.
#include <stdio.h>

// function main begins program execution
int main( void )
{
   int c; // define variable

   // demonstrate postincrement
   c = 5; // assign 5 to c
   printf( "%d\n", c ); // print 5
   printf( "%d\n", c++ ); // print 5 then postincrement
   printf( "%d\n\n", c ); // print 6

   // demonstrate preincrement
   c = 5; // assign 5 to c
   printf( "%d\n", c ); // print 5
   printf( "%d\n", ++c ); // preincrement then print 6
   printf( "%d\n", c ); // print 6
} // end function main
```

Fig. 3.13 | Preincrementing and postincrementing. (Part 1 of 2.)

```
5
5
6

5
6
6
```

Fig. 3.13 | Preincrementing and postincrementing. (Part 2 of 2.)

3.12 Increment and Decrement Operators (Cont.)

- The program displays the value of `c` before and after the `++` operator is used.
- The decrement operator (`--`) works similarly.

Good Programming Practice 3.5

Unary operators should be placed directly next to their operands with no intervening spaces.

3.12 Increment and Decrement Operators (Cont.)

- The three assignment statements in Fig. 3.10
  - ```
    passes = passes + 1;
    failures = failures + 1;
    student = student + 1;
    ```

  can be written more concisely with *assignment operators* as
  - ```
    passes += 1;
    failures += 1;
    student += 1;
    ```

  with *preincrement operators* as
  - ```
    ++passes;
    ++failures;
    ++student;
    ```

  or with *postincrement operators* as
  - ```
    passes++;
    failures++;
    student++;
    ```

3.12 Increment and Decrement Operators (Cont.)

- It’s important to note here that when incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the *same* effect.

3.12 Increment and Decrement Operators (Cont.)

- It's only when a variable appears in the context of a larger expression that preincrementing and postincrementing have *different* effects (and similarly for predecrementing and postdecrementing).
- Of the expressions we've studied thus far, only a simple variable name may be used as the operand of an increment or decrement operator.

Common Programming Error 3.8

Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error, e.g., writing `++(x + 1)`.

Error-Prevention Tip 3.4

C generally does not specify the order in which an operator's operands will be evaluated (although we'll see exceptions to this for a few operators in Chapter 4). Therefore you should use increment or decrement operators only in statements in which one variable is incremented or decremented by itself.

3.12 Increment and Decrement Operators (Cont.)

- Figure 3.14 lists the precedence and associativity of the operators introduced to this point.
- The operators are shown top to bottom in decreasing order of precedence.
- The second column indicates the associativity of the operators at each level of precedence.
- Notice that the conditional operator (`?:`), the unary operators increment (`++`), decrement (`--`), plus (`+`), minus (`-`) and casts, and the assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=` associate from right to left.
- The third column names the various groups of operators.
- All other operators in Fig. 3.14 associate from left to right.

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

Fig. 3.14 | Precedence and associativity of the operators encountered so far in the text.

3.13 Secure C Programming

Arithmetic Overflow

- Figure 2.5 presented an addition program which calculated the sum of two int values (line 18) with the statement

```
sum = integer1 + integer2; // assign total to sum
```

- Even this simple statement has a potential problem—adding the integers could result in a value that's *too large* to store in an int variable.
- This is known as arithmetic overflow and can cause undefined behavior, possibly leaving a system open to attack.

3.13 Secure C Programming (Cont.)

- The maximum and minimum values that can be stored in an `int` variable are represented by the constants `INT_MAX` and `INT_MIN`, respectively, which are defined in the header `<limits.h>`.
- You can see your platform's values for these constants by opening the header `<limits.h>` in a text editor.
- It's considered a good practice to ensure that before you perform arithmetic calculations like the one in line 18 of Fig. 2.5, they will not overflow.
- The code for doing this is shown on the CERT website `www.securecoding.cert.org`—just search for guideline "INT32-C."
- The code uses the `&&` (logical AND) and `||` (logical OR) operators, which are introduced in the Chapter 4.

3.13 Secure C Programming (Cont.)

Unsigned Integers

- In Fig. 3.6, line 8 declared as an `unsigned int` the variable counter because it's used to count only *non-negative values*.
- In general, counters that should store only non-negative values should be declared with `unsigned` before the integer type.
- Variables of `unsigned` types can represent values from 0 to approximately twice the positive range of the corresponding signed integer types.
- You can determine your platform's maximum unsigned `int` value with the constant `UINT_MAX` from `<limits.h>`.

3.13 Secure C Programming (Cont.)

- The class-averaging program in Fig. 3.6 could have declared as `unsigned int` the variables `grade`, `total` and `average`.
- Grades are normally values from 0 to 100, so the total and average should each be greater than or equal to 0.
- We declared those variables as `int`s because we can't control what the user actually enters—the user could enter negative values.
- Worse yet, the user could enter a value that's not even a number. (We'll show how to deal with such inputs later in the book.)

3.13 Secure C Programming (Cont.)

- Sometimes sentinel-controlled loops use invalid values to terminate a loop.
- For example, the class-averaging program of Fig. 3.8 terminates the loop when the user enters the sentinel `-1` (an invalid grade), so it would be improper to declare variable grade as an `unsigned int`.
- As you'll see, the end-of-file (`EOF`) indicator—which is introduced in the next chapter and is often used to terminate sentinel-controlled loops—is also a negative number.

3.13 Secure C Programming (Cont.)

scanf_s and printf_s

- The C11 standard's Annex K introduces more secure versions of `printf` and `scanf` called `printf_s` and `scanf_s`. Annex K is designated as optional, so not every C vendor will implement it.
- Microsoft implemented its own versions of `printf_s` and `scanf_s` prior to the publication of the C11 standard and immediately began issuing warnings for every `scanf` call.
- The warnings say that `scanf` is deprecated—it should no longer be used—and that you should consider using `scanf_s` instead.

3.13 Secure C Programming (Cont.)

- There are two ways to eliminate Visual C++'s `scanf` warnings—you can use `scanf_s` instead of `scanf` or you can disable these warnings.
- For the input statements we've used so far, Visual C++ users can simply replace `scanf` with `scanf_s`. You can disable the warning messages in Visual C++ as follows:

1. Type *Alt F7* to display the Property Pages dialog for your project.
2. In the left column, expand Configuration Properties > C/C++ and select Preprocessor.
3. In the right column, at the end of the value for Preprocessor Definitions, insert
   ```
   ;_CRT_SECURE_NO_WARNINGS
   ```
4. Click OK to save the changes.