Good Programming Practice 3.5

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.
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.
Initialize total to zero
Initialize counter to zero

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)

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”

**Fig. 3.7**  Pseudocode algorithm that uses sentinel-controlled repetition to solve the class average problem.
Software Engineering Observation 3.4
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.
Formulating Algorithms with Top-Down, Stepwise Refinement
Case Study 2: Sentinel-Controlled Repetition

- 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 3.5

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.
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 decimal 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 in detail after the program is presented.
/* Fig. 3.8: fig03_08.c */
Class average program with sentinel-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
    int counter; /* number of grades entered */
    int grade; /* grade value */
    int total; /* sum of grades */

    float average; /* number with decimal point for average */

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

    /* processing phase */
    /* get first grade from user */
    printf( "Enter grade, -1 to end: " ); /* prompt for input */
    scanf( "%d", &grade ); /* read grade from user */

Fig. 3.8 | C program and sample execution for the class average problem with sentinel-controlled repetition. (Part 1 of 3.)
/* loop while sentinel value not yet read from user */
while ( grade != -1 ) {
    total = total + grade; /* add grade to total */
    counter = counter + 1; /* increment counter */
}

/* get next grade from user */
printf( "Enter grade, -1 to end: "); /* prompt for input */
scanf("%d", &grade); /* read next grade */

/* termination phase */
/* if user entered at least one grade */
if ( counter != 0 ) {
    /* calculate average of all grades entered */
    average = ( float ) total / counter; /* avoid truncation */
    /* display average with two digits of precision */
    printf( "Class average is %.2f\n", average );
}
else { /* if no grades were entered, output message */
    printf( "No grades were entered\n" );
}

---

Fig. 3.8  |  C program and sample execution for the class average problem with sentinel-controlled repetition. (Part 2 of 3.)
46  return 0; /* indicate program ended successfully */
48  } /* end function main */

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  |  C program and sample execution for the class average problem with sentinel-controlled repetition. (Part 3 of 3.)
• Notice the compound statement in the `while` loop (line 24) in Fig. 3.8. Once again, the braces are necessary for all four statements to be 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.

  ```c
  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 3.6

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.
Formulating Algorithms with Top-Down, Stepwise Refinement
Case Study 2: Sentinel-Controlled Repetition

- 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 are 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.
• Dividing two integers results in integer division in which any fractional part of the calculation is lost (i.e., truncated).

• Since 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.
• Line 38
  • \( \text{average} = (\text{float}) \frac{\text{total}}{\text{counter}}; \)
• 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 integer value stored in counter.
Most computers can evaluate 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 promotion (also called implicit conversion) on selected operands.

For example, in an expression containing the data types `int` and `float`, copies of `int` operands are made and promoted to `float`.

In our example, after a copy of `counter` is made and promoted to `float`, the calculation is performed and the result of the floating-point division is assigned to `average`. 
C provides a set of rules for promotion of operands of different types.

Chapter 5 presents a discussion of all the standard data types and their order of promotion.

Cast operators are available for most data types.

The cast operator is formed by placing parentheses around a data type name.

The 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 like −7 or +5.
• 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 multiplicativa operators *, / and %.
• 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.
• 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 3.8

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.
Common Programming Error 3.9

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.
Despite the fact that 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 98.6, it may actually be 98.5999473210643.

The point here is that calling this number simply 98.6 is fine for most applications.
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 clearly the stored floating-point value can be only an approximation.
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.
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 have been asked to write a program to summarize the results. You have 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.
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!”
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 if 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 if more than 8 students passed the exam.
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.
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.
Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures

• 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.
• 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.
• 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 of blank lines to set off the *If...else*  
  to improve program readability.
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.

Notice that blank lines are also used to set off the `while` statement for program readability.
• 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.
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.
Performance Tip 3.1

Initializing variables when they’re defined can help reduce a program’s execution time.
Performance Tip 3.2

Many of the performance tips we mention in this text result in nominal improvements, so the reader may be tempted to ignore them. The cumulative effect of all these performance enhancements can make a program perform significantly faster. Also, significant improvement is realized when a supposedly nominal improvement is placed in a loop that may repeat a large number of times.
/* Fig. 3.10: fig03_10.c */
Analysis of examination results */
#include <stdio.h>

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

    /* initialize variables in definitions */
    int passes = 0; /* number of passes */
    int failures = 0; /* number of failures */
    int student = 1; /* student counter */
    int result; /* one exam result */

    /* process 10 students using counter-controlled loop */
    while ( student <= 10 ) {

        /* prompt user for input and obtain value from user */
        printf( "Enter result (1=pass, 2=fail): " );
        scanf( "%d", &result );

        /* if result 1, increment passes */
        if ( result == 1 ) {
            passes = passes + 1;
        }
    }

    return 0;
}

Fig. 3.10 | C program and sample executions for examination results problem. (Part 1 of 4.)
```c
    } /* end if */
    else {/* otherwise, increment failures */
        failures = failures + 1;
    } /* end else */

    student = student + 1; /* increment student counter */
} /* end while */

/* termination phase; display number of passes and failures */
printf("Passed %d\n", passes);
printf("Failed %d\n", failures);

/* if more than eight students passed, print "Bonus to instructor!" */
if ( passes > 8 ) {
    printf("Bonus to instructor!\n");
} /* end if */

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

**Fig. 3.10**  C program and sample executions for examination results problem. (Part 2 of 4.)
<table>
<thead>
<tr>
<th>Enter Result (1=pass, 2=fail)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>2</td>
</tr>
</tbody>
</table>

Passed 6
Failed 4

Fig. 3.10 | C program and sample executions for examination results problem. (Part 3 of 4.)
<table>
<thead>
<tr>
<th>Enter Result (1=pass, 2=fail):</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enter Result (1=pass, 2=fail):</td>
<td>1</td>
</tr>
<tr>
<td>Enter Result (1=pass, 2=fail):</td>
<td>1</td>
</tr>
<tr>
<td>Enter Result (1=pass, 2=fail):</td>
<td>2</td>
</tr>
<tr>
<td>Enter Result (1=pass, 2=fail):</td>
<td>1</td>
</tr>
<tr>
<td>Enter Result (1=pass, 2=fail):</td>
<td>1</td>
</tr>
<tr>
<td>Enter Result (1=pass, 2=fail):</td>
<td>1</td>
</tr>
<tr>
<td>Enter Result (1=pass, 2=fail):</td>
<td>1</td>
</tr>
</tbody>
</table>

Passed 9  
Failed 1  
Bonus to instructor!

**Fig. 3.10** | C program and sample executions for examination results problem. (Part 4 of 4.)
Software Engineering Observation 3.6

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 3.7

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.
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.
Assignment Operators

- Any statement of the form
  - \( \text{variable} = \text{variable operator expression}; \)

  where \text{operator is one of the binary operators} +, -, *, / or % (or others we’ll discuss in Chapter 10), can be written in the form
  - \( \text{variable operator}= \text{expression}; \)

- 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 Operators

<table>
<thead>
<tr>
<th>Assignment operator</th>
<th>Sample expression</th>
<th>Explanation</th>
<th>Assigns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+=</code></td>
<td><code>c += 7</code></td>
<td><code>c = c + 7</code></td>
<td>10 to c</td>
</tr>
<tr>
<td><code>-=</code></td>
<td><code>d -= 4</code></td>
<td><code>d = d - 4</code></td>
<td>1 to d</td>
</tr>
<tr>
<td><code>*=</code></td>
<td><code>e *= 5</code></td>
<td><code>e = e * 5</code></td>
<td>20 to e</td>
</tr>
<tr>
<td><code>/=</code></td>
<td><code>f /= 3</code></td>
<td><code>f = f / 3</code></td>
<td>2 to f</td>
</tr>
<tr>
<td><code>%=</code></td>
<td><code>g %= 9</code></td>
<td><code>g = g % 9</code></td>
<td>3 to g</td>
</tr>
</tbody>
</table>

*Assume: `int c = 3, d = 5, e = 4, f = 6, g = 12;`*

**Fig. 3.14** | Arithmetic assignment operators.
Assignment 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 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.
Assignment Operators

- 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.
### Increment and Decrement Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Sample expression</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>++a</td>
<td>Increment a by 1, then use the new value of a in the expression in which a resides.</td>
</tr>
<tr>
<td>++</td>
<td>a++</td>
<td>Use the current value of a in the expression in which a resides, then increment a by 1.</td>
</tr>
<tr>
<td>--</td>
<td>--b</td>
<td>Decrement b by 1, then use the new value of b in the expression in which b resides.</td>
</tr>
<tr>
<td>--</td>
<td>b--</td>
<td>Use the current value of b in the expression in which b resides, then decrement b by 1.</td>
</tr>
</tbody>
</table>

**Fig. 3.12** Increment and decrement operators
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.
/* 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", 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 */

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

Fig. 3.13  |  Preincrementing vs. postincrementing. (Part 1 of 2.)
Increment and Decrement Operators

- The program displays the value of `c` before and after the `++` operator is used.
- The decrement operator `(--)` works similarly.
Good Programming Practice 3.7
Unary operators should be placed directly next to their operands with no intervening spaces.
Increment and Decrement Operators

- 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++;
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.
Increment and Decrement Operators

- 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 3.10

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 avoid using statements with increment or decrement operators in which a particular variable being incremented or decremented appears more than once.
Increment and Decrement Operators

- 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 describes 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.