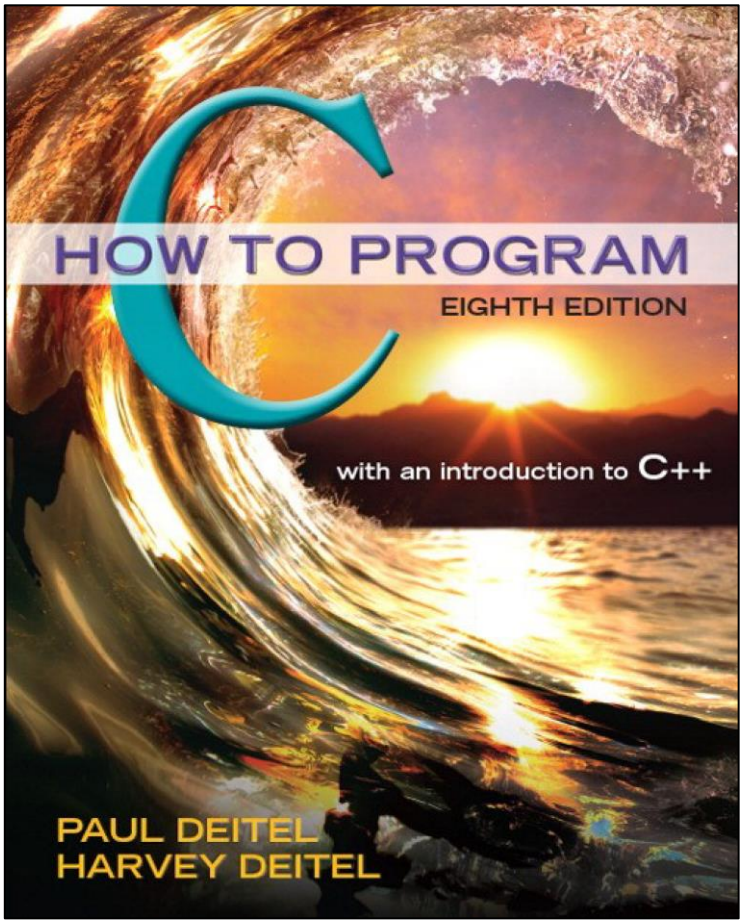
C How to Program

Eighth Edition



Chapter 3

Structured Program Development in C

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Learning Objectives

- 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` iteration statement to execute statements in a program repeatedly.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Learn structured programming.
- Use increment, decrement and assignment operators.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Outline (1 of 2)

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` Iteration Statement

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Iteration

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Outline (2 of 2)

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

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

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.2 Algorithms (1 of 3)

- 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.2 Algorithms (2 of 3)

- 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.2 Algorithms (3 of 3)

- 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**.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.3 Pseudocode (1 of 3)

- **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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.3 Pseudocode (2 of 3)

- A carefully prepared pseudocode program may be converted easily to a corresponding C program.
- Pseudocode consists only of action and decision 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.3 Pseudocode (3 of 3)

- 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 calculate on 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (1 of 11)

- 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (2 of 11)

- 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."

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (3 of 11)

- 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 **iteration structure**.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (4 of 11)

- 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 Figure 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**.

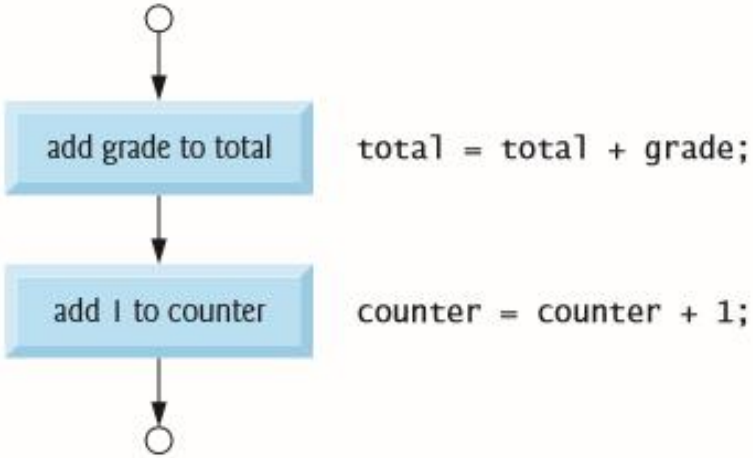
Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (5 of 11)

- 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 Figure 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.1 Flowcharting C's Sequence Structure



Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (6 of 11)

- 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 Figure 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (7 of 11)

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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (8 of 11)

- 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (9 of 11)

Iteration Statements in C

- C provides three types of iteration 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (10 of 11)

- C has only seven control statements: sequence, three types of selection and three types of iteration.
- 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 Figure 3.1(see slide 17), 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.4 Control Structures (11 of 11)

- 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.5 The `if` Selection Statement (1 of 6)

- 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).

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.5 The `if` Selection Statement (3 of 6)

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

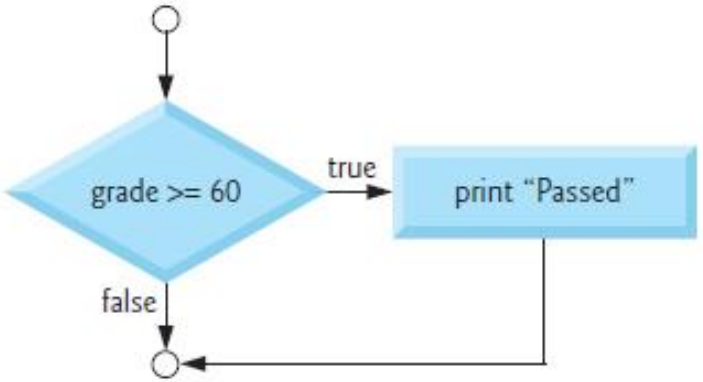
```
if ( grade >= 60 ) {
   printf( "Passed\n" );
} // end if
```

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

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.2 Flowcharting the Single-Selection `if` Statement



Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if…else` Selection Statement (2 of 14)

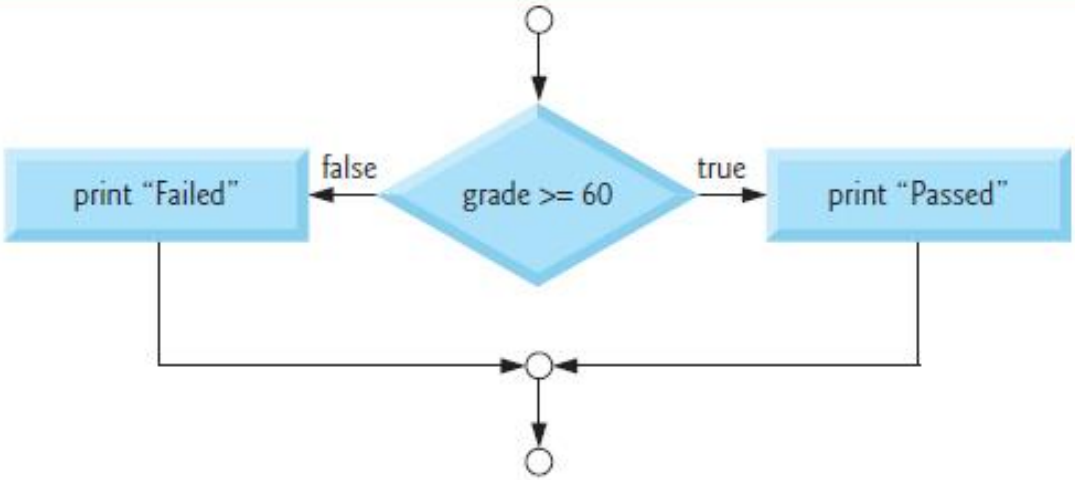
- The preceding pseudocode **If…else** statement may be written in C as

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

- The flowchart of Figure 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.3 Flowcharting the Double-Selection `if...else` Statement



Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if...else` Selection Statement (3 of 14)

- 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**.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if...else` Selection Statement (4 of 14)

- 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if...else` Selection Statement (6 of 14)

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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if...else` Selection Statement (7 of 14)

```
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"
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if...else` Selection Statement (8 of 14)

- 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" );
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if...else` Selection Statement (10 of 14)

- 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" );
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.6 The `if...else` Selection Statement (12 of 14)

- 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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.7 The `while` Iteration Statement (1 of 5)

- An iteration **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 iteration 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.7 The `while` Iteration Statement (2 of 5)

- The statement(s) contained in the **while** iteration 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 iteration terminates, and the first pseudocode statement **after** the iteration structure is executed.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.7 The `while` Iteration Statement (3 of 5)

- 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` iteration statement finishes executing, `product` will contain the desired answer:

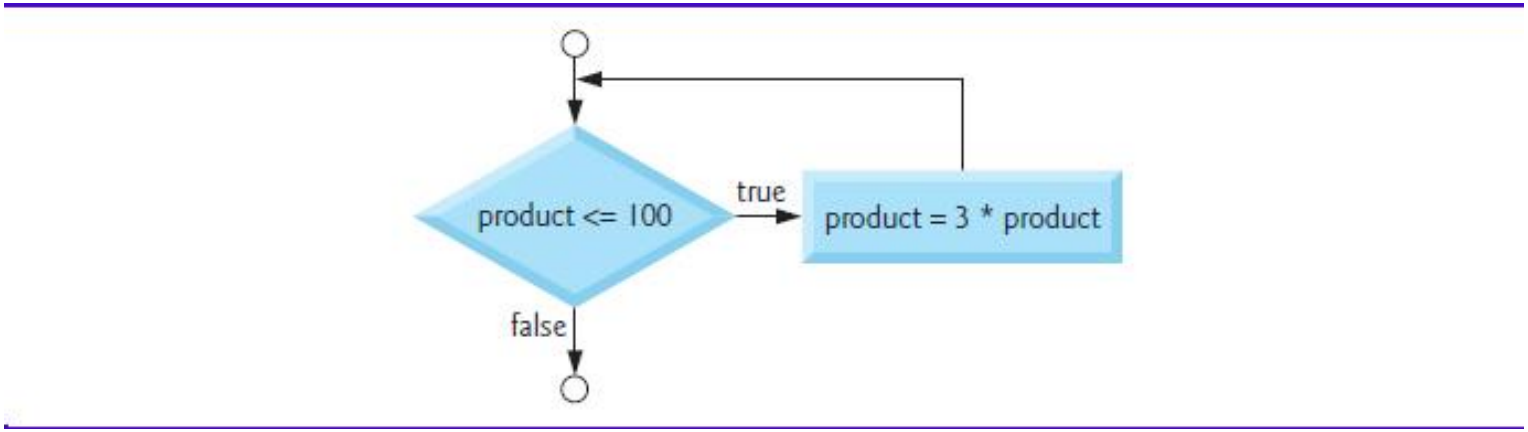
```
product = 3;
while ( product <= 100 ) {
   product = 3 * product;
} // end while
```

- The flowchart of Figure 3.4(see slide 39) illustrates the flow of control in the `while` iteration statement.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.4 Flowcharting the `while` Iteration Statement



Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.7 The `while` Iteration Statement (5 of 5)

- 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 iteration, and the final value of `product` is 243.
- Program execution continues with the next statement after the `while`.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Iteration (1 of 6)

- To illustrate how algorithms are developed, we solve several variations of a class-average 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 average calculation, and print the result.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.5 Pseudocode Algorithm That Uses Counter-Controlled Iteration to Solve the Class-Average Problem

```
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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.6 Class-Average Problem with Counter-Controlled Iteration (1 of 2)

```
1  // Fig. 3.6: fig03_06.c
2  // Class average program with counter-controlled iteration.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     unsigned int counter; // number of grade to be entered next
9     int grade; // grade value
10    int total; // sum of grades entered by user
11    int average; // average of grades
12
13    // initialization phase
14    total = 0; // initialize total
15    counter = 1; // initialize loop counter
16
17    // processing phase
18    while ( counter <= 10 ) { // loop 10 times
19       printf( "%s", "Enter grade: " ); // prompt for input
20       scanf( "%d", &grade ); // read grade from user
21       total = total + grade; // add grade to total
22       counter = counter + 1; // increment counter
23    } // end while
24
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.6 Class-Average Problem with Counter-Controlled Iteration (2 of 2)

```
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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Iteration (6 of 6)

- The average 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (1 of 21)

- Let's generalize the class-average problem.
- Consider the following problem:
  - Develop a class-average 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?

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (2 of 21)

- 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 iteration is often called **indefinite iteration** because the number of iterations isn't known before the loop begins executing.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (3 of 21)

- 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 average calculation).

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.7 Pseudocode Algorithm That Uses Sentinel-Controlled Iteration to Solve the Class-Average Problem

```
1   Initialize total to zero
2   Initialize counter to zero
3
4   Input the first grade (possibly the sentinel)
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"
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.8 Class-Average Program with Sentinel-Controlled Iteration (1 of 3)

```
1  // Fig. 3.8: fig03_08.c
2  // Class-average program with sentinel-controlled iteration.
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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.8 Class-Average Program with Sentinel-Controlled Iteration (2 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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.8 Class-Average Program with Sentinel-Controlled Iteration (3 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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (14 of 21)

- In the `while` loop in Figure 3.8, the braces are **necessary** to ensure that all four statements execute 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 `-1` is not input as the first grade.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (15 of 21)

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` 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (17 of 21)

- 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`.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (19 of 21)

- 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (20 of 21)

- 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` 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Iteration (21 of 21)

- 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
    ```

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved
    ```

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (1 of 11)

- 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (2 of 11)

- 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Statements (3 of 11)

- 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!”

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.9 Pseudocode for Examination-Results Problem

```
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!"
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.10 Analysis of Examination Results (1 of 4)

```
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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.10 Analysis of Examination Results (2 of 4)

```
21          // if result 1, increment passes
22          if ( result == 1 ) {
23             passes = passes + 1;
24          } // end if
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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.10 Analysis of Examination Results (3 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
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.10 Analysis of Examination Results (4 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!
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.11 Assignment Operators (1 of 2)

- 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.11 Assignment Operators (2 of 2)

- 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.

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.11 Arithmetic Assignment Operators

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

Assignment operator	Sample expression	Explanation	Assigns
+=	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

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.12 Increment and Decrement Operators (1 of 8)

- C also provides the unary **increment operator**, ++, and the unary **decrement operator**, `--`, which are summarized in Figure 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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.12 Increment and Decrement Operators

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.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.13 Preincrementing and Postincrementing (1 of 2)

```
1  // Fig. 3.13: fig03_13.c
2  // Preincrementing and postincrementing.
3  #include <stdio.h>
4
5  // function main begins program execution
6  int main( void )
7  {
8     int c; // define variable
9
10    // demonstrate postincrement
11    c = 5; // assign 5 to c
12    printf( "%d\n", c ); // print 5
13    printf( "%d\n", c++ ); // print 5 then postincrement
14    printf( "%d\n\n", c ); // print 6
15
16    // demonstrate preincrement
17    c = 5; // assign 5 to c
18    printf( "%d\n", c ); // print 5
19    printf( "%d\n", ++c ); // preincrement then print 6
20    printf( "%d\n", c ); // print 6
21 } // end function main
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.13 Preincrementing and Postincrementing (2 of 2)

```
5
5
6

5
6
6
```

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

3.12 Increment and Decrement Operators (5 of 8)

- The three assignment statements in Figure 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++;
```

Pearson
Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

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 $++(x+1)$.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Figure 3.14 Precedence and associativity of the operators encountered so far in the text

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

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved

Copyright

This work is protected by United States copyright laws and is provided solely for the use of instructors in teaching their courses and assessing student learning. Dissemination or sale of any part of this work (including on the World Wide Web) will destroy the integrity of the work and is not permitted. The work and materials from it should never be made available to students except by instructors using the accompanying text in their classes. All recipients of this work are expected to abide by these restrictions and to honor the intended pedagogical purposes and the needs of other instructors who rely on these materials.

Pearson

Copyright © 2016, 2013, 2010 Pearson Education, Inc. All Rights Reserved