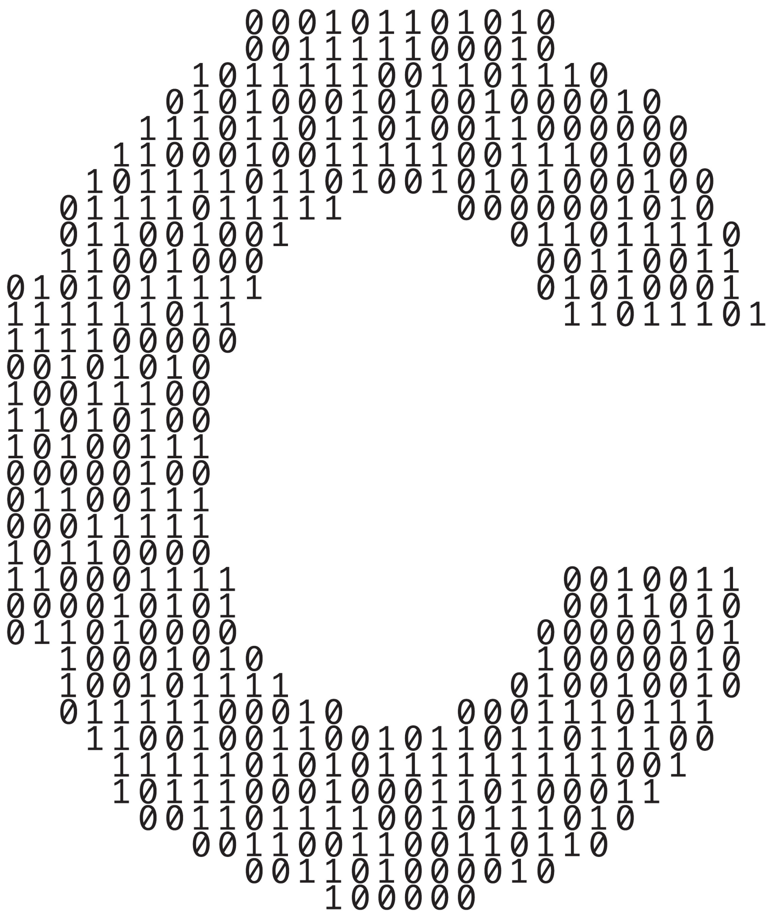
Structured Program Development

3



Objectives

In this chapter, you'll:

- Use basic problem-solving techniques.
- Develop algorithms through the process of top-down, stepwise refinement.
- Select actions to execute based on a condition using the `if` and `if`...`else` selection statements.
- Execute statements in a program repeatedly using the `while` iteration statement.
- Use counter-controlled iteration and sentinel-controlled iteration.
- Use structured programming techniques.
- Use increment, decrement and assignment operators.
- Continue our presentation of Secure C programming.

Outline

3.1 Introduction

Before writing a program to solve a problem, you must have a thorough understanding of the problem and a carefully planned solution approach. Chapters 3 and 4 discuss developing structured computer programs. In Section 4.11, we summarize the structured programming techniques developed here and in Chapter 4.

3.2 Algorithms

The solution to any computing problem involves executing a series of actions in a specific order. An **algorithm** is a **procedure** for solving a problem in terms of

1. the **actions** to execute, and
2. the **order** in which these actions should execute.

The following example shows that correctly specifying the order in which the actions should execute is important.

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, and
6. carpool to work.

This routine gets the executive to work well prepared to make critical decisions. 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 should execute in a computer program is called program control. In this and the next chapter, we investigate C's program control capabilities.

✓ Self Check

1 *(Fill-in-the-Blank)* 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 ________.

Answer: algorithm.

3.3 Pseudocode

Pseudocode is an informal artificial language similar to everyday English that helps you develop algorithms before converting them to structured C programs. Pseudocode is convenient and user friendly. It helps you "think out" a program before writing it in a programming language. Computers do not execute pseudocode.

Pseudocode consists purely of characters, so you may type it in any text editor. Often, converting carefully prepared pseudocode to C is as simple as replacing a pseudocode statement with its C equivalent.

Pseudocode describes the *actions* and *decisions* that will execute once you convert the pseudocode to C and run the program. Definitions are not executable statements—they're simply messages to the compiler. For example, the definition

```
int i = 0;
```

tells the compiler variable `i`'s type, instructs the compiler to reserve space in memory for the variable and initializes it to `0`. But this definition does not perform an action when the program executes, such as input, output, a calculation or a comparison. So, some programmers do not include definitions in their pseudocode. Others choose to list each variable and briefly mention its purpose.

✓ Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?

a) Pseudocode is useful for developing algorithms that will be converted to structured C programs.

b) Pseudocode is a computer programming language that's more concise than C.

c) Pseudocode consists purely of characters, so you may conveniently type it in any text-editor program.
d) Pseudocode describes the actions and decisions that will execute once you convert it to C and run the program.

Answer: b) is *false*. Actually, pseudocode is not an actual computer programming language. It helps you "think out" a program before writing it in a programming language.

3.4 Control Structures

Normally, statements in a program execute one after the other in the order in which you write them. This is called **sequential execution**. As you'll soon see, various C statements enable you to specify that the next statement to execute 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. 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**."

The research of Böhm and Jacopini[1] 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." It was not until well into the 1970s that the programming profession started taking structured programming seriously. 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.

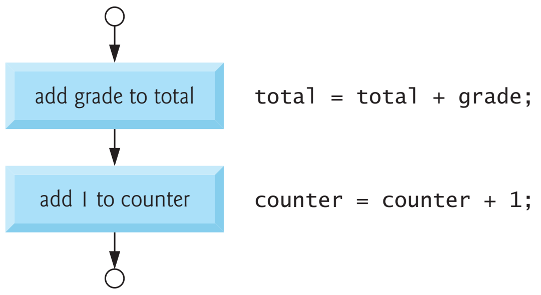
Böhm and Jacopini's work demonstrated that all programs could be written in terms of three **control structures**, namely the **sequence structure**, the **selection structure** and the **iteration structure**. The sequence structure is simple—unless directed otherwise, the computer executes C statements one after the other in the order in which they're written.

Flowcharts

A **flowchart** is a graphical representation of an algorithm or of a portion of an algorithm. You draw flowcharts using certain special-purpose symbols such as rectangles, diamonds, rounded rectangles, and small circles, connected by arrows called **flowlines**.

Flowcharts help you develop and represent algorithms, although pseudocode is preferred by most programmers. Flowcharts clearly show how control structures operate. Consider the following flowchart for a sequence structure in a portion of an algorithm that calculates the class average on a quiz:

1. C. Böhm and G. Jacopini, "Flow Diagrams, Turing Machines, and Languages with Only Two Formation Rules," *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 336–371. This classic computer-science paper is available at various online sites.



```
total = total + grade;

counter = counter + 1;
```

The rectangle (or action) symbol indicates any action, such as a calculation, input or output. The flowlines indicate the order in which to perform the actions. This program segment first adds `grade` to `total`, then adds `1` to `counter`. As we'll soon see, anywhere in a program a single action may be placed, you may place several actions in sequence.

When drawing a flowchart for a complete algorithm, the first symbol is a rounded rectangle symbol containing "Begin," and the last is a rounded rectangle containing "End." When drawing only a portion of an algorithm, we omit the rounded rectangle symbols in favor of using small circles called connector symbols.

Selection Statements in C

C provides three types of selection structures in the form of statements:

- The `if` single-selection statement (Section 3.5) selects (performs) an action (or group of actions) only if a condition is *true*.
- The `if...else` double-selection statement (Section 3.6) performs one action (or group of actions) if a condition is *true* and a different action (or group of actions) if the condition is *false*.
- The `switch` multiple-selection statement (discussed in the next chapter) performs one of many different actions, depending on the value of an expression.

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`. These statements perform tasks repeatedly. We discuss `do...while` and `for` in the next chapter.

Summary of Control Statements

That's all there is. C has only seven control statements: sequence, three types of selection and three types of iteration. You form each program by combining as many of each type of control statement as is appropriate for the algorithm the program implements.

We'll see that each control statement's flowchart representation 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.

We can attach the control-statement flowchart segments to one another by connecting the exit point of one to the entry point of the next. This is similar to a child stacking building blocks, so we call this control-statement stacking. You'll see later in this chapter that the only other way to connect control statements is via nesting.

Thus, any C program we'll ever need to build can be constructed from only seven control statements combined in only two ways. This is the essence of simplicity.

✓ Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?

a) Normally, statements in a program execute one after the other in the order in which they're written. This is called transfer of control.
b) Programs can be be written without any `goto` statements.
c) All programs can be written in terms of only three control structures—the sequence structure, the selection structure and the iteration structure.
d) 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.

Answer: a) is *false*. It's actually called sequential execution.

2 *(Multiple Choice)* Which of the following statements is *false*?

a) C has only seven control statements: sequence, three types of selection and three types of iteration.
b) Single-entry/single-exit control statements make it easy to build clear programs.
c) Connecting the exit point of one control statement to the entry point of the next is called control-statement stacking.
d) The only other way to connect control statements is via roosting. 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.

Answer: d) is *false*. The only other way to connect control statements is via nesting.

3.5 The `if` Selection Statement

Selection statements choose among alternative courses of action. For example, suppose the passing grade on an exam is 60. The following pseudocode statement determines whether the condition "student's grade is greater than or equal to 60" is *true* or *false*:

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

If *true*, then "Passed" is printed, and the next pseudocode statement in order is "performed." Remember that pseudocode isn't a real programming language. If *false*, the printing is ignored, and the next pseudocode statement in order is performed.

The preceding pseudocode is written in C as

```
if (grade >= 60) {
   puts("Passed");
} // end if
```

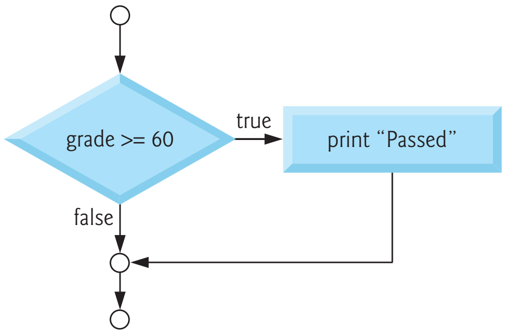
Of course, you'll also need to declare the `int` variable `grade`, but the C `if` statement code corresponds closely to the pseudocode. This is one of the properties of pseudocode that makes it such a useful program-development tool.

Indentation in the `if` Statement

The indentation in the `if` statement's second line is optional but highly recommended. It emphasizes the inherent structure of structured programs. The compiler ignores white-space characters such as blanks, tabs and newlines used for indentation and vertical spacing.

`if` Statement Flowchart

The following flowchart segment illustrates the single-selection `if` statement:



It contains perhaps the most important flowchart symbol—the diamond (or decision) symbol, which indicates a decision is to be made. The decision symbol's expression typically is a condition that can be *true* or *false*. The two flowlines emerging from it indicate the paths to take when the expression is *true* or *false*. Decisions can be based on any expression's value—zero is *false*, and nonzero is *true*.

The `if` statement is a single-entry/single-exit statement. We'll soon learn that the flowchart segment for the remaining control structures also can contain rectangle symbols to indicate the actions to be performed and diamond symbols to indicate decisions to be made. This is the action/decision model of programming we've been emphasizing.

✓ Self Check

1 *(Multiple Choice)* Which of the following statements is *false*?

a) The pseudocode statement

```
If student's grade is greater than or equal to 60
    Print "Passed"
```

can be written in C as

```
if (grade >= 60) {
   puts("Passed");
} // end if
```

b) The two flowlines emerging from the decision flowchart symbol indicate the directions to take when the expression in the symbol is *true* or *false*.

c) Decisions can be based on any expression—if the expression evaluates to nonzero, it's treated as *false*, and if it evaluates to zero, it's treated as *true*.

d) The `if` statement is a single-entry/single-exit statement.

Answer: c) is *false*. Actually, decisions 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*.

3.6 The if...else Selection Statement

The if...else selection statement specifies different actions to perform when the condition is *true* or *false*. For example, the pseudocode statement

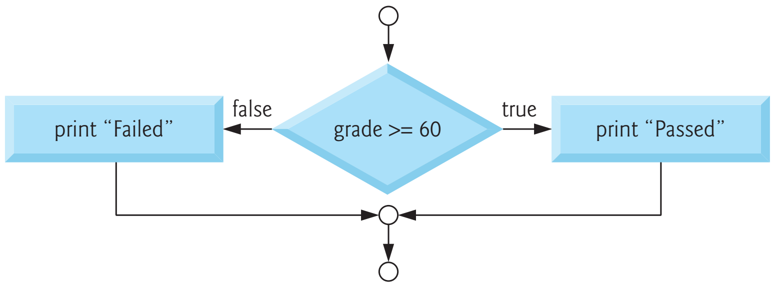
```
If student's grade is greater than or equal to 60
    Print "Passed"
else
    Print "Failed"
```

prints "Passed" if the student's grade is greater than or equal to 60; otherwise, it prints "Failed." In either case, after printing, the next pseudocode statement in sequence "executes." The *else*'s body also is indented. If there are several levels of indentation in a program, each should be indented the same additional amount of space. The preceding pseudocode may be written in C as

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

if...else Statement Flowchart

The following flowchart illustrates the if...else statement's flow of control:



Conditional Expressions

The **conditional operator** (?:) is closely related to the if...else statement. This operator is C's only **ternary operator**—that is, it takes three operands. A conditional operator and its three operands form a **conditional expression**. The first operand is a condition. The second is the conditional expression's value if the condition is *true*. The third is the conditional expression's value if the condition is *false*. For example, the conditional-expression argument to the following puts statement evaluates to the string "Passed" if the condition grade >= 60 is true; otherwise, it evaluates to the string "Failed":

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

Conditional operators can be used in places where if...else statements cannot, including expressions and arguments to functions (such as printf). Use expressions of the same type for the second and third operands of the conditional operator (?:) to avoid subtle errors. ERR ⊗

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 prints: A for 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.

```
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"
```

This pseudocode may be written in C as

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

If the variable grade is greater than or equal to 90, all four conditions are true, but only the puts statement after the first test executes. Then, the else part of the "outer" if...else statement is skipped, bypassing the rest of the nested if...else statement.

Most programmers write the preceding `if` statement as

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

Both forms are equivalent. The latter form avoids the deep indentation to the right, which decreases program readability and sometimes causes lines to wrap.

Blocks and Compound Statements

To include several statements in an `if`'s body, you must enclose the statements in braces (`{` and `}`). A set of statements contained within a pair of braces is called a **compound statement** or a **block**. A compound statement can be placed anywhere in a program that a single statement can be placed.

The following `if...else` statement's `else` part includes a compound statement containing two statements to execute if the condition is *false*:

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

If `grade` is less than `60`, both `puts` statements in the `else` execute and the code prints:

```
Failed.
You must take this course again.
```

The braces surrounding the two statements in the `else` clause are important. Without them, the statement

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

would be outside the `else`'s body (and outside the `if...else` statement) and would execute regardless of whether the grade was less than 60, so even a passing student would have to take the course again. To avoid problems like this, always include your control statements' bodies in braces (`{` and `}`), even if those bodies contain only a single statement. This solves the "dangling-else" problem, which we discuss in this chapter's exercises.

Kinds of Errors

A syntax error (such as misspelling "else") 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. ERR

Empty Statement

Anywhere a single or compound statement can be placed, it's possible to place an empty statement, represented by a semicolon (;). Placing a semicolon after an if's condition, as in ERR

```
if (grade >= 60);
```

leads to a logic error in single-selection if statements and a syntax error in double-selection and nested if...else statements.

Type both braces of compound statements before typing the individual statements within the braces. This helps avoid omission of one or both of the braces, preventing syntax errors (such as an if statement whose if part has multiple statements, which requires a pair of braces) and logic errors. Many integrated development environments and code editors insert the closing brace for you as soon as you type the opening one.

✓ Self Check

1 *(True/False)* The following code includes a compound statement in an if...else statement's else part:

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

Answer: *False.* The curly braces around the two puts statements in the else part of this if...else statement are missing. The correct code with the compound statement is

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

2 *(True/False)* The conditional expression argument to the following puts statement

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

evaluates to the string "Passed" if the condition grade >= 60 is *true*; otherwise, it evaluates to the string "Failed".

Answer: *False*. This statement won't compile because the conditional operator's ? and : are reversed. The correct statement to produce the desired result is

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

3.7 The while Iteration Statement

An iteration statement (also called a repetition statement or loop) repeats an action 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*, the shopper performs the action "Purchase next item and cross it off my list" repeatedly while the condition remains *true*. 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 statement "executes."

Calculating the First Power of 3 Greater Than 100

As a while statement example, consider a program segment that finds the first power of 3 larger than 100. The integer variable product is initialized to 3. When the following code segment finishes executing, product will contain the desired answer:

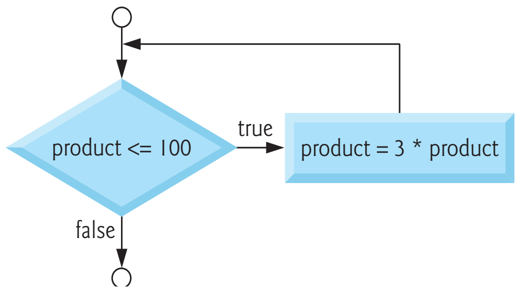
```
int product = 3;

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

The loop repeatedly multiplies product by 3, so it takes on the values 9, 27 and 81 successively. When product becomes 243, the condition product <= 100 becomes *false*, terminating the iteration—product's final value is 243. Execution continues with the next statement after the while. An action in the while statement's body must eventually cause the condition to become *false*; otherwise, the loop will never terminate—a logic error called an infinite loop. The statement(s) contained in a while iteration statement constitute its body, which may be a single statement or a compound statement.

ERR ⊗

while Statement Flowchart

The following flowchart segment illustrates the preceding while iteration statement:



The flowchart clearly shows the iteration—the flowline emerging from the rectangle points back to the flowline entering the decision. The loop tests the condition in the diamond during each iteration until the condition eventually becomes *false*. At this point, the `while` statement exits and control continues with the next statement in sequence.

✓ Self Check

1 *(Program Segment)* The `while` statement program segment in this section finds the first power of 3 larger than 100. Rewrite this program segment so that it will find the first power of 2 greater than or equal to 1024, leaving it in `product`.
Answer: See below.

```
int product = 2;

while (product < 1024) {
   product = 2 * product;
}
```

2 *(Fill-in-the-Blank)* An action in a `while` statement's body must eventually cause the condition to become *false*; otherwise, the loop will never terminate. This is a logic error called a(n) ________.
Answer: infinite loop.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Iteration

To illustrate how algorithms are developed, we solve two variations of a class-averaging problem in this section and the next. 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 the sum of the grades divided by the number of students. The algorithm to solve this problem must input the grades, then calculate and display the class average.

Pseudocode for the Class-Average Problem

Let's use pseudocode to list the actions to execute and specify the order in which they should execute. We use counter-controlled iteration 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, we know that ten students took a quiz, so we need to input 10 grades. Iteration terminates when the counter exceeds 10. In this case study, we simply present the final pseudocode algorithm (Fig. 3.1) and the corresponding C program (Fig. 3.2). In the next case study, we show how to develop pseudocode algorithms. Counter-controlled iteration is often called definite iteration because the number of iterations is known before the loop begins executing.

```
Set total to zero
Set grade counter to one

While grade counter is less than or equal to ten
    Input the next grade
    Add the grade into the total
    Add one to the grade counter

Set the class average to the total divided by ten
Print the class average
```

Fig. 3.1 | Pseudocode algorithm that uses counter-controlled iteration to solve the class-average problem.

```c
// fig03_02.c
// Class average program with counter-controlled iteration.
#include <stdio.h>

// function main begins program execution
int main(void) {
   // initialization phase
   int total = 0; // initialize total of grades to 0
   int counter = 1; // number of the grade to be entered next

   // processing phase
   while (counter <= 10) { // loop 10 times
      printf("%s", "Enter grade: "); // prompt for input
      int grade = 0; // grade value
      scanf("%d", &grade); // read grade from user
      total = total + grade; // add grade to total
      counter = counter + 1; // increment counter
   } // end while

   // termination phase
   int average = total / 10; // integer division
   printf("Class average is %d\n", average); // display result
} // 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. 3.2 | Class-average problem with counter-controlled iteration.

A total is a variable (line 8) used to accumulate the sum of a series of values. A counter is a variable (line 9) used to count—in this case, to count the number of grades entered. Variables for totals should be initialized to zero; otherwise, the sum would include the previous value stored in the total's memory location. You should initialize all counters and totals. Counters typically are 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. If a counter or total isn't initialized, the results of your program will probably be incorrect. These are examples of logic errors. ⊗ERR

The average was 81 in the preceding sample execution, but the sum of the grades we input was 817. Of course, 817 divided by 10 should yield 81.7—a number with a decimal point. The next section shows how to deal with such *floating-point numbers*.

✓ Self Check

1 *(Fill-in-the-Blank)* Counter-controlled iteration is often called ________ iteration because the number of iterations is known before the loop begins executing.
Answer: definite.

2 *(True/False)* Variables used to store totals should be initialized to one before being used in a program; otherwise, the sum would include the previous value stored in the total's memory location.
Answer: *False*. Actually, variables used to store totals should 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.

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

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, we knew there were 10 grades in advance. In this example, no indication is given of how many grades the user might input. The program must process an *arbitrary* number of grades. How can the program determine when to stop inputting grades? How will it know when to calculate and print the class average?

Sentinel Values

One way is to use a sentinel value to indicate "end of data entry." A sentinel value also is called a signal value, a dummy value, or a flag value. The user types 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.

You should choose a sentinel value that cannot be confused with an acceptable input value. Grades on a quiz are non-negative integers, so –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. The sentinel value –1 should not enter into the averaging calculation.

Top-Down, Stepwise Refinement

We approach the class-average program with a technique called **top-down, stepwise refinement**, which is essential to developing 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. 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 smaller tasks listed 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 should execute in order, one after the other. Each refinement, as well as the top itself, is a complete specification of the algorithm. Only the level of detail varies.

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 grades,
- a count of how many grades have been processed,
- a variable to receive the value of each grade as it is input and
- a variable to hold the calculated average.

The pseudocode statement

```
Initialize variables
```

can be refined as follows:

```
Initialize total to zero
Initialize counter to zero
```

Only the total and counter need to be initialized. The variables for the calculated average and the grade the user inputs need not be initialized because their values will be calculated and input from the user, respectively. The pseudocode statement

```
Input, sum, and count the quiz grades
```

requires an *iteration structure* that successively inputs each grade. Because we do not know how many grades are to be processed, we'll use sentinel-controlled iteration. The user will enter legitimate grades one at a time. After entering the last legitimate grade, 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 (possibly the sentinel)
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)
```

In pseudocode, we do not use braces around the set of statements that form a loop's body. We simply indent the body statements under the *while*. Again, pseudocode is an informal program-development aid.

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"
```

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"**). You should explicitly test for this case and handle it appropriately in your program, such as by printing an error message, rather than allowing the fatal error to occur. ERR

Complete Second Refinement

The complete second refinement is shown in Fig. 3.3. We include some blank lines in the pseudocode for readability.

```
Initialize total to zero
Initialize counter to zero

Input the first grade (possibly the sentinel)
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)
```

Fig. 3.3 | Pseudocode algorithm that uses sentinel-controlled iteration to solve the class-average problem. (Part 1 of 2.)

```
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. 3.3 | Pseudocode algorithm that uses sentinel-controlled iteration to solve the class-average problem. (Part 2 of 2.)

Phases in a Basic Program

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.

Number of Pseudocode Refinements

The pseudocode algorithm in Fig. 3.3 solves the more general class-average problem. This algorithm was developed after only two levels of refinement. Sometimes more levels are necessary. You terminate the top-down, stepwise refinement process when the pseudocode algorithm provides sufficient detail for you to convert the pseudocode to C.

The most challenging part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, producing a working C program usually is straightforward. Many programmers write programs without ever using program-development tools such as pseudocode. They feel their ultimate goal is to solve the problem and that writing pseudocode merely delays producing final outputs. This may work for small programs you develop for your own use. But for the substantial programs and software systems you'll likely work on in industry, a formal development process is essential.

Class-Average Program for an Arbitrary Number of Grades

Figure 3.4 shows the C program and two sample executions. 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. So this program introduces the data type **`double`** to handle numbers with decimal points—that is, **floating-point numbers**. We introduce a cast operator to force the averaging calculation to use floating-point numbers. These features are explained after the program listing. Note that lines 13 and 23 both include the sentinel value in the prompts requesting data entry. This is a good practice in a sentinel-controlled loop.

```c
// fig03_04.c
// Class-average program with sentinel-controlled iteration.
#include <stdio.h>

// function main begins program execution
int main(void) {
   // initialization phase
   int total = 0; // initialize total
   int counter = 0; // initialize loop counter

   // processing phase
   // get first grade from user
   printf("%s", "Enter grade, -1 to end: "); // prompt for input
   int grade = 0; // grade value
   scanf("%d", &grade); // read grade from user

   // 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("%s", "Enter grade, -1 to end: "); // prompt for input
      scanf("%d", &grade); // read next grade
   } // end while

   // termination phase
   // if user entered at least one grade
   if (counter != 0) {

      // calculate average of all grades entered
      double average = (double) total / counter; // avoid truncation

      // display average with two digits of precision
      printf("Class average is %.2f\n", average);
   } // end if
   else { // if no grades were entered, output message
      puts("No grades were entered");
   } // end else
} // 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. 3.4 | Class-average program with sentinel-controlled iteration.

Always Use Braces in a while Statement

Without this while loop's braces (lines 18 and 25), only the statement on line 19 would be in the loop's body. The code would be incorrectly interpreted as

```
while (grade != -1)
   total = total + grade; // add grade to total
counter = counter + 1; // increment counter

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

ERR ⊗ This would cause an infinite loop if the user did not input -1 as the first grade.

Converting Between Types Explicitly and Implicitly

Averages often are values such as 7.2 or –93.5 that contain fractional parts. These floating-point numbers can be represented by the data type double. Line 32 defines the variable average as type double to capture the fractional result of our calculation. Normally, the result of the calculation total / counter (line 32) is an integer because total and counter are both int variables. Dividing two ints results in **integer division**—any fractional part of the calculation is **truncated** (that is, lost). You can produce a floating-point calculation with integer values by first creating temporary floating-point numbers. C provides the unary **cast operator** to accomplish this task. Line 32

```
double average = (double) total / counter;
```

uses the cast operator (double) to create 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 double version of total—divided by the int value stored in counter.

C requires the operand data types in arithmetic expressions only to be identical. In mixed-type expressions, the compiler performs an operation called **implicit conversion** on selected operands to ensure that they're of the same type. For example, in an expression containing the data types int and double, copies of int operands are made and implicitly converted to type double. After we explicitly convert total to a double, the compiler implicitly makes a double copy of counter, then performs floating-point division and assigns the floating-point result to average. Chapter 5 discusses C's rules for converting operands of different types.

Cast operators are formed by placing parentheses around a type name. A cast is a **unary 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. Cast operators group 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 3.4 uses the printf conversion specification %.2f (line 35) to format average's value. The f specifies that a floating-point value will be printed. The .2 is the **preci-**

sion—the value will have two (2) digits to the right of the decimal point. If the %f conversion specification is used without specifying the precision, the default precision is 6 digits to the right of the decimal point, as if the conversion specification %.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. The following statements display the values 3.45 and 3.4, respectively:

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

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 degrees Fahrenheit, 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. We'll say more about this issue later.

Floating-point numbers often develop 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. 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. For this reason, you also should not compare floating-point values for equality. ERR

✓ Self Check

1 *(Fill-in-the-Blank)* Sentinel-controlled iteration is often called ________ iteration because the number of iterations isn't known before the loop begins executing.
Answer: indefinite.

2 *(Multiple Choice)* Which of the following statements is *false*?

a) 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.

b) You terminate top-down, stepwise refinement when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C.

c) Experience has shown that the most difficult part of solving a problem on a computer is producing a working C program from the algorithm.

d) 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.

Answer: c) is *false*. Actually, 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.

3 *(Program Segment)* When floating-point values are printed with precision, the printed value is rounded. Write statements that display the value 98.5999473210643 with one, four and ten digits of precision, respectively, and specify what's displayed in each case.
Answer: See below.

```
printf("%.1f\n", 98.5999473210643); // displays 98.6
printf("%.4f\n", 98.5999473210643); // displays 98.5999
printf("%.10f\n", 98.5999473210643); // displays 98.5999473211
```

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

Let's work another complete problem. We'll 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 by **nesting** 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'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 is a 1 if the student passed the exam or a 2 if the student failed.

Your program should analyze the results of the exam as follows:

1. *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.*
2. *Count the number of test results of each type.*
3. *Display a summary of the test results indicating the number of students who passed and the number who failed.*
4. *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:

1. The program must process 10 test results. We'll use a counter-controlled loop.
2. Each test result is a number—either a 1 or a 2. Each time the program reads a test result, it must determine whether the result is a 1 or a 2. We'll test for a

1 in our algorithm. If the number is not a 1, we'll assume that it's a 2. Exercise 3.27 asks you to ensure that every test result is a 1 or a 2.

3. 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.
4. After the program has processed all the results, it must decide whether more than 8 students passed the exam and, if so, print `"Bonus to Instructor!"`.

Pseudocode Representation of the Top

Let's proceed with top-down, stepwise refinement. We begin with a pseudocode representation of the top:

```
Analyze exam results and decide whether instructor should receive a bonus
```

Once again, it's important to emphasize that the top is a complete representation of the program, but multiple refinements are likely to be needed before the pseudocode can be naturally evolved into a C program.

First Refinement

Our first refinement is:

```
Initialize variables
Input the ten quiz grades and count passes and failures
Print an exam-results summary and decide whether to bonus the instructor
```

Here, too, even though we have a *complete* representation of the entire program, further refinement is necessary.

Second Refinement

We now commit to specific variables. We need counters to record the passes and failures, a counter to control the looping process, and a variable to store the user input. The pseudocode statement

```
Initialize variables
```

can be refined as follows:

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one
```

Only the counter 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 we know in advance that there are precisely ten exam results, so counter-controlled looping is appropriate. Inside the loop (that is, **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 counter. 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
```

The pseudocode statement

```
Print an exam-results summary and decide whether to bonus the instructor
```

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!"
```

Complete Second Refinement

Figure 3.5 contains the complete second refinement. We use blank lines for readability.

```
Initialize passes to zero
Initialize failures to zero
Initialize student to one

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

Print the number of passes
Print the number of failures
If more than eight students passed
    Print "Bonus to instructor!"
```

Fig. 3.5 | Pseudocode for examination-results problem.

Implementing the Algorithm

This pseudocode is now sufficiently refined for conversion to C. Figure 3.6 shows the C program and two sample executions.

```c
// fig03_06.c
// Analysis of examination results.
#include <stdio.h>

// function main begins program execution
int main(void) {
   // initialize variables in definitions
   int passes = 0;
   int failures = 0;
   int student = 1;

   // process 10 students using counter-controlled loop
   while (student <= 10) {
      // prompt user for input and obtain value from user
      printf("%s", "Enter result (1=pass,2=fail): ");
      int result = 0; // one exam result
      scanf("%d", &result);

      // if result 1, increment passes
      if (result == 1) {
         passes = passes + 1;
      } // 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) {
      puts("Bonus to instructor!");
   } // end if
} // end function main
```

```
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. 3.6 | Analysis of examination results. (Part 1 of 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): 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. 3.6 | Analysis of examination results. (Part 2 of 2.)

✓ Self Check

1 *(Fill-in-the-Blank)* Control statements may be stacked on top of one another (in sequence) just as a child stacks building blocks. The only other structured way control statements may be connected in C is ________—that is, placing a control statement inside another control statement.
Answer: nesting.

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 operator's right to the value of the variable on the operator's left then stores the result in the variable on the left. So, the assignment `c += 3` adds `3` to `c`'s current value. The following table 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`

✓ Self Check

1 *(Fill-in-the-Blank)* The statement

```
b = b * 5;
```

can be abbreviated with the multiplication assignment operator `*=` as ________.
Answer: `b *= 5;`

2 *(Fill-in-the-Blank)* What does the assignment `c -= 3;` do? ________
Answer: Subtracts `3` from `c`'s current value.

3.12 Increment and Decrement Operators

The unary **increment operator** (`++`) and the unary **decrement operator** (`--`) add one to and subtract one from an integer variable, respectively. The following table summarizes the two versions of each operator:

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.

To increment the variable `c` by 1, you can use the operator `++` rather than the expressions `c = c + 1` or `c += 1`. If you place `++` or `--` before a variable (i.e., prefixed), they're referred to as the **preincrement** or **predecrement operators**. If you place `++` or `--` after a variable (i.e., postfixed), they're referred to as the **postincrement** or **postdecrement operators**. By convention, unary operators should be placed next to their operands with no intervening spaces.

Figure 3.7 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. The program displays the value of `c` before and after using `++`. The decrement operator (`--`) works similarly.

```
// fig03_07.c
// Preincrementing and postincrementing.
#include <stdio.h>

```

Fig. 3.7 | Preincrementing and postincrementing. (Part 1 of 2.)

```
// function main begins program execution
int main(void) {
   // demonstrate postincrement
   int 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
```

```
5
5
6

5
6
6
```

Fig. 3.7 | Preincrementing and postincrementing. (Part 2 of 2.)

The three assignment statements in Fig. 3.6

```
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++;
```

When incrementing or decrementing a variable in a statement by itself, the preincrement and postincrement forms have the same effect. 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).

Only a simple variable name may be used as a `++` or `--` operator's operand. Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error—e.g., `++(x + 1)`.

ERR ⊗

C generally does not specify the order in which an operator's operands will evaluate. We'll see exceptions to this for a few operators in the next chapter. To avoid subtle errors, the ++ and -- operators should be used only in statements that modify exactly one variable.

The following table lists in decreasing precedence order the operators shown so far.

Operators	Grouping	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

The third column names the various groups of operators. Notice that the conditional operator (`?:`), the unary operators increment (`++`), decrement (`--`), plus (`+`), minus (`-`) and casts, and the assignment operators `=`, `+=`, `-=`, `*=`, `/=` and `%=` group right-to-left. The other operators group left-to-right.

✓ Self Check

1 *(Multiple Choice)* Given the following code:

```
--i;
```

which of the following statements describes what this code does?

a) Increment `i` by 1, then use the new value of `i` in the expression in which `i` resides.
b) Use `i`'s current value in the expression in which `i` resides, then increment `i` by 1.
c) Decrement `i` by 1, then use the new value of `i` in the expression in which `i` resides.
d) Use `i`'s current value in the expression in which `i` resides, then decrement `i` by 1.

Answer: c.

2 *(What Does This Code Do?)* What does this program display as `x`'s final value?

```
#include <stdio.h>

int main(void) {
   int x = 7;
   printf("%d\n", x);
   printf("%d\n", x++);
   printf("%d\n\n", x);
```

```
    x = 8;
    printf("%d\n", x);
    printf("%d\n", ++x);
    printf("%d\n", x);
}
```

Answer: 9.

3 *(Multiple Choice)* Which of the following expressions contains a syntax error?

a) `++x + 1`
b) `x++ + x`
c) `++(x) + 1`
d) `++(x + 1)`

Answer: d. Only a simple variable name may be used as the operand of a `++` or `--` operator. Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error—e.g., `++(x + 1)`.

SEC 3.13 Secure C Programming

Arithmetic Overflow

Figure 2.4 presented an addition program that calculated the sum of two `int` values with the statement

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

ERR Even this simple statement has a potential problem. Adding the integers could result in a value too large to store in the `int` variable `sum`. This is known as **arithmetic overflow** and can cause undefined behavior, possibly leaving a system open to attack.

The constants `INT_MAX` and `INT_MIN` represent the platform-specific maximum and minimum values that can be stored in an `int` variable. These constants are defined in the header `<limits.h>`. There are similar constants for the other integral types that we'll introduce in the next chapter. You can see your platform's values for these constants by opening the header `<limits.h>` in a text editor.[2]

It's good practice to ensure that before you perform arithmetic calculations like the one above, they will not overflow. For an example, see the CERT website.

```
https://wiki.sei.cmu.edu/confluence/display/c/
```

Search for guideline INT32-C. The code uses the `&&` (logical AND) and `||` (logical OR) operators, which we discuss in Chapter 4. In industrial-strength code, you should perform checks like these for all calculations. Later chapters show other programming techniques for handling such errors.

`scanf_s` and `printf_s`

The C11 standard's Annex K introduced more secure versions of `printf` and `scanf` called `printf_s` and `scanf_s`. We discuss these functions and the corresponding secu-

2. Use your system's search feature to locate the file `limits.h`.

rity issues in Sections 6.13 and 7.13. Annex K is designated as optional, so not every C vendor implements it. In particular, the GNU C++ and Clang C++ compilers do not implement Annex K, so using `scanf_s` and `printf_s` might compromise your code's portability among compilers.

Microsoft implemented its own Visual C++ versions of `printf_s` and `scanf_s` before the C11 standard, and its compiler immediately began issuing warnings for every `scanf` call. The warnings said that `scanf` was deprecated—it should no longer be used—and that you should consider using `scanf_s` instead. Microsoft now treats what used to be a warning about `scanf` as an error. A program with `scanf` will not compile on Visual C++ and you will not be able to execute the program.

Many organizations have coding standards that require code to compile without warning messages. There are two ways to eliminate Visual C++'s `scanf` warnings—use `scanf_s` instead of `scanf` or 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.

You'll no longer receive warnings on `scanf` (or any other functions that Microsoft has deprecated for similar reasons). For industrial-strength coding, disabling the warnings is discouraged. We'll say more about `scanf_s` and `printf_s` in a later Secure C Coding Guidelines section.

✓ Self Check

1 *(Fill-in-the-Blank)* Adding two integers such that the result is a value that's too large to store in an `int` variable is known as arithmetic ________ and can cause undefined behavior, possibly leaving a system open to attack.
Answer: overflow.

2 *(Fill-in-the-Blank)* The platform-specific 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 ________.
Answer: `<limits.h>`.

Summary

Section 3.1 Introduction

• Before writing a program to solve a particular problem, you must have a thorough understanding of the problem and a carefully planned approach to solving it.

Section 3.2 Algorithms

- The solution to any computing problem involves executing a series of **actions** in a specific **order** (p. 138).
- An **algorithm** (p. 138) is a **procedure** (p. 138) for solving a problem in terms of the actions (p. 138) to execute and the order in which these actions should execute.

Section 3.3 Pseudocode

- **Pseudocode** (p. 139) is an artificial and informal language that helps you develop algorithms.
- Pseudocode is similar to everyday English; it's not an actual computer programming language.
- Pseudocode programs help you "think out" a program.
- Pseudocode consists purely of characters. You may type pseudocode using any text editor.
- Carefully prepared pseudocode can be converted easily to corresponding C programs.
- Pseudocode consists only of actions and decisions.

Section 3.4 Control Structures

- Normally, statements in a program execute one after the other in the order in which they're written. This is called **sequential execution** (p. 140).
- Various C statements enable you to specify that the next statement to execute may be other than the next one in sequence. This is called **transfer of control** (p. 140).
- **Structured programming** has become almost synonymous with "**goto elimination**" (p. 140).
- Structured programs are clearer, easier to debug and modify and more likely to be bug-free.
- All programs can be written using **sequence**, **selection** and **iteration control structures** (p. 140).
- Unless directed otherwise, the computer automatically executes C statements in sequence.
- A **flowchart** (p. 140) is a graphical representation of an algorithm drawn using **rectangles**, **diamonds**, **rounded rectangles** and **small circles** connected by arrows called **flowlines** (p. 140).
- The **rectangle (action) symbol** (p. 141) indicates any type of action, including a calculation or an input/output operation.
- **Flowlines** indicate the order in which the actions are performed.
- When drawing a flowchart that represents a complete algorithm, we use as the first symbol a rounded rectangle containing "Begin" and as the last a rounded rectangle containing "End." When drawing only a portion of an algorithm, we omit the rounded-rectangle symbols in favor of using small circles called **connector symbols**.
- The **`if` single-selection statement** selects or ignores a single action (or group of actions).
- The **`if...else` double-selection statement** (p. 141) selects between two different actions (or groups of actions).
- The **`switch` multiple-selection statement** (p. 141) selects among many different actions based on the value of an expression.
- C provides three types of **iteration statements** (also called repetition statements), namely **`while`**, **`do...while`** and **`for`**.
- Control-statement flowchart segments can be attached to one another with control-statement **stacking** (p. 141)—connecting the exit point of one to the entry point of the next.
- Control statements also may be **nested**.
- C uses **single-entry/single-exit control statements** (p. 141).

Section 3.5 The `if` Selection Statement

- Selection structures are used to choose among alternative courses of action.
- The **diamond (decision) symbol** (p. 143) indicates that a decision is to be made.
- The **decision symbol**'s expression typically is a condition that can be *true* or *false*. The decision symbol has two flowlines emerging from it indicating the directions to take when the expression is *true* or *false*.
- A decision can be based on any expression's value—zero is *false* and nonzero is *true*.

Section 3.6 The `if…else` Selection Statement

- The **conditional operator** (`?:`, p. 144) is closely related to the `if…else` statement.
- The conditional operator is C's only **ternary operator**—it takes three operands. The first is a condition. The second is the value for the **conditional expression** (p. 144) if the condition is *true*. The third is the value for the conditional expression if the condition is *false*.
- **Nested `if…else` statements** (p. 145) test for multiple cases by placing `if…else` statements inside `if…else` statements.
- A set of statements within a pair of braces is called a **compound statement** or a **block** (p. 146).
- 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.

Section 3.7 The `while` Iteration Statement

- The **`while` iteration statement** (p. 148) specifies that an action repeats while a condition is *true*. Eventually, the condition will become *false*. At this point, the iteration terminates, and the first statement after the iteration statement executes.

Section 3.8 Formulating Algorithms Case Study 1: Counter-Controlled Iteration

- **Counter-controlled iteration** (p. 149) uses a variable called a **counter** (p. 149) to specify the number of times a set of statements should execute.
- Counter-controlled iteration is often called **definite iteration** (p. 149) because the number of iterations is known before the loop begins executing.
- A **total** (p. 151) is a variable used to accumulate the sum of a series of values. Variables used to store totals should be initialized to zero.
- A **counter** is a variable used to count. Counter variables typically are initialized to zero or one, depending on their use.
- An **uninitialized variable** contains a **"garbage" value** (p. 151)—the value last stored in the memory location reserved for that variable.

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

- A **sentinel value** (p. 151; also called a **signal value**, a **dummy value**, or a **flag value**) is used in a sentinel-controlled loop to indicate the **"end of data entry."**
- **Sentinel-controlled iteration** is often called **indefinite iteration** (p. 151) because the number of iterations is not known before the loop begins executing.
- The sentinel value must be chosen so that it cannot be confused with an acceptable input value.

- In **top-down, stepwise refinement** (p. 152), the **top** is a statement that conveys the program's overall function. It's a complete representation of a program. In the **refinement process**, we divide the top into smaller tasks and list these in execution order.
- The type `double` (p. 154) represents **floating-point numbers** with decimal points.
- When two integers are divided, any fractional part of the result is **truncated** (p. 156).
- To produce a floating-point calculation with integer values, you can **cast** the integers to floating-point numbers. C provides the **unary cast operator** `(double)` to accomplish this task.
- Cast operators (p. 156) perform **explicit conversions**.
- C requires the operands in arithmetic expressions to have the same data type. To ensure this, the compiler performs **implicit conversion** (p. 156) on selected operands.
- A cast operator is formed by placing parentheses around a type name. The cast operator is a **unary operator**—it takes only one operand.
- **Cast operators group 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 `*`, `/` and `%`.
- The `printf` conversion Specification `%.2f` specifies that a floating-point value will be displayed with two digits to the right of the decimal point. If the `%f` conversion specification is used (without specifying the precision), the **default precision** (p. 157) is 6.
- When floating-point values are printed with precision, the printed value is **rounded** (p. 157) to the indicated number of decimal positions for display purposes.

Section 3.11 Assignment Operators

- C provides several assignment operators for **abbreviating assignment expressions** (p. 162).
- The `+=` operator adds the value of the expression on its right to the value of the variable on its left and stores the result in the variable on its left.
- Assignment operators are provided for each of the binary operators `+`, `-`, `*`, `/` and `%`.

Section 3.12 Increment and Decrement Operators

- C provides the **unary increment operator**, `++` (p. 163), and the **unary decrement operator**, `--` (p. 163), for use with integral types.
- If `++` or `--` operators are placed before a variable, they're referred to as the **preincrement** or **predecrement operators**, respectively. If `++` or `--` operators are placed after a variable, they're referred to as the **postincrement** or **postdecrement operators**, respectively.
- Preincrementing (predecrementing) a variable causes it to be incremented (decremented) by 1, then the new value of the variable is used in the expression in which it appears.
- Postincrementing (postdecrementing) a variable uses the current value of the variable in the expression in which it appears, then the variable value is incremented (decremented) by 1.
- When incrementing or decrementing a variable in a statement by itself, pre- and postincrement have the same effect. When a variable appears in the context of a larger expression, pre- and postincrementing have different effects (and similarly for pre- and postdecrementing).

Section 3.13 Secure C Programming

- Adding integers can result in a value that's too large to store in an `int` variable. This is known as **arithmetic overflow** and can cause unpredictable runtime behavior, possibly leaving a system open to attack.

- The maximum and minimum values that can be stored in an `int` variable are represented by the constants **`INT_MAX`** and **`INT_MIN`**, respectively, from the **header `<limits.h>`**.
- It's considered good practice to ensure that arithmetic calculations will not overflow before you perform them. In industrial-strength code, you should perform checks for all calculations that can result in overflow or underflow (p. 166).
- 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 compiler vendor will implement it.
- Microsoft implemented its own versions of `printf_s` and `scanf_s` before the C11 standard and 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.
- Many organizations have coding standards that require code to compile without warning messages. There are two ways to eliminate Visual C++'s `scanf` warnings. You can either start using `scanf_s` immediately or disable this warning message.

Self-Review Exercises

3.1 Fill-In the blanks in each of the following questions.

a) A procedure for solving a problem in terms of the actions to execute and the order in which the actions should execute is called a(n) ________.

b) Specifying the execution order of statements by the computer is called ________.

c) All programs can be written in terms of three types of control statements: ________, ________ and ________.

d) The ________ selection statement is used to execute one action when a condition is *true* and another action when that condition is *false*.

e) Several statements grouped together in braces (`{` and `}`) are called a(n) ________.

f) The ________ iteration statement specifies that a statement or group of statements is to be executed repeatedly while some condition remains *true*.

g) Iterating a specific number of times is called ________ iteration.

h) When it's not known in advance how many times a set of statements will be repeated, a(n) ________ value can be used to terminate the iteration.

3.2 Write four different C statements that each add 1 to integer variable `x`.

3.3 Write a single C statement to accomplish each of the following:

a) Multiply the variable `product` by 2 using the `*=` operator.

b) Multiply the variable `product` by 2 using the `=` and `*` operators.

c) Test whether the value of the variable `count` is greater than 10. If it is, print `"Count is greater than 10"`.

d) Calculate the remainder after `quotient` is divided by `divisor` and assign the result to `quotient`. Write this statement two different ways.

e) Print the value `123.4567` with two digits of precision. What value is printed?

f) Print the floating-point value `3.14159` with three digits to the right of the decimal point. What value is printed?

3.4 Write a C statement to accomplish each of the following tasks.

a) Define variable `x` to be of type `int` and set it to `1`.
b) Define variable `sum` to be of type `int` and set it to `0`.
c) Add variable `x` to variable `sum` and assign the result to variable `sum`.
d) Print `"The sum is: "` followed by the value of variable `sum`.

3.5 Combine the statements from Exercise 3.4 into a program that calculates the sum of the integers from 1 to 10. Use the `while` statement to loop through the calculation and increment statements. The loop should terminate when `x` becomes 11.

3.6 Write single C statements to perform each of the following tasks:

a) Input integer variable `x` with `scanf`. Use the conversion specification `%d`.
b) Input integer variable `y` with `scanf`. Use the conversion specification `%d`.
c) Set integer variable `i` to `1`.
d) Set integer variable `power` to `1`.
e) Multiply integer variable `power` by `x` and assign the result to `power`.
f) Increment variable `i` by `1`.
g) Test `i` to see if it's less than or equal to `y` in the condition of a `while` statement.
h) Output integer variable `power` with `printf`.

3.7 Write a C program that uses the statements in the preceding exercise to calculate `x` raised to the `y` power. The program should have a `while` iteration control statement.

3.8 Identify and correct the errors in each of the following:

a)
```
while (c <= 5) {
   product *= c;
   ++c;
```
b)
```
scanf("%.4f", &value);
```
c)
```
if (gender == 1) {
   puts("Woman");
}
else; {
   puts("Man");
}
```

3.9 What's wrong with the following `while` iteration statement (assume `z` has value 100), which is supposed to calculate the sum of the integers from 100 down to 1?

```
while (z >= 0) {
   sum += z;
}
```

Answers to Self-Review Exercises

3.1 a) Algorithm. b) Program control. c) Sequence, selection, iteration. d) `if...else`. e) Compound statement or block. f) `while`. g) Counter-controlled or definite. h) Sentinel.

3.2 See the answer below:

```
x = x + 1;
x += 1;
++x;
x++;
```

3.3 See the answers below:

a) `product *= 2;`

b) `product = product * 2;`

c)
```
if (count > 10) {
   puts("Count is greater than 10.");
}
```

d) `quotient %= divisor;`
   `quotient = quotient % divisor;`

e) `printf("%.2f", 123.4567);`
   `123.46` is displayed.

f) `printf("%.3f\n", 3.14159);`
   `3.142` is displayed.

3.4 See the answers below:

a) `int x = 1;`

b) `int sum = 0;`

c) `sum += x;` or `sum = sum + x;`

d) `printf("The sum is: %d\n", sum);`

3.5 See below.

```
// Calculate the sum of the integers from 1 to 10
#include <stdio.h>

int main(void) {
   int x = 1; // set x
   int sum = 0; // set sum

   while (x <= 10) { // loop while x is less than or equal to 10
      sum += x; // add x to sum
      ++x; // increment x
   } // end while

   printf("The sum is: %d\n", sum); // display sum
} // end main function
```

3.6 See the answers below:

a) `scanf("%d", &x);`

b) `scanf("%d", &y);`

c) `i = 1;`

d) `power = 1;`

e) `power *= x;`

f) `++i;`

g) `while (i <= y)`
h) `printf("%d", power);`

3.7 See below.

```c
// raise x to the y power
#include <stdio.h>

int main(void) {
   printf("%s", "Enter first integer: ");
   int x = 0;
   scanf("%d", &x); // read value for x from user
   printf("%s", "Enter second integer: ");
   int y = 0;
   scanf("%d", &y); // read value for y from user

   int i = 1;
   int power = 1; // set power

   while (i <= y) { // loop while i is less than or equal to y
      power *= x; // multiply power by x
      ++i; // increment i
   } // end while

   printf("%d\n", power); // display power
} // end main function
```

3.8 See the answers below:

a) Error: Missing the closing right brace of the `while` body.
Correction: Add closing right brace after the statement `++c;`.

b) Error: Precision used in a `scanf` conversion specification.
Correction: Remove `.4` from the conversion specification.

c) Error: Semicolon after the `else` part of the `if...else` statement results in a logic error. The second `puts` will always execute.
Correction: Remove the semicolon after `else`.

3.9 The value of the variable `z` is never changed in the `while` statement. Therefore, an infinite loop is created. To prevent the infinite loop, `z` must be decremented so that it eventually becomes `0`.

Exercises

3.10 Identify and correct the errors in each of the following. [*Note*: There may be more than one error in each piece of code.]

a)
```c
if (sales => 5000)
   puts("Sales are greater than or equal to $5000")
else
   puts("Sales are less than $5000");
```

b)
```
int x = 1, product = 0;
while (x <= 10); {
   product *= x;
   ++x;
}
```
c)
```
While (x <= 100)
   total =+ x;
   ++x;
```
d)
```
while (y < 10) {
   printf("%d\n", y);
}
```

3.11 Fill in the blanks in each of the following:

a) In ________ ________, statements execute one after the other in the order they are written.

b) ________ programs help you “think out” a program.

c) All programs can be written in terms of ________, ________, control structures.

d) Flowcharts are drawn using ________, ________, ________ and ________ connected by arrows called flowlines.

e) Flowlines indicate the ________ in which the actions are performed.

f) The ________ multiple-selection statement selects one among many options based on the value of an expression.

g) The ________ operator is C’s only ternary operator. It takes three operands, ________ a condition, ________ the value for the conditional expression if the condition is true, and ________ the value for the conditional expression if the condition is false.

h) The `if` statement is a ________ structure.

3.12 What does the following program print?

```
#include <stdio.h>

int main()
{
   int y;
   int x = 1;
   int total = 0;
   while (x <= 10) {
      y = x * x * x;
      printf("%d\n", y);
      total += y;
      ++x;
   } // end while

   printf("The Total is %d \n", total);
} // end main
```

3.13 Write a single pseudocode statement that indicates each of the following:

a) Display the message `"Enter your name:"`.
b) Assign the product of variables `a`, `b`, `c`, and `d` to variable `p`.
c) The following condition is to be tested in a conditional statement: if `x` is greater than `y`, then `x` is assigned the value `10`, otherwise `x` is assigned the value `20`.
d) Obtain values for variables `a`, `b`, `c`, and `d` from the keyboard.

3.14 Formulate a pseudocode algorithm for each of the following:

a) Obtain three numbers from the keyboard, compute their product, and display the result.
b) Obtain two numbers from the keyboard, and determine and display which (if either) is the smaller of the two numbers.
c) Obtain a series of positive numbers from the keyboard, and determine and display their average. Assume that the user types the sentinel value '`-1`' to indicate "end of data entry."

3.15 State which of the following are *true*, and which are *false*. If a statement is *false*, explain why.

a) An algorithm is a procedure for solving a problem in terms of the actions to be executed, without specifying the order of the actions.
b) Unless directed otherwise, the computer automatically executes C statements in sequence.
c) The `if...else` double-selection statement selects a single action.
d) A logic error affects the program when the program is compiled. It does not fail or terminate the program prematurely.
e) You can determine your platform's maximum unsigned `int` value with the constant `UINT_MAX` from `<limits.h>`.

For Exercises 3.16–3.20, perform each of these steps:

1. Read the problem statement.
2. Formulate the algorithm using pseudocode and top-down, stepwise refinement.
3. Write a C program.
4. Test, debug, and execute the C program.

3.16 *(Sales Tax)* Sales tax is collected from buyers and remitted to the government. A retailer must file a monthly sales tax report that lists the sales for the month and the amount of sales tax collected, at both the county and state levels. Develop a program that will input the total collections for a month, calculate the sales tax on the collections, and display the county and state taxes. Assume that states have a 4% sales tax, and counties have a 5% sales tax. Here is a sample input/output dialog:

```
Enter total amount collected (-1 to quit): 45678
Enter name of month: January
Total Collections: $45678.00
Sales: $41906.42
```

```
County Sales Tax: $2095.32
State Sales Tax: $1676.26
Total Sales Tax Collected: $3771.58

Enter total amount collected (-1 to quit): 98000
Enter name of month: February
Total Collection: $98000
Sales: $89908.26
County Sales Tax: $4495.41
State Sales Tax: $3596.33
Total Sales Tax Collected: $8091.74
```

3.17 *(Mortgage Calculator)* Develop a C program to calculate the interest accrued on a bank customer's mortgage. For each customer, the following facts are available:

a) The mortgage amount
b) The mortgage term
c) The interest rate

The program should input each fact, calculate the total interest payable, and add it to the mortgage amount to get the total amount payable. It should calculate the required monthly payment by dividing the total amount payable by the number of months in the mortgage term. The program should display the required monthly payment. The program should process each customer's account at a time. Here is a sample input/output dialog:

```
Enter mortgage amount in dollars:
Enter Mortgage term (in years):
Enter Interest rate:
The Monthly Payable Interest is:

Enter mortgage amount in dollars:
Enter Mortgage term (in years):
Enter Interest rate:
The Monthly Payable Interest is:
```

3.18 *(Sales-Commission Calculator)* A large chemical company pays its salespeople on a commission basis. The salespeople receive $200 per week plus 9% of their gross sales for that week. For example, a salesperson who sells $5000 worth of chemicals in a week receives $200 plus 9% of $5000, or a total of $650. Develop a program that will use `scanf` to input each salesperson's gross sales for last week and calculate and display that salesperson's earnings. Process one salesperson's figures at a time. Here is a sample input/output dialog:

```
Enter sales in dollars (-1 to end): 5000.00
Salary is: $650.00

Enter sales in dollars (-1 to end): 1234.56
Salary is: $311.11

Enter sales in dollars (-1 to end): -1
```

3.19 *(Interest Calculator)* The simple interest on a loan is calculated by the formula

```
interest = principal * rate * days / 365;
```

The preceding formula assumes that `rate` is the annual interest rate, so it divides by 365 (days per year). Develop a program that uses `scanf` to input `principal`, `rate` and `days` for several loans, and will calculate and display the simple interest for each loan, using the preceding formula. Here is a sample input/output dialog:

```
Enter loan principal (-1 to end): 1000.00
Enter interest rate: .1
Enter term of the loan in days: 365
The interest charge is $100.00

Enter loan principal (-1 to end): 1000.00
Enter interest rate: .08375
Enter term of the loan in days: 224
The interest charge is $51.40

Enter loan principal (-1 to end): -1
```

3.20 *(Salary Calculator)* Develop a program that will determine the gross pay for each of several employees. The company pays "straight time" for the first 40 hours worked by each employee and pays "time-and-a-half" for all hours worked in excess of 40 hours. You're given a list of the company's employees, the number of hours each worked last week and each employee's hourly rate. Your program should use `scanf` to input this information for each employee and determine and display the employee's gross pay. Here is a sample input/output dialog:

```
Enter # of hours worked (-1 to end): 39
Enter hourly rate of the worker ($00.00): 10.00
Salary is $390.00

Enter # of hours worked (-1 to end): 40
Enter hourly rate of the worker ($00.00): 10.00
Salary is $400.00

Enter # of hours worked (-1 to end): 41
Enter hourly rate of the worker ($00.00): 10.00
Salary is $415.00

Enter # of hours worked (-1 to end): -1
```

3.21 *(Preincrementing vs. Postincrementing)* Write a program that demonstrates the difference between preincrementing and postincrementing using the increment operator `++`.

3.22 *(Checking if a Number is Prime)* A prime number is any natural number greater than 1 that is divisible only by 1 and by itself. Write a C program that reads an integer, and determines whether it is a prime number or not.

3.23 *(Find the Largest Number)* Finding the largest number (i.e., the maximum of a group of numbers) is used frequently in computer applications. For example, a program that determines the winner of a sales contest would input the number of units sold by each salesperson. The salesperson who sells the most units wins the contest. Write a pseudocode program and then a program that uses `scanf` to input a series of 10 non-negative numbers and determines and prints the largest of the numbers. Your program should use three variables:

a) `counter`—A counter to count to 10 (i.e., to keep track of how many numbers have been input and to determine when all 10 numbers have been processed).
b) `number`—The current number input to the program.
c) `largest`—The largest number found so far.

3.24 *(Tabular Output)* Write a program that uses looping to print the following table of values. Use the tab escape sequence, `\t`, in the `printf` statement to separate the columns with tabs.

```
N         N2        N3        N4

1         1         1         1
2         4         8         16
3         9         27        81
4         16        64        256
5         25        125       625
6         36        216       1296
7         49        343       2401
8         64        512       4096
9         81        729       6561
10        100       1000      10000
```

3.25 *(Tabular Output)* Write a program that utilizes looping to produce the following table of values:

```
N        N+3      N+6      N+9

7        10       13       63
14       17       20       126
21       24       27       189
28       31       34       252
35       38       41       315
```

3.26 *(Find the Two Largest Numbers)* Using an approach similar to Exercise 3.23, find the *two* largest values of the 10 numbers. You may input each number only *once*.

3.27 *(Validating User Input)* Modify the program in Figure 3.6 to validate its inputs. For each input, if the value is other than 1 or 2, keep looping until the user enters a correct value.

3.28 What does the following program print?

```c
#include <stdio.h>

int main(void) {
   int count = 1; // initialize count

   while (count <= 10) { // loop 10 times
```

```
      // output line of text
      puts((count % 2) ? "****" : "++++++++");
      ++count; // increment count
   } // end while
} // end function main
```

3.29 What does the following program print?

```
#include <stdio.h>

int main(void)
{
   int outer_count = 1; // initialize count
   while (outer_count <= 10) { // loop 10 times
      int inner_count = 1;
      while (inner_count <= outer_count) {
         printf("* ");
         inner_count++;
      } // end inner while
      printf("\n");
      outer_count++;
   } // end outer while
} // end main
```

3.30 *(Dangling-Else Problem)* Determine the output for each of the following when `x` is `9` and `y` is `11`, and when `x` is `11` and `y` is `9`. The compiler ignores the indentation in a C program. Also, the compiler always associates an `else` with the previous `if` unless told to do otherwise by the placement of braces `{}`. On first glance, you may not be sure which `if` an `else` matches, so this is referred to as the "dangling-else" problem. We eliminated the indentation from the following code to make the problem more challenging. [*Hint*: Apply indentation conventions you have learned.]

a)
```
if (x < 10)
if (y > 10)
puts("*****");
else
puts("#####");
puts("$$$$$");
```

b)
```
if (x < 10) {
if (y > 10)
puts("*****");
}
else {
puts("#####");
puts("$$$$$");
}
```

3.31 *(Another Dangling-Else Problem)* Modify the following code to produce the output shown. Use proper indentation techniques. You may not make any changes other than inserting braces. The compiler ignores the indentation in a program. We eliminated the indentation from the following code to make the problem more challenging. [*Note*: It's possible that no modification is necessary.]

```
if (y == 8)
if (x == 5)
puts("@@@@@");
else
puts("#####");
puts("$$$$$");
puts("&&&&&");
```

a) Assuming x = 5 and y = 8, the following output is produced.

```
@@@@@
$$$$$
&&&&&
```

b) Assuming x = 5 and y = 8, the following output is produced.

```
@@@@@
```

c) Assuming x = 5 and y = 8, the following output is produced.

```
@@@@@
&&&&&
```

d) Assuming x = 5 and y = 7, the following output is produced.

```
#####
$$$$$
&&&&&
```

3.32 *(Square of Asterisks)* Write a program that reads in the side of a square and then prints that square out of asterisks. Your program should work for squares of all side sizes between 1 and 20. For example, if your program reads a size of 4, it should print

```
****
****
****
****
```

3.33 *(Hollow Square of Asterisks)* Modify the program you wrote in the preceding exercise so that it prints a hollow square. For example, if your program reads a size of 5, it should print

```
*****
*   *
*   *
*   *
*****
```

3.34 *(Floyd's Triangle)* Floyd's Triangle is a right-angled triangular array of natural numbers. It is defined by filling rows with consecutive integers. Thus, row 1 will have the number 1, row 2 will have the numbers 2 and 3, and so on. Write a program that draws a 10-line Floyd's triangle. An outer loop can control the number of lines to be printed and an inner loop can ensure that each row contains the correct number of integers.

3.35 *(Printing the Decimal Equivalent of a Binary Number)* Input a binary integer (5 digits or fewer) containing only 0s and 1s and print its decimal equivalent. [*Hint*: Use the remainder and division operators to pick off the "binary" number's digits one at a time from right-to-left. Just as in the decimal number system, in which the rightmost digit has a positional value of 1, and the next digit left has a positional value of 10, then 100, then 1000, and so on, in the binary number system the rightmost digit has a positional value of 1, the next digit left has a positional value of 2, then 4, then 8, and so on. Thus the decimal number 234 can be interpreted as 4 * 1 + 3 * 10 + 2 * 100. The decimal equivalent of binary 1101 is 1 * 1 + 0 * 2 + 1 * 4 + 1 * 8 or 1 + 0 + 4 + 8 or 13.]

3.36 *(Armstrong Numbers)* Armstrong numbers are numbers that are equal to the sum of their digits raised to power of the number of digits in them. The number 153, for example, equals $1^3 + 5^3 + 3^3$. Thus, it is an Armstrong number. Write a program to display all three-digit Armstrong numbers.

3.37 *(Detecting Multiples of a Number)* Write a program that prints 500 dollar-signs ($) one after the other, separated by a space. After every fiftieth dollar sign, the program should print a newline character. [*Hint*: Count from 1 to 500. Use the remainder operator to recognize when the counter reaches a multiple of 50.]

3.38 *(Counting 9s)* Write a program that reads an integer (5 digits or fewer) and determines and prints how many digits in the integer are 9s.

3.39 *(Checkerboard Pattern of Asterisks)* Write a program that displays the following checkerboard pattern:

```
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
* * * * * * * *
 * * * * * * * *
```

Your program must use only three output statements, one of each of the following forms:

```
printf("%s", "* ");
printf("%s", " ");
puts(""); // outputs a newline
```

3.40 *(Multiples of 3 with an Infinite Loop)* Write a program that keeps printing the multiples of the integer 3, namely 3, 9, 27, 91, 273, and so on. Your loop should not terminate (i.e., you should create an infinite loop). What happens when you run this program?

3.41 *(Diameter, Circumference and Area of a Circle)* Write a program that reads the radius of a circle (as a `double` value) and computes and prints the diameter, the circumference and the area. Use the value 3.14159 for π.

3.42 What's wrong with the following statement? Rewrite it to accomplish what the programmer was probably trying to do.

```
printf("%d", --(x * y));
```

3.43 *(Sides of a Triangle)* Write a program that reads three nonzero integer values and determines and prints whether they could represent the sides of a triangle.

3.44 *(Sides of a Right Triangle)* Write a program that reads three nonzero integers and determines and prints whether they could be the sides of a right triangle.

3.45 *(Factorial)* The factorial of a non-negative integer n is written $n!$ (pronounced "n factorial") and is defined as follows:

$n! = n \cdot (n - 1) \cdot (n - 2) \cdot \ldots \cdot 1$ (for values of n greater than or equal to 1)

and

$n! = 1$ (for $n = 0$).

For example, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$, which is 120.

a) Write a program that reads a non-negative integer and computes and prints its factorial.

b) Write a program that estimates the value of the mathematical constant e by using the formula:

$$e = 1 + \frac{1}{1!} + \frac{1}{2!} + \frac{1}{3!} + \ldots$$

c) Write a program that computes the value of e^x by using the formula

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots$$

3.46 *(World Population Growth)* World population has grown considerably over the centuries. Continued growth could eventually challenge the limits of breathable air, drinkable water, arable land, and other limited resources. There's evidence that growth has been slowing in recent years, and that world population could peak some time this century, then start to decline.

For this exercise, research world population growth issues. This is a controversial topic, so be sure to investigate various viewpoints. Get estimates for the current world population and its growth rate. Write a program that calculates world population growth each year for the next 100 years, *using the simplifying assumption that the current growth rate will stay constant.* Print the results in a table. The first column should display the year from 1 to 100. The second column should display the anticipated world population at the end of that year. The third column should display the numerical increase in the world population that would occur that year. Using your results, determine the years in which the population would become double and eventually quadruple what it is today.

3.47 *(Enforcing Privacy with Cryptography)* The explosive growth of Internet communications and data storage on Internet-connected computers has greatly increased privacy concerns. The field of cryptography is concerned with coding data to make it difficult (and hopefully—with the most advanced schemes—impossible) for unauthorized users to read. In this exercise, you'll investigate a simple scheme for *encrypting* and *decrypting* data. A company that wants to send data over the Internet has asked you to write a program that will encrypt it so that it may be transmitted more securely. All the data is transmitted as four-digit integers. Your application should read a four-digit integer entered by the user and *encrypt* it as follows: Replace each digit with the result of adding 7 to the digit and getting the remainder after dividing the new value by 10. Then swap the first digit with the third, and swap the second digit with the fourth. Then print the encrypted integer. Write a separate application that inputs an encrypted four-digit integer and *decrypts* it (by reversing the encryption scheme) to form the original number. [*Optional reading project:* In industrial-strength applications, you'll want to use much stronger encryption techniques than presented in this exercise. Research "public-key cryptography" in general and the PGP (Pretty Good Privacy) specific public-key scheme. You may also want to investigate the RSA scheme, which is widely used in industrial-strength applications.]