Chapter 5:
Repetition and Loop Statements

Problem Solving & Program Design in C

Eighth Edition

By Jeri R. Hanly &
Elliot B. Koffman

Addison Wesley is
an imprint of

PEARSON

Outline

5.1 REPETITION IN PROGRAMS

5.2 COUNTING LOOPS AND THE WHILE STATEMENT

5.3 COMPUTING A SUM OR A PRODUCT IN A LOOP

5.4 THE FOR STATEMENT

5.5 CONDITIONAL LOOPS

5.6 LOOP DESIGN

5.7 NESTED LOOPS

5.8 THE DO-WHILE STATEMENT AND FLAG-CONTROLLED LOOPS

5.9 PROBLEM SOLVING ILLUSTRATED

- *CASE STUDY: COLLECTING AREA FOR SOLAR-HEATED HOUSE*

5.10 HOW TO DEBUG AND TEST PROGRAMS

5.11 COMMON PROGRAMMING ERRORS

Introduction

- Three types of program structure
  - *sequence*
  - *selection*
  - *repetition* ← This chapter
- The repetition of steps in a program is called **loop**
- This chapter discuss three C loop control statement
  - ***while***
  - ***for***
  - ***do-while***

5.1 Repetition in programs

- **Loop**
  - A control structure that repeats a group of steps in a program
- **Loop body**
  - The statements that are repeated in the loop
- Three questions to determine whether loops will be required in the general algorithm: **(Figure5.1)**

1. Were there any steps I repeated as I solved the problem? If so, which ones?
2. If the answer to question 1 is yes, did I know in advance how many times to repeat the steps?
3. If the answer to question 2 is no, how did I know how long to keep repeating the steps?

Figure 5.1 Flow Diagram of Loop Choice Process

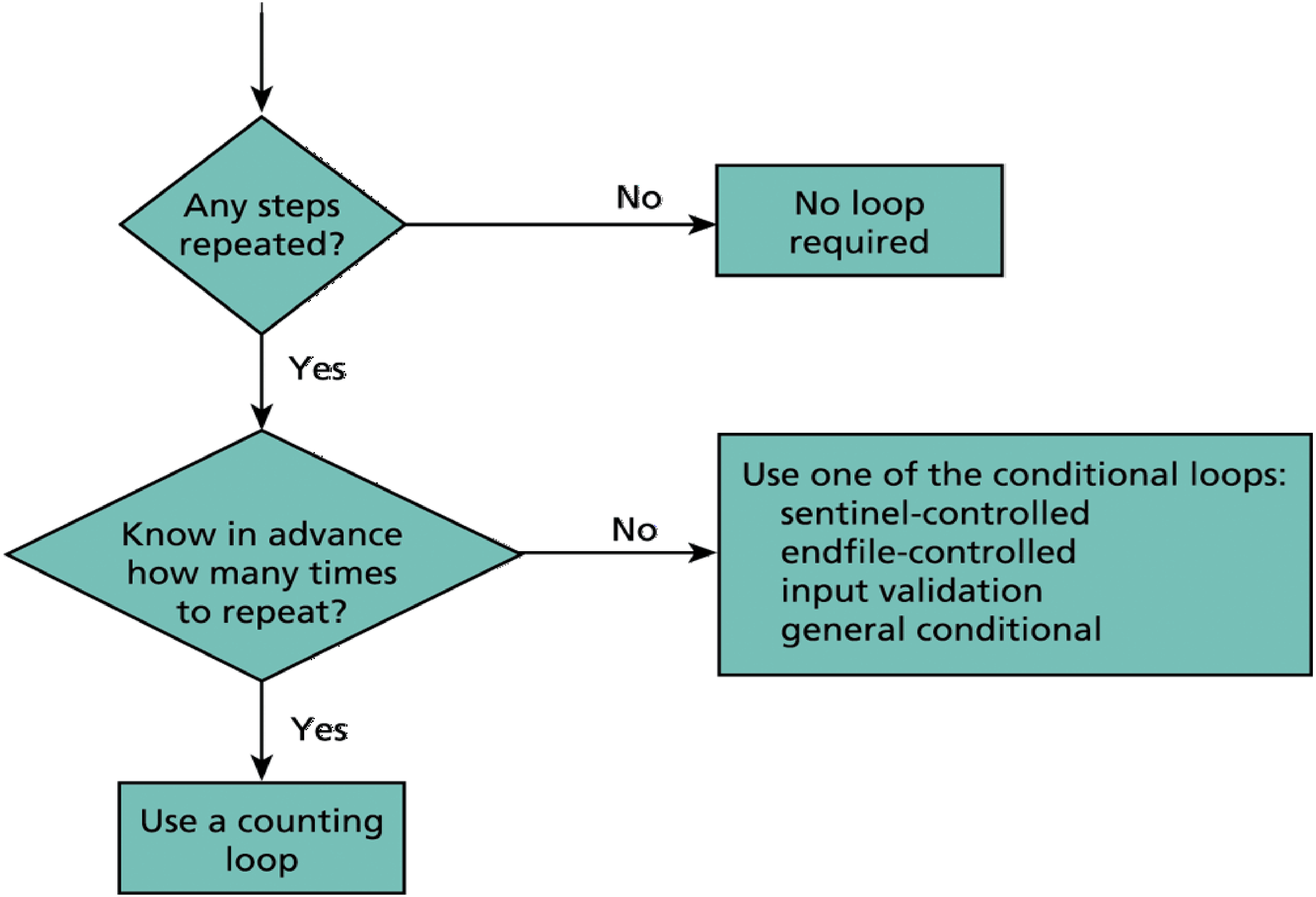


Table 5.1 Comparison of Loop Kinds

Kind	When Used	C Implementation Structure	Section Containing an Example

Counting loop	We can determine before loop execution how many loop executions will be needed to solve the problem	while
		for
			5.2
			5.4

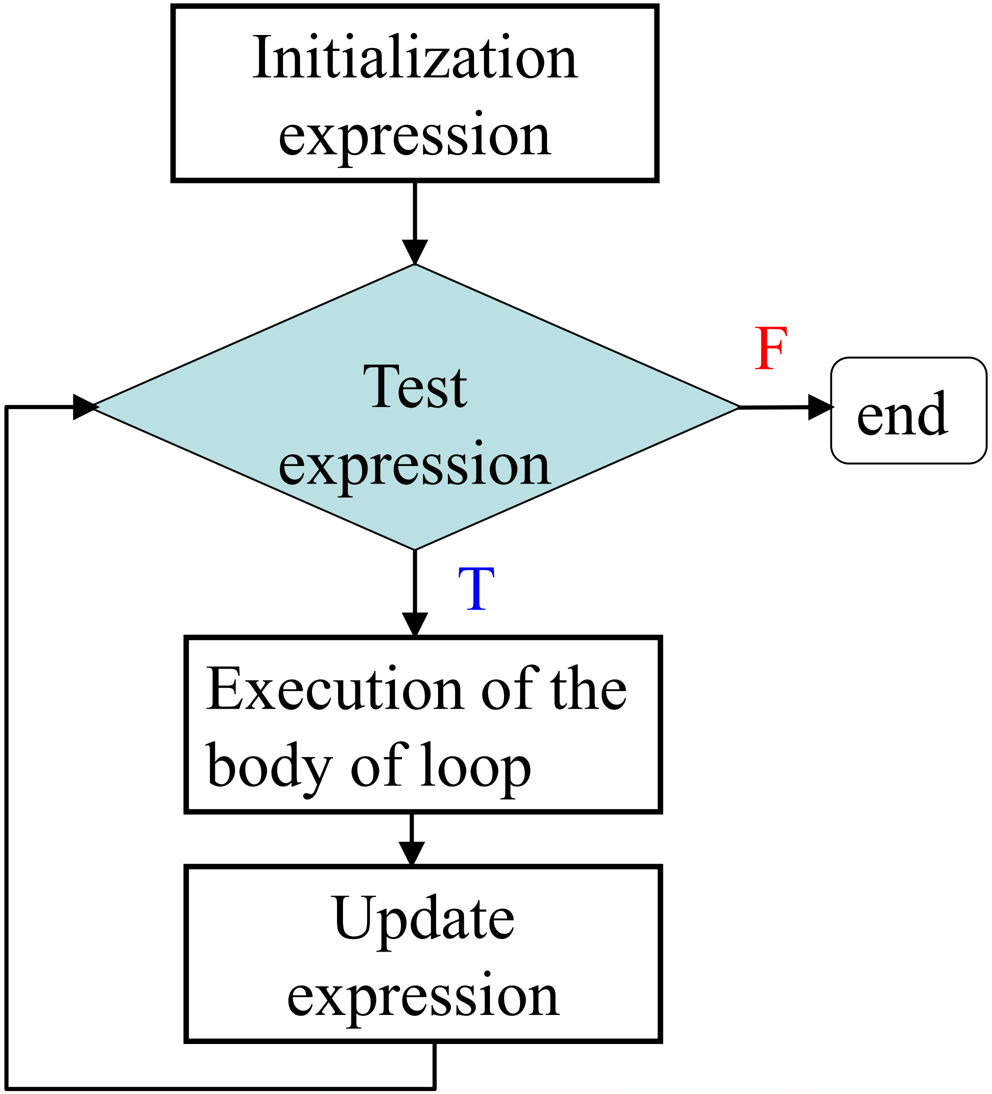
Sentinel-controlled loop	Input of a list of data of any length ended by a special value	while, for	5.6
Endfile-controller loop	Input of a single list of data of any length from a data file	while, for	5.6
Input validation loop	Repeated interactive input of a data value until a value within the valid range is entered	do-while	5.8
General conditional loop	Repeated processing of data until a desired condition is met	while, for	5.5

5.4 The *for* Statement

- Three loop control components
  - – Initialization of the loop variable
  - – Test of the loop repetition condition
  - – Change (update) of the loop control variable
- Using a for statement in a counting loop.

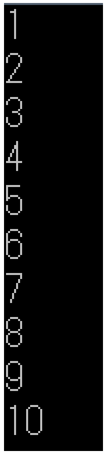
  (Figure 5.5)

Flowchart



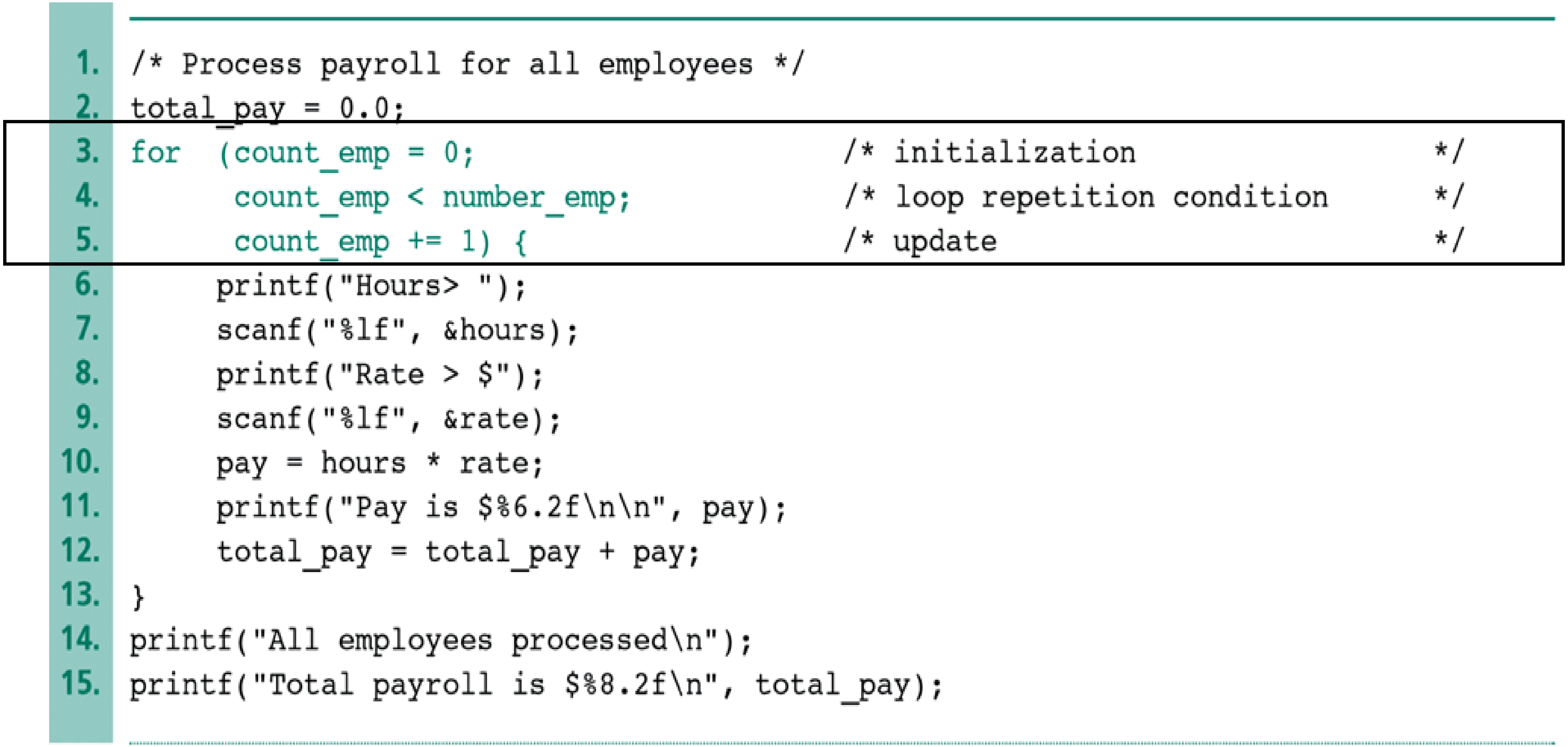
Example: print 1 to 10

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
   int i;

   for(i=1; i<=10; i++)
   {
        printf(“%d\n”, i);
    }
  return (0);
}
```



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

Figure 5.5
Using a for Statement in a Counting Loop



```
/* Process payroll for all employees */
total_pay = 0.0;
for  (count_emp = 0;                         /* initialization              */
      count_emp < number_emp;                /* loop repetition condition   */
      count_emp += 1) {                      /* update                      */
     printf("Hours> ");
     scanf("%lf", &hours);
     printf("Rate > $");
     scanf("%lf", &rate);
     pay = hours * rate;
     printf("Pay is $%6.2f\n\n", pay);
     total_pay = total_pay + pay;
}
printf("All employees processed\n");
printf("Total payroll is $%8.2f\n", total_pay);
```

Compound Assignment Operators

- Assignment statements of the form
  - – variable = variable op expression
    - Where op is a C arithmetic operator

  ⇒ variable op = expression

  – Table 5.3

  Ex. time = time – 1; ⇒ time **-=** 1;

Compound Assignment Operators

Simple Assignment Operators	Compound Assignment Operators
count_emp = count_emp + 1;	count_emp += 1;
time = time -1;	time -= 1;
product = product * item;	product *= item;
total = total / number;	total /= number;
n = n % (x+1);	n %= x+1;

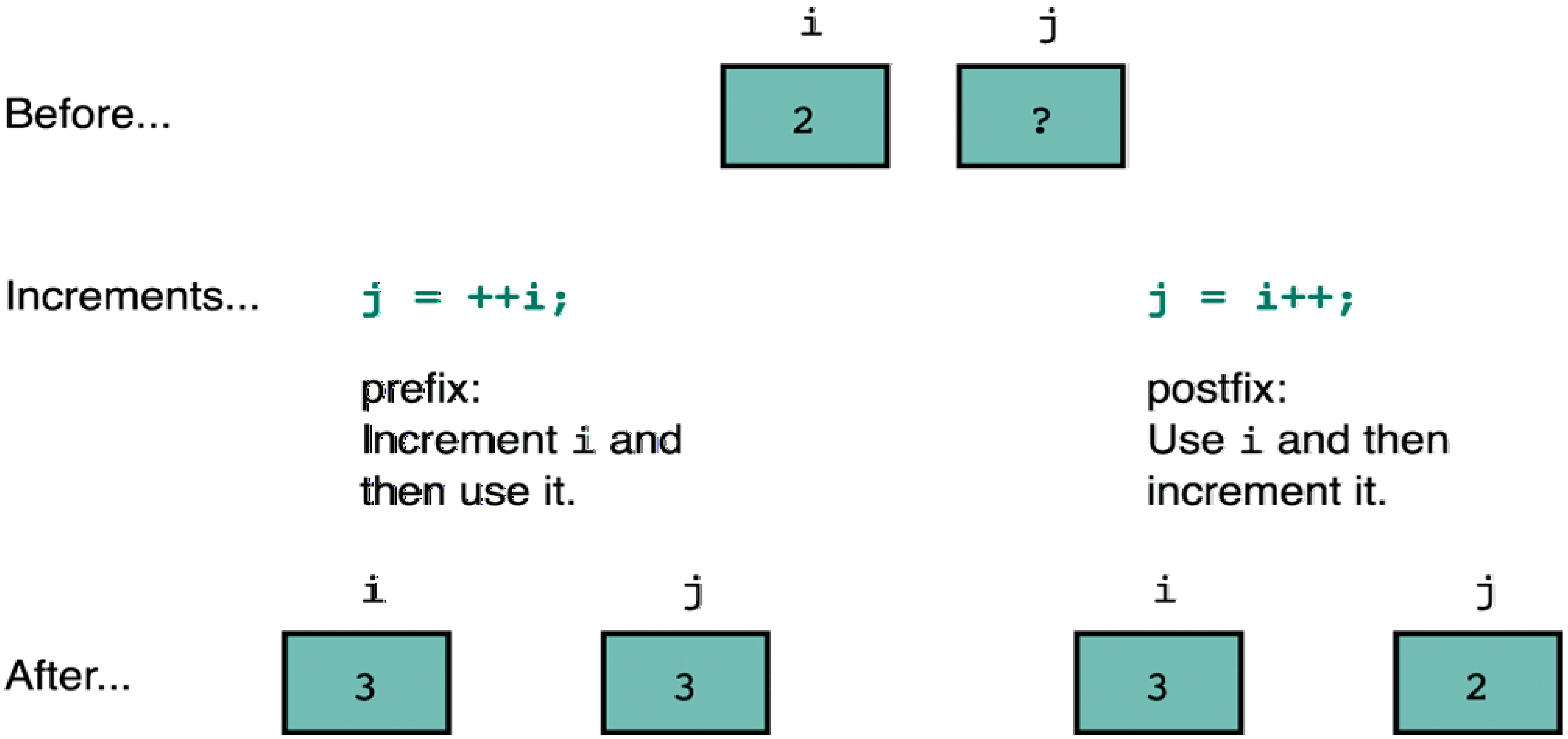
Syntax of *for* Statement

- Syntax: **for (initialization expression;**
  loop repetition condition;
  update expression)
  statement

- Example: /* Display N asterisks */

```
for (count_star = 0; count_star <N; count_star += 1)
      printf(“*”);
```

Increment and Decrement Operators



Before...

Increments...

`j = ++i;`

prefix:
Increment `i` and
then use it.

`j = i++;`

postfix:
Use `i` and then
increment it.

After...

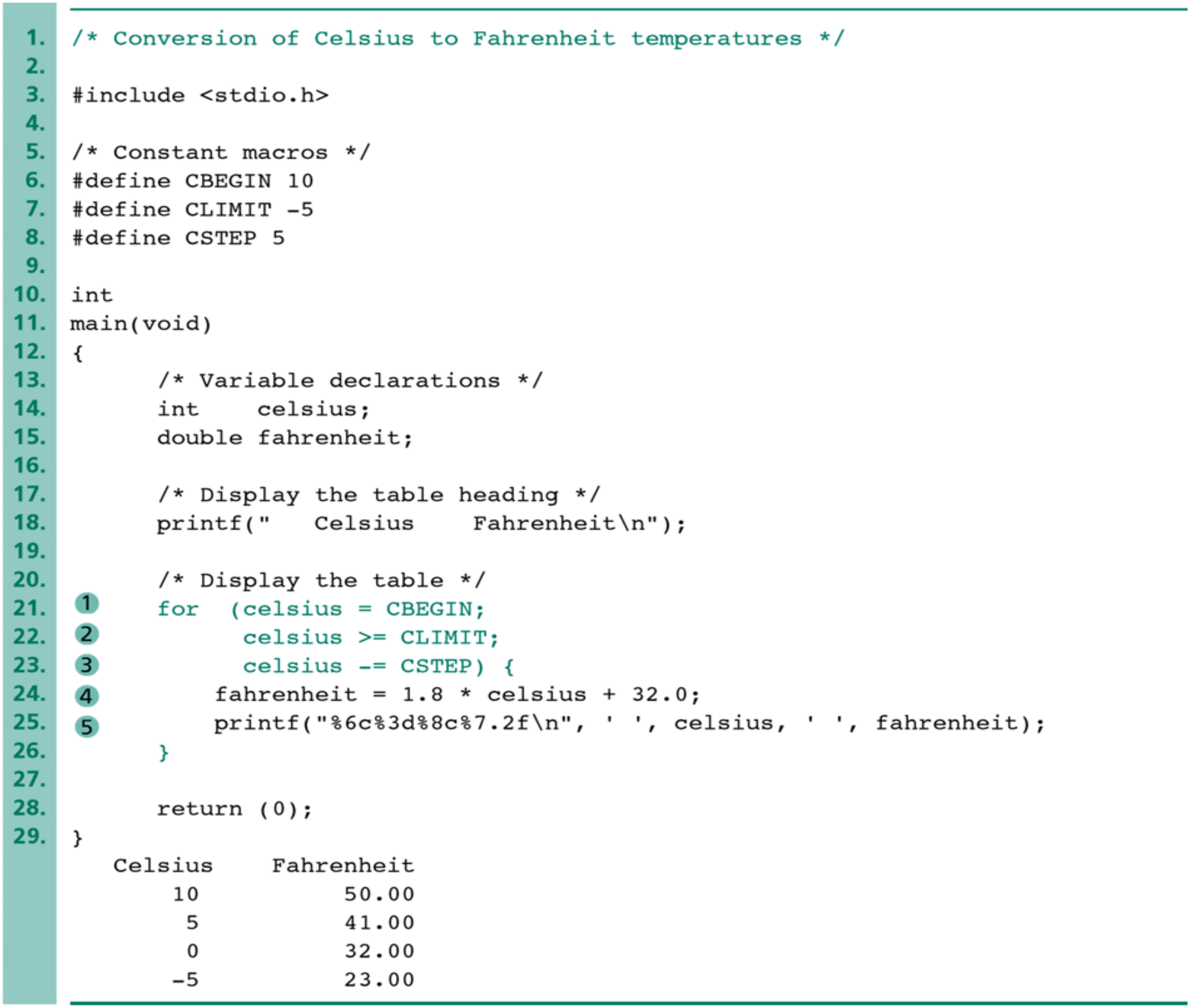
Figure 5.6 Comparison of Prefix and Postfix Increments

Increments and Decrements Other Than 1

- Example 5.4

  Use a loop that counts down by five to display a Celsius-to-Fahrenheit conversion table.

  (Figure 5.8)



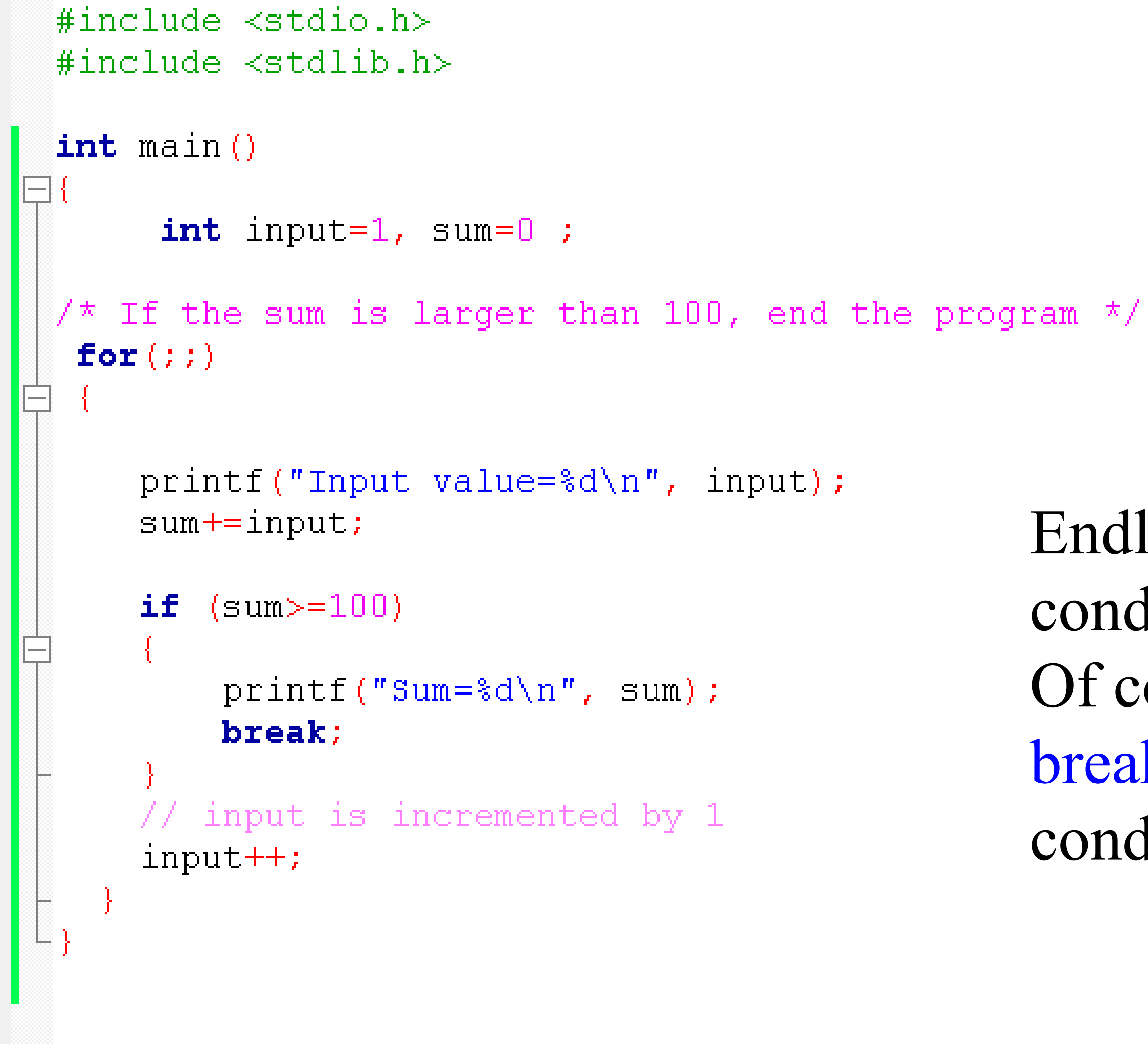
```
/* Conversion of Celsius to Fahrenheit temperatures */

#include <stdio.h>

/* Constant macros */
#define CBEGIN 10
#define CLIMIT -5
#define CSTEP 5

int
main(void)
{
      /* Variable declarations */
      int    celsius;
      double fahrenheit;

      /* Display the table heading */
      printf("   Celsius    Fahrenheit\n");

      /* Display the table */
①     for  (celsius = CBEGIN;
②           celsius >= CLIMIT;
③           celsius -= CSTEP) {
④         fahrenheit = 1.8 * celsius + 32.0;
⑤         printf("%6c%3d%8c%7.2f\n", ' ', celsius, ' ', fahrenheit);
      }

      return (0);
}
   Celsius    Fahrenheit
       10           50.00
        5           41.00
        0           32.00
       -5           23.00
```

Figure 5.8 Displaying a Celsius-to-Fahrenheit Conversion

5.5 Conditional Loops

- Not be able to determine the exact number of loop repetitions before loop execution begins
  - Ex.
    - Print an initial prompting message
    - Get the number of observed values
    - While the sum of values is higher than a threshold
      - ➢ The task is done and break

Endless Lop

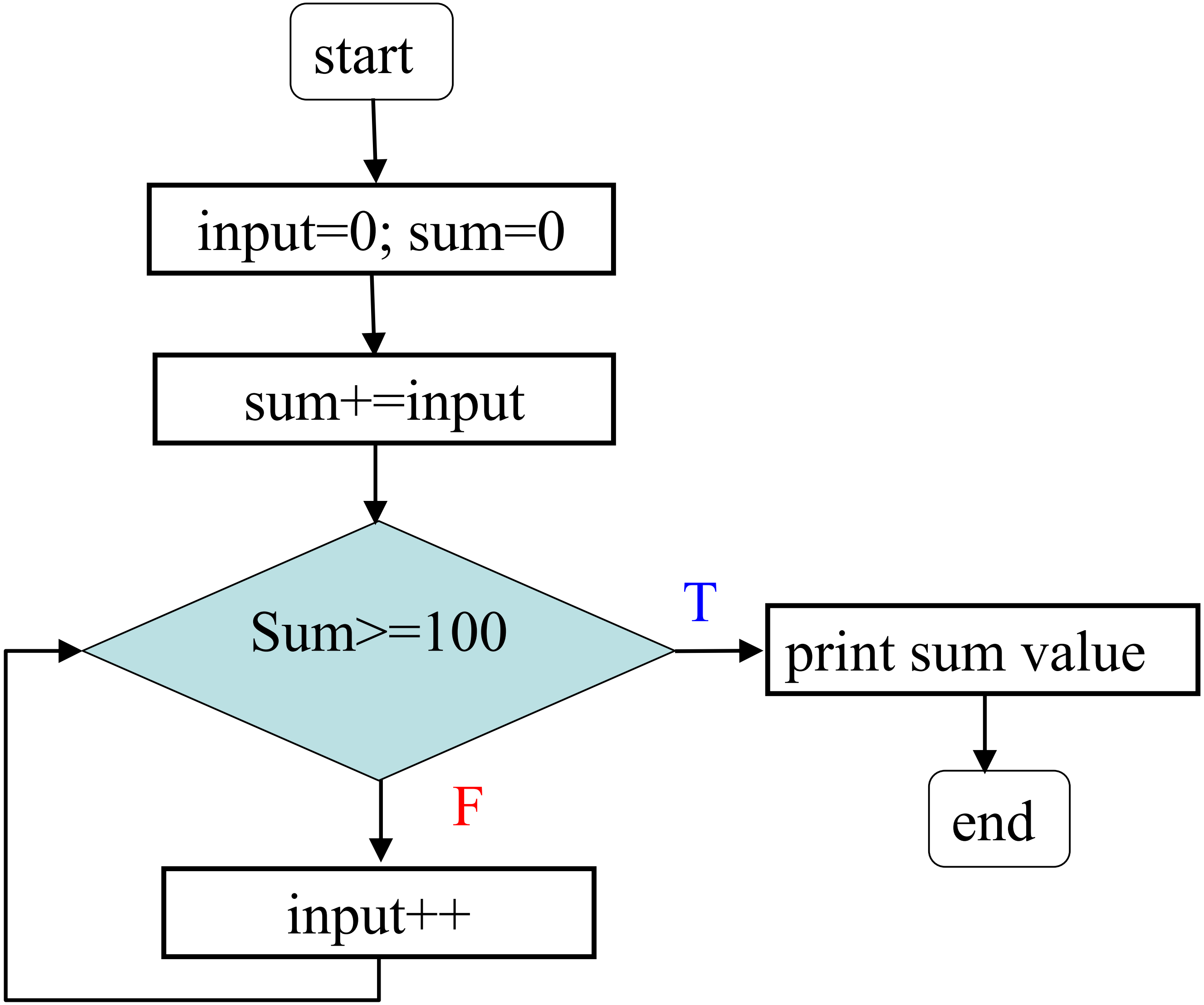


```
#include <stdio.h>
#include <stdlib.h>

int main()
{
     int input=1, sum=0 ;

/* If the sum is larger than 100, end the program */
 for(;;)
 {

    printf("Input value=%d\n", input);
    sum+=input;

    if (sum>=100)
    {
        printf("Sum=%d\n", sum);
        break;
    }
    // input is incremented by 1
    input++;
  }
}
```

for (; ;)

Endless loop can be used for conditional loop.
Of course, you should use **if** to break the loop once the condition is satisfied.

Flowchart


start
input=0; sum=0
sum+=input
Sum>=100
T
print sum value
end
F
input++

Practice

- Write a C code to computes the factorial of an integer represented by the formal parameter n.
  - Product = 1×2 × 3 × … × n
  - The condition to end the program is : product > 10000.

The *while* Statement

- Loop repetition condition
  - The condition that controls loop repetition
- Loop control variable
  - The variable whose value controls loop repetition
- The loop control variable must be
  - Initialization
  - Testing
  - Updating

Syntax of the while Statement

- Syntax： **while (*loop repetition condition*)**
  statement
- Example：

```
/* Display N asterisks. */
  count_star = 0；// initialization
   while (count_star < N) {
           printf(“*”)；
         count_star = count_star + 1；// update
            }
```

do while

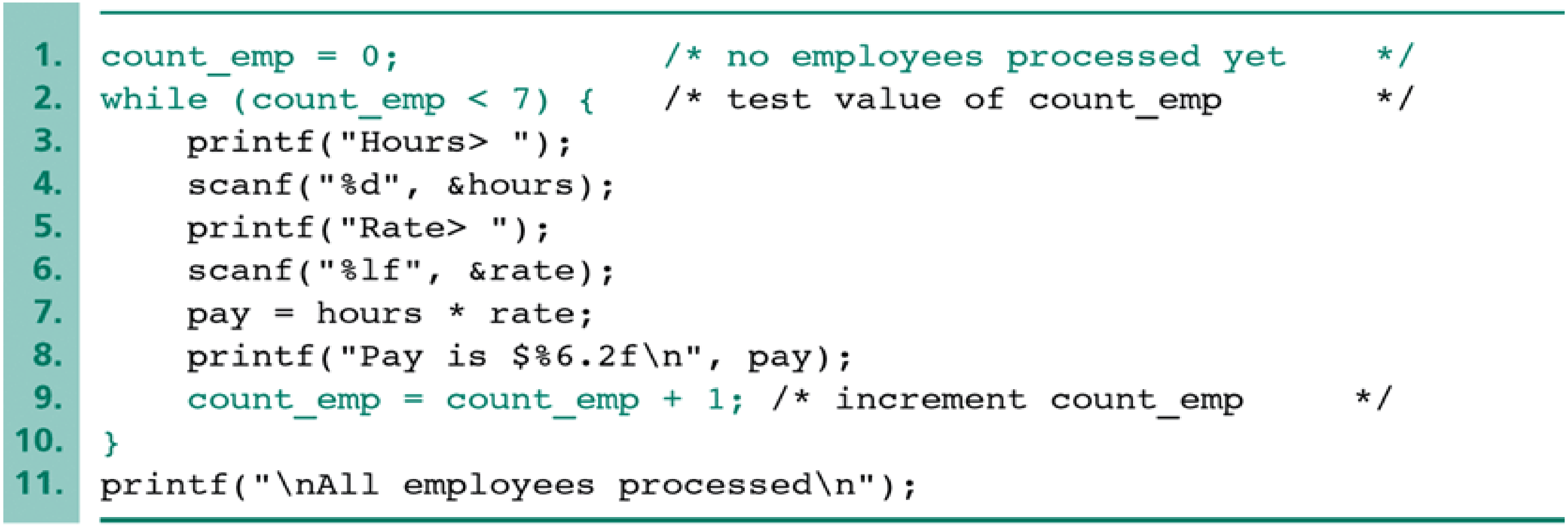
Syntax of *for* Statement

- Syntax: **for (initialization expression;**
  loop repetition condition;
  update expression)
  statement

- Example: /* Display N asterisks */

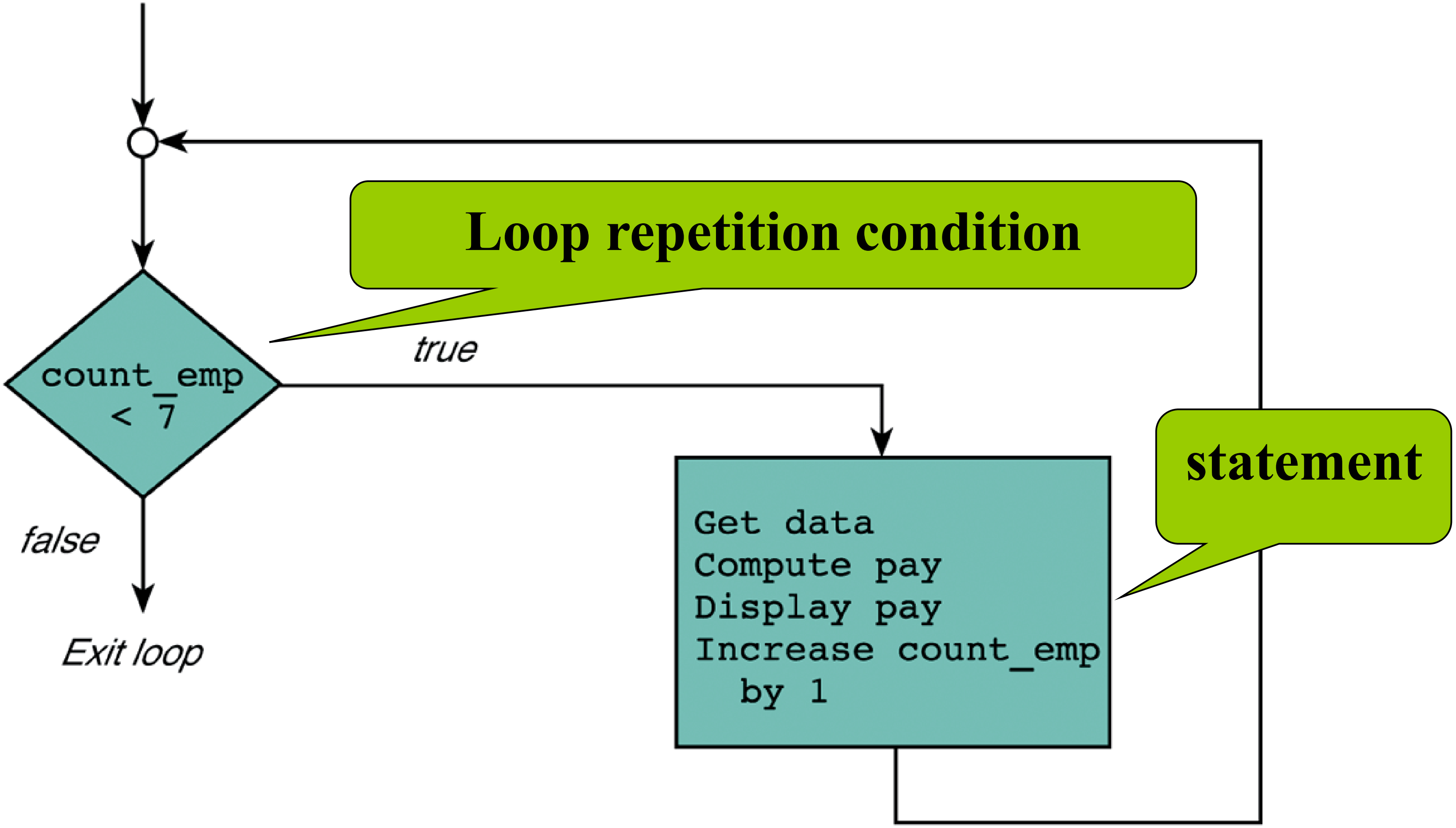
```
for (count_star = 0; count_star <N; count_star += 1)
    printf("*");
```

Figure 5.2 Program Fragment with a Loop



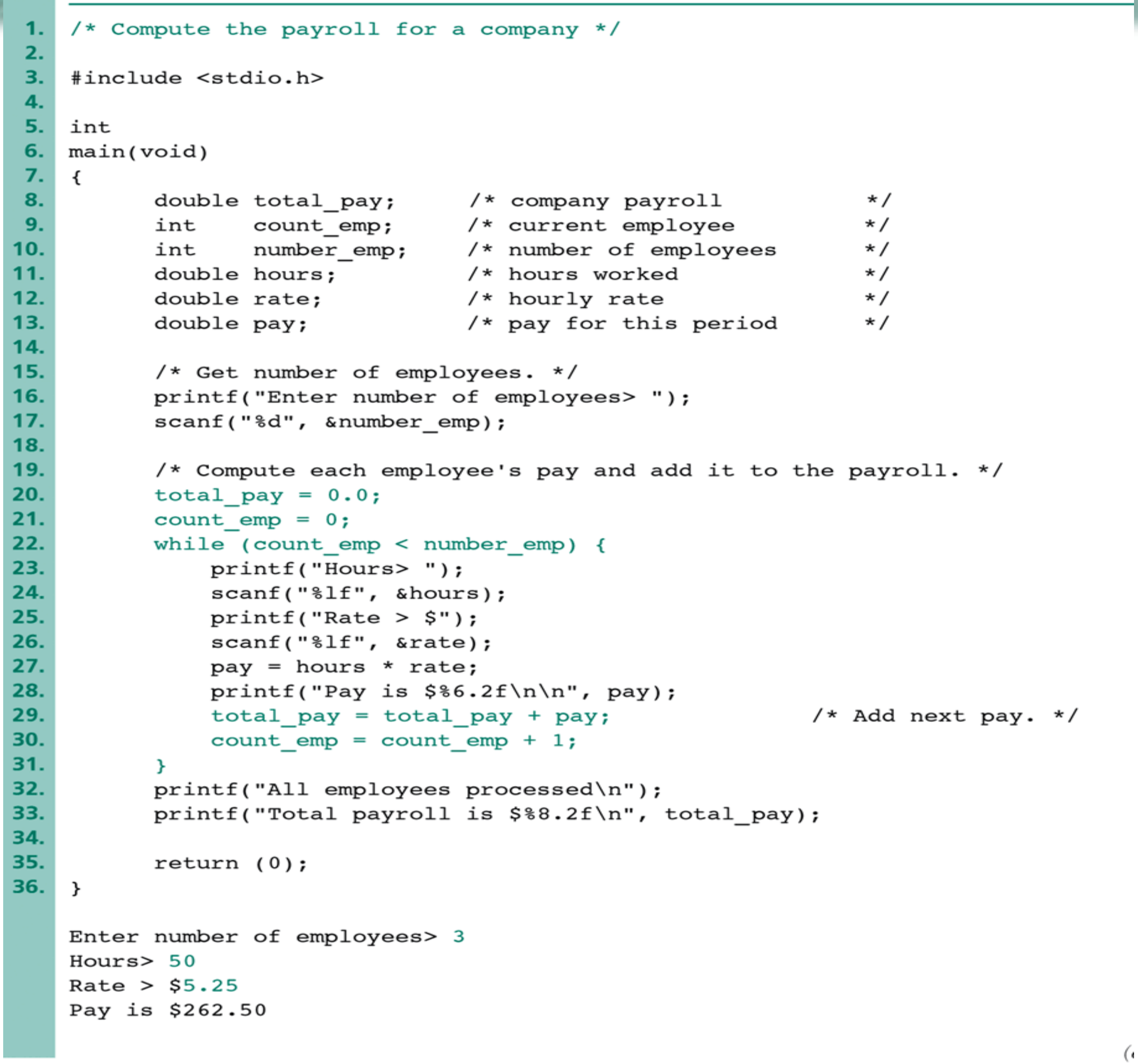
```
count_emp = 0;                /* no employees processed yet    */
while (count_emp < 7) {       /* test value of count_emp       */
    printf("Hours> ");
    scanf("%d", &hours);
    printf("Rate> ");
    scanf("%lf", &rate);
    pay = hours * rate;
    printf("Pay is $%6.2f\n", pay);
    count_emp = count_emp + 1; /* increment count_emp         */
}
printf("\nAll employees processed\n");
```

Flowchart for a while Loop



5.3 Computing a Sum or a Product in a Loop

- accumulator
  - a variable used to store a value being computed in increments during the execution of a loop
- Fig.5.4 Program to Compute Company Payroll

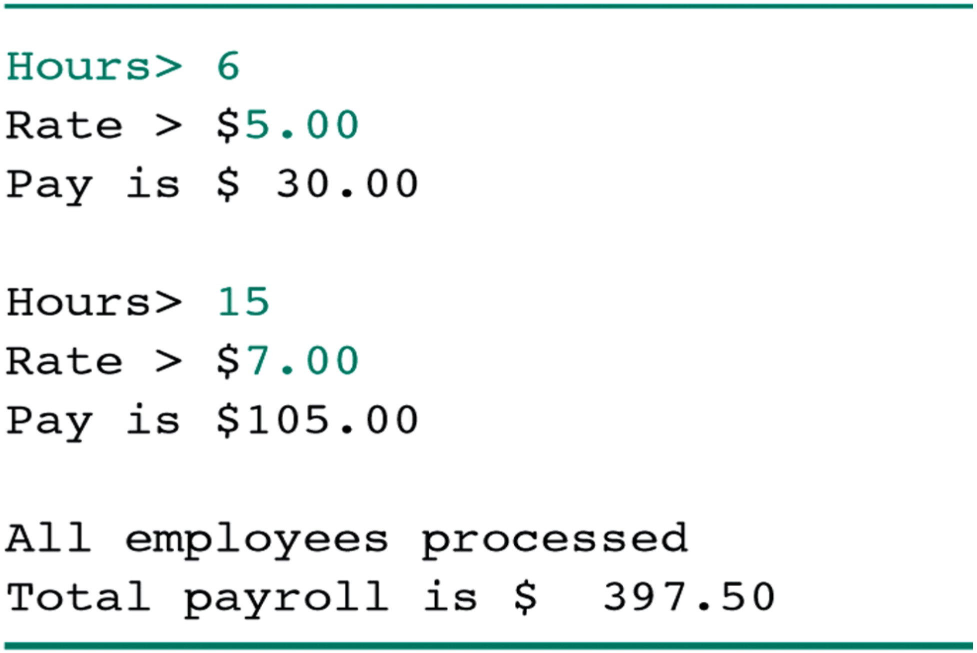


```c
/* Compute the payroll for a company */

#include <stdio.h>

int
main(void)
{
      double total_pay;     /* company payroll           */
      int    count_emp;     /* current employee          */
      int    number_emp;    /* number of employees       */
      double hours;         /* hours worked              */
      double rate;          /* hourly rate               */
      double pay;           /* pay for this period       */

      /* Get number of employees. */
      printf("Enter number of employees> ");
      scanf("%d", &number_emp);

      /* Compute each employee's pay and add it to the payroll. */
      total_pay = 0.0;
      count_emp = 0;
      while (count_emp < number_emp) {
          printf("Hours> ");
          scanf("%lf", &hours);
          printf("Rate > $");
          scanf("%lf", &rate);
          pay = hours * rate;
          printf("Pay is $%6.2f\n\n", pay);
          total_pay = total_pay + pay;              /* Add next pay. */
          count_emp = count_emp + 1;
      }
      printf("All employees processed\n");
      printf("Total payroll is $%8.2f\n", total_pay);

      return (0);
}
```

```
Enter number of employees> 3
Hours> 50
Rate > $5.25
Pay is $262.50
```

Figure 5.4 Program to Compute Company Payroll

Figure 5.4 Program to Compute Company Payroll (cont'd)



```
Hours> 6
Rate > $5.00
Pay is $ 30.00

Hours> 15
Rate > $7.00
Pay is $105.00

All employees processed
Total payroll is $   397.50
```

5.6 Loop Design

- **Sentinel value**
  - An end marker that follows the last item in a list of data
- A loop that processes data until the sentinel value is entered has the form

```
1. Get a line of data
2. while the sentinel value has not been encountered
        3. Process the data line
        4. Get another line of data
```

- For program readability, we usually name the sentinel by defining a **constant macro**.

Example 5.6
(Figure 5.10)

- A program that calculates the sum of a collection of exam scores is a candidate for using a sentinel value.
- Sentinel loop

```
1. Initialize sum to zero
2. Get first score
3. while score is not the sentinel
        4. Add score to sum
        5. Get next score
```

Before the while loop

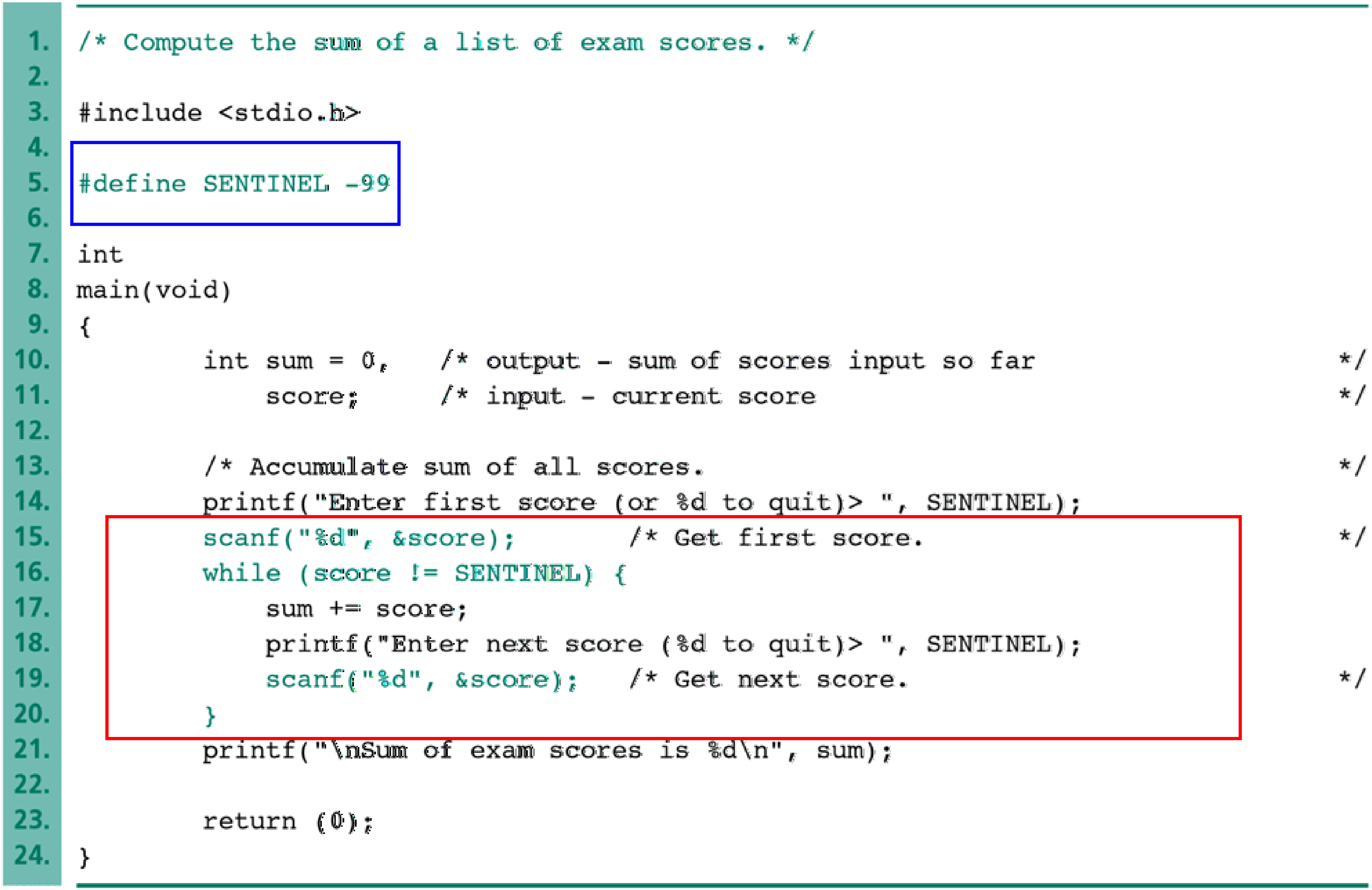
Example 5.6 (cont) (Figure 5.10)

- Incorrect sentinel loop

```
1. Initialize sum to zero
2. while score is not the sentinel
        3. Get score
        4. Add score to sum
```

- Two problems
  - No initializing input statement
  - Sentinel value will be added to sum before exit

Figure 5.10 Sentinel-Controlled while Loop



```c
/* Compute the sum of a list of exam scores. */

#include <stdio.h>

#define SENTINEL -99

int
main(void)
{
        int sum = 0,   /* output - sum of scores input so far                   */
            score;     /* input - current score                                 */

        /* Accumulate sum of all scores.                                        */
        printf("Enter first score (or %d to quit)> ", SENTINEL);
        scanf("%d", &score);       /* Get first score.                          */
        while (score != SENTINEL) {
            sum += score;
            printf("Enter next score (%d to quit)> ", SENTINEL);
            scanf("%d", &score);   /* Get next score.                           */
        }
        printf("\nSum of exam scores is %d\n", sum);

        return (0);
}
```

Sentinel-Controlled Structure

- One input to get the loop going (initialization)
- Second to keep it going (updating)

Using a for Statement to Implement a Sentinel Loop

- The *for* statement form of the *while* loop in Fig.5.10

```
 /* Accumulate sum of all scores */

printf(“Enter first score (or %d to quit)>”, SENTINEL);
for (scanf(“%d”, &score); score != SENTINEL;
      scanf(“%d”, &score)) {
   sum += score;
   printf(“Enter next score(%d to quit)>”, SENTINEL);
}
```

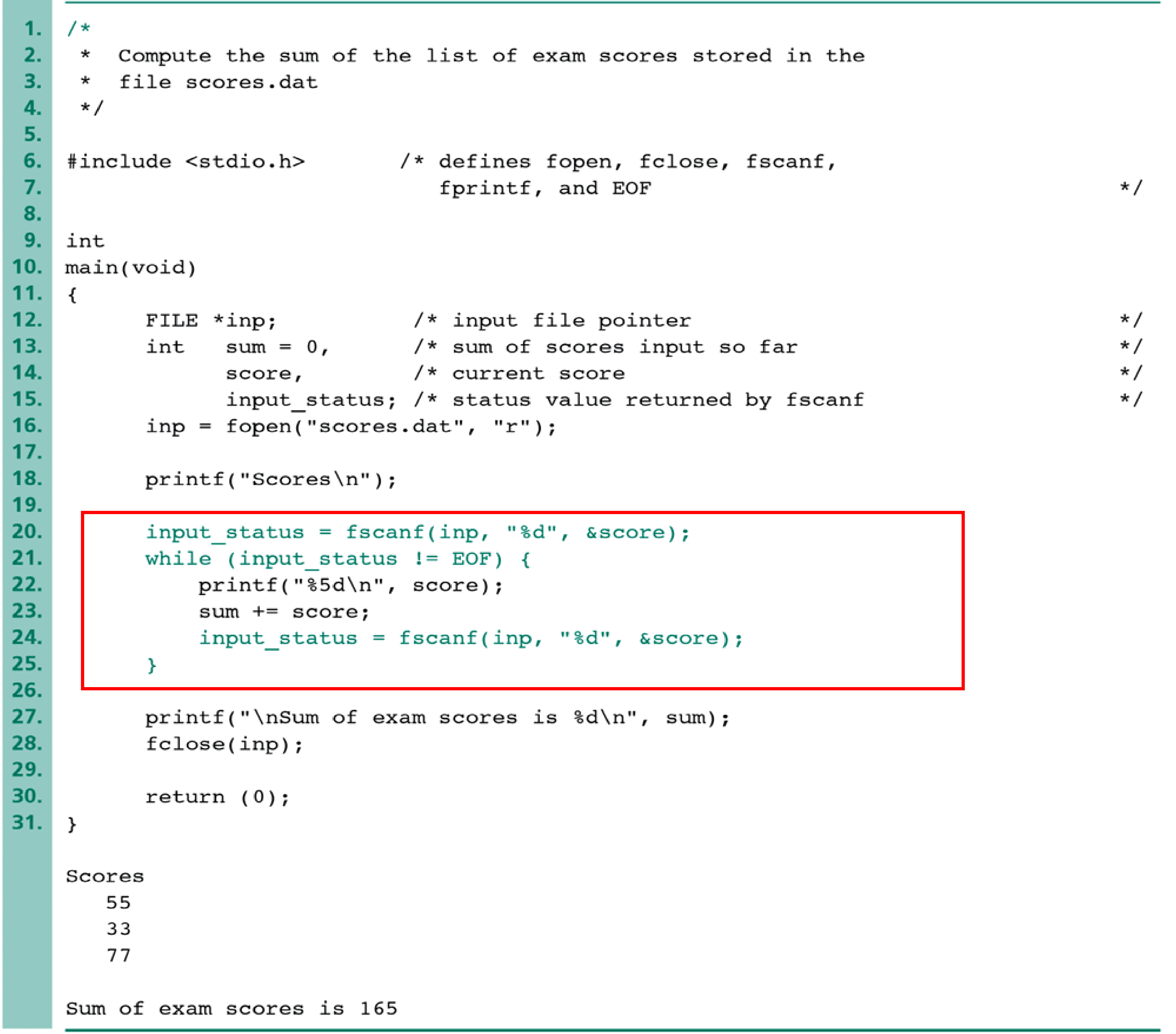
Endfile-Controlled Loops

- Pseudocode for an endfile-controlled loop

```
1. Get the first data values and save input status
2. While input status does not indicate that end of file
   has been reached
      3. Process data value
      4. Get next data value and save input status
```

Program-Controlled Input and Output Files

- declare a file pointer variable
  - File *inp ,   /* pointer to input file   */
              outp ;   / pointer to output file */
- the calls to function fopen
  - inp = fopen("b:distance.dat", "r") ;
  - outp = fopen("b:distance.out", "w") ;
- use of the functions
  - fscanf(inp, "%lf", &miles);
  - fprintf(outp, "The distance in miles is %.2f. \n", miles);
- end of use
  - fclose(inp);
  - fclose(outp);



```c
/*
 *  Compute the sum of the list of exam scores stored in the
 *  file scores.dat
 */

#include <stdio.h>       /* defines fopen, fclose, fscanf,
                            fprintf, and EOF                                   */

int
main(void)
{
      FILE *inp;          /* input file pointer                                */
      int   sum = 0,      /* sum of scores input so far                        */
            score,        /* current score                                     */
            input_status; /* status value returned by fscanf                   */
      inp = fopen("scores.dat", "r");

      printf("Scores\n");

      input_status = fscanf(inp, "%d", &score);
      while (input_status != EOF) {
          printf("%5d\n", score);
          sum += score;
          input_status = fscanf(inp, "%d", &score);
      }

      printf("\nSum of exam scores is %d\n", sum);
      fclose(inp);

      return (0);
}
```

```
Scores
   55
   33
   77

Sum of exam scores is 165
```

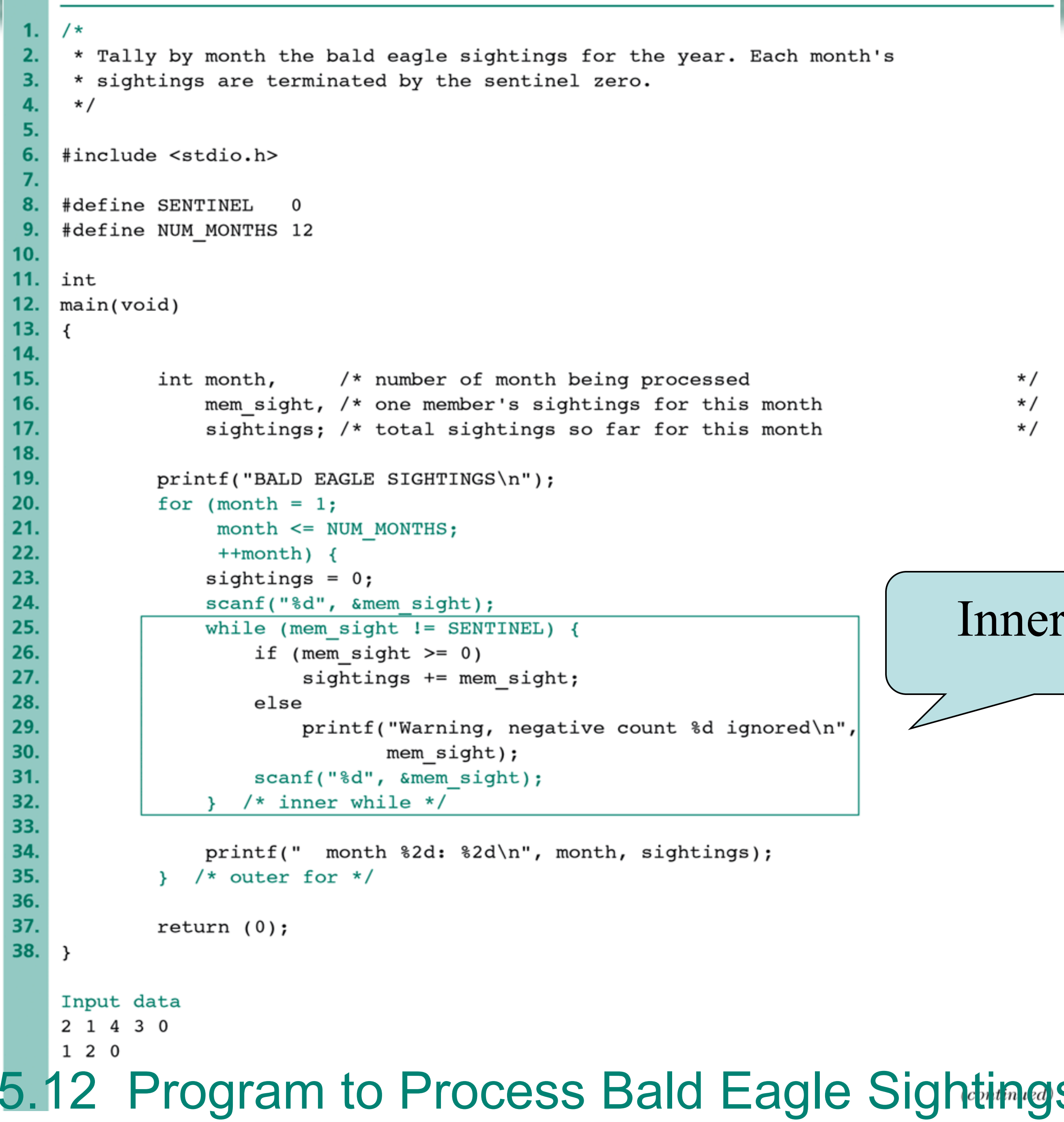
Figure 5.11 Batch Version of Sum of Exam Scores Program

Practice

- Find the maximum and minimum value in a file.
  - Note: you don’t know how many numbers are in the file and you should use the status returned by fscanf to know whether it is the end.

5.7 Nested Loops

- Nested loops consist of an outer loop with one or more inner loops.
- Example 5.7 (Figure 5.12)
  - The program contains a sentinel loop nested within a counting loop.



```c
/*
 * Tally by month the bald eagle sightings for the year. Each month's
 * sightings are terminated by the sentinel zero.
 */

#include <stdio.h>

#define SENTINEL   0
#define NUM_MONTHS 12

int
main(void)
{

        int month,     /* number of month being processed                    */
            mem_sight, /* one member's sightings for this month              */
            sightings; /* total sightings so far for this month              */

        printf("BALD EAGLE SIGHTINGS\n");
        for (month = 1;
             month <= NUM_MONTHS;
             ++month) {
            sightings = 0;
            scanf("%d", &mem_sight);
            while (mem_sight != SENTINEL) {
                if (mem_sight >= 0)
                    sightings += mem_sight;
                else
                    printf("Warning, negative count %d ignored\n",
                           mem_sight);
                scanf("%d", &mem_sight);
            }  /* inner while */

            printf("  month %2d: %2d\n", month, sightings);
        }  /* outer for */

        return (0);
}
```

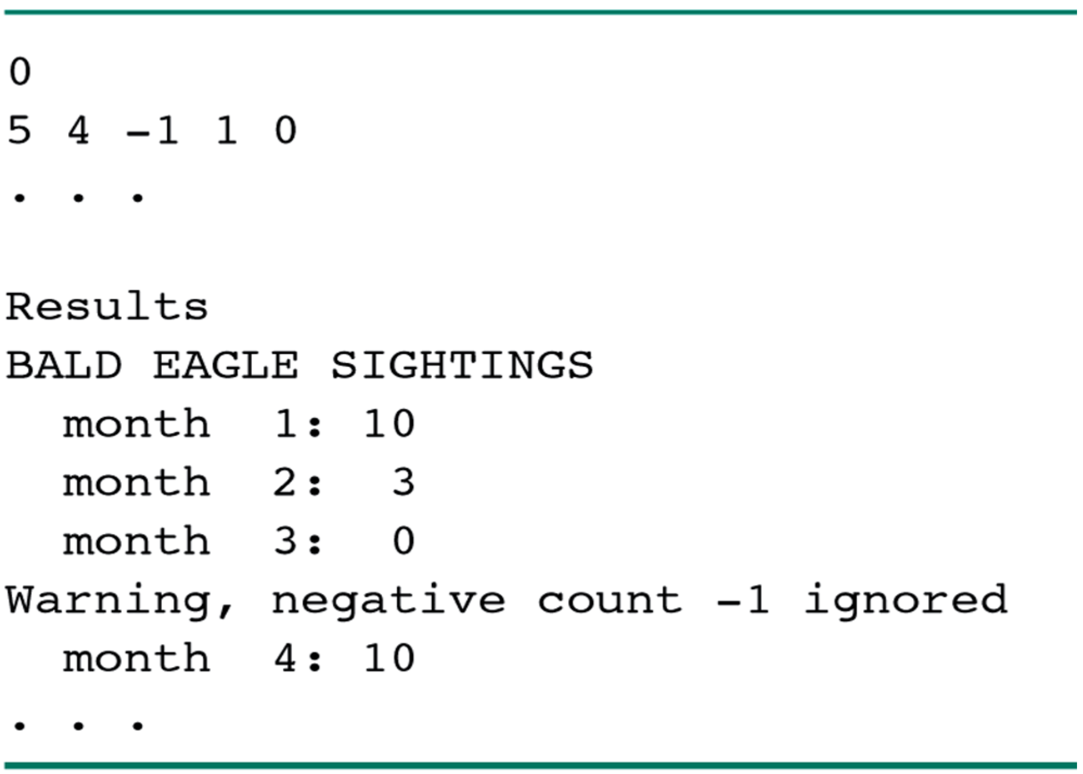
Input data
```
2 1 4 3 0
1 2 0
```

(continued)

Inner loop

Figure 5.12 Program to Process Bald Eagle Sightings for a Year

Figure 5.12 Program to Process Bald Eagle Sightings for a Year (cont’d)

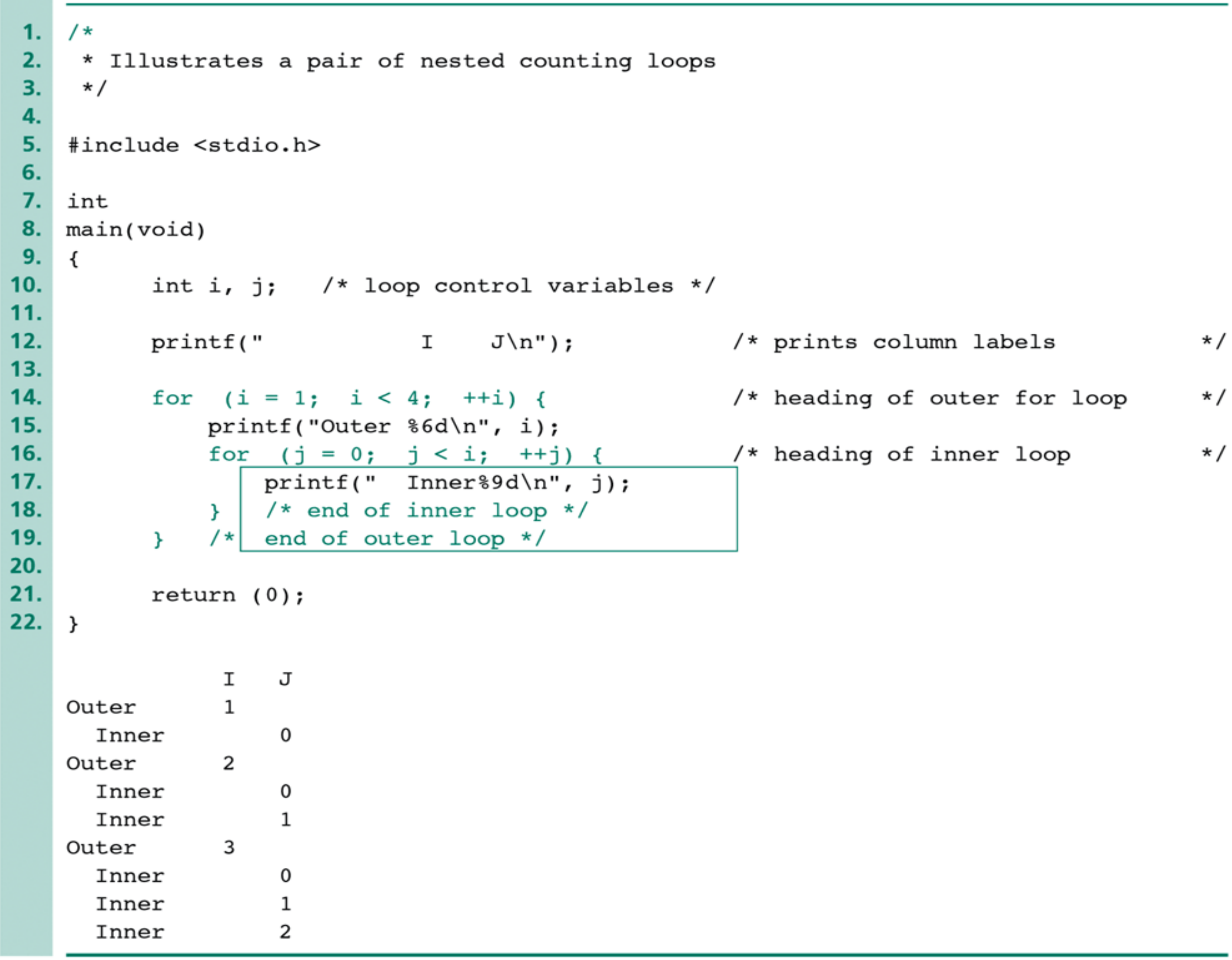


```
0
5 4 -1 1 0
. . .

Results
BALD EAGLE SIGHTINGS
  month  1: 10
  month  2:  3
  month  3:  0
Warning, negative count -1 ignored
  month  4: 10
. . .
```

Example 5.8
(Figure 5.13)

- A sample run of a program with two nested counting loops.
- Not be able to use the same variable as the loop control variable of both an outer and an inner for loop in the same nest.

Figure 5.13 Nested Counting Loop Program



```
/*
 * Illustrates a pair of nested counting loops
 */

#include <stdio.h>

int
main(void)
{
      int i, j;   /* loop control variables */

      printf("           I   J\n");          /* prints column labels          */

      for  (i = 1;  i < 4;  ++i) {             /* heading of outer for loop     */
          printf("Outer %6d\n", i);
          for  (j = 0;  j < i;  ++j) {         /* heading of inner loop         */
              printf("  Inner%9d\n", j);
          }   /* end of inner loop */
      }   /* end of outer loop */

      return (0);
}
```

```
           I   J
Outer      1
  Inner        0
Outer      2
  Inner        0
  Inner        1
Outer      3
  Inner        0
  Inner        1
  Inner        2
```

5.8 The *do-while* Statement and Flag-Controlled Loops

- do-while statement execute at least one time.
- Pseudocode

```
1. Get data value
2. If data value isn’t in the acceptable range, go back
   to step 1.
```

Syntax of *do-while* Statement

- Syntax：

```
do
        statement
while (loop repetition condition);
```

- Example：

```
/* Find first even number input */
do
        status = scanf(“%d”, &num)
while (status>0 && (num%2) != 0)
```

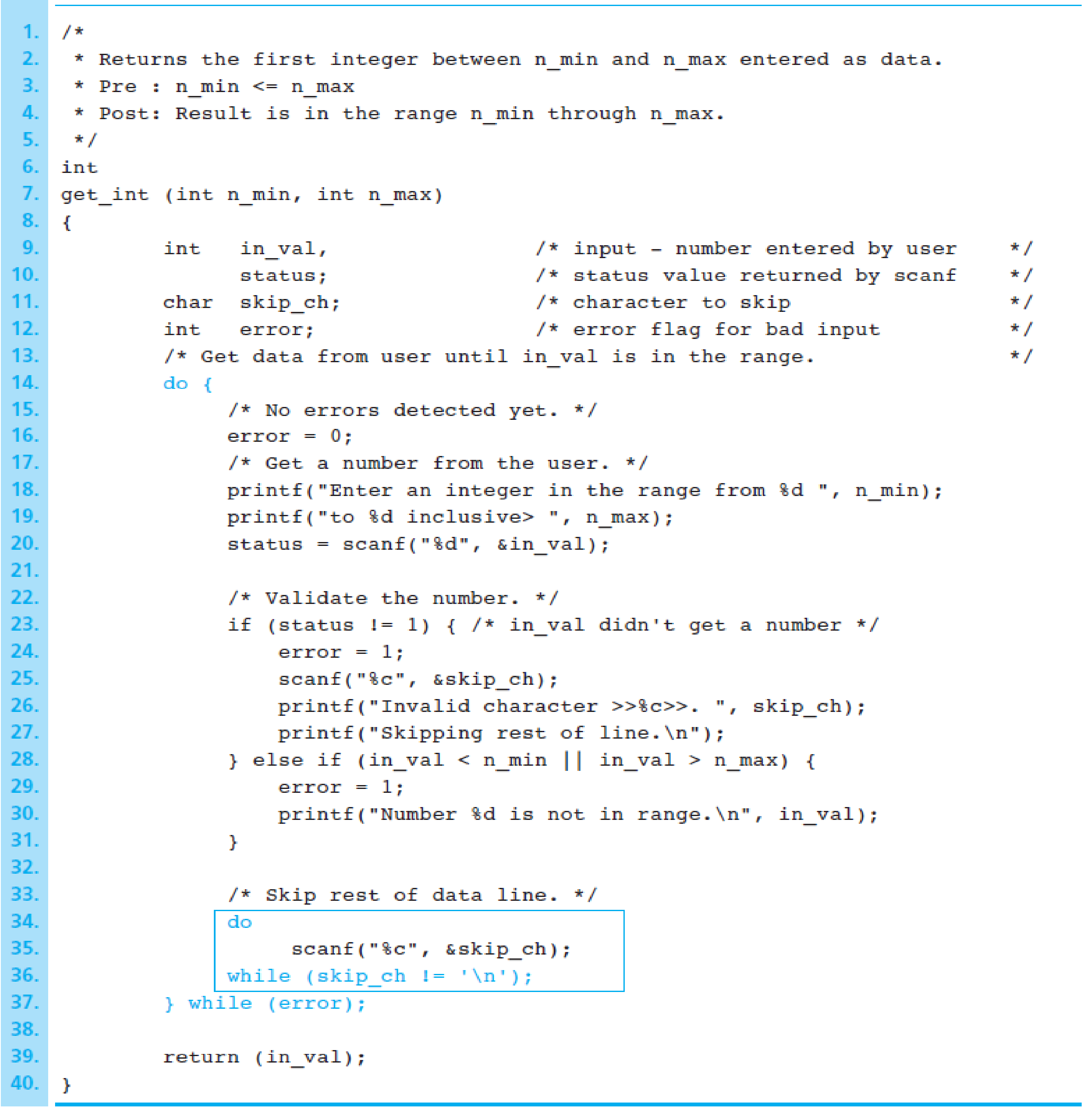
while

Flag-Controlled Loops for Input Validation

- flag
  - A type **int** variable used to represent whether or not a certain event has occurred
  - A flag has one of two values
    - 1 (true)
    - 0 (false)

Example 5.10
(Figure 5.14)

- Function get_int returns an integer value that is in the range specified by its two arguments.
- The outer do-while structure implements the stated purpose of the function.
- The type int variable error acts as a program flag.
- error is initialized to 0 and is changed to 1 when an error is detected.



```c
/*
 * Returns the first integer between n_min and n_max entered as data.
 * Pre : n_min <= n_max
 * Post: Result is in the range n_min through n_max.
 */
int
get_int (int n_min, int n_max)
{
        int   in_val,                /* input - number entered by user    */
              status;                /* status value returned by scanf    */
        char  skip_ch;               /* character to skip                 */
        int   error;                 /* error flag for bad input          */
        /* Get data from user until in_val is in the range.               */
        do {
             /* No errors detected yet. */
             error = 0;
             /* Get a number from the user. */
             printf("Enter an integer in the range from %d ", n_min);
             printf("to %d inclusive> ", n_max);
             status = scanf("%d", &in_val);

             /* Validate the number. */
             if (status != 1) { /* in_val didn't get a number */
                 error = 1;
                 scanf("%c", &skip_ch);
                 printf("Invalid character >>%c>>. ", skip_ch);
                 printf("Skipping rest of line.\n");
             } else if (in_val < n_min || in_val > n_max) {
                 error = 1;
                 printf("Number %d is not in range.\n", in_val);
             }

             /* Skip rest of data line. */
             do
                  scanf("%c", &skip_ch);
             while (skip_ch != '\n');
        } while (error);

        return (in_val);
}
```

Figure 5.14 Validating Input Using do-while Statement

5.9 Iterative Approximation

- How to find roots of equations?
- The ***bisection*** method is one way of approximating a root of the equation *f*(*x*)=0.

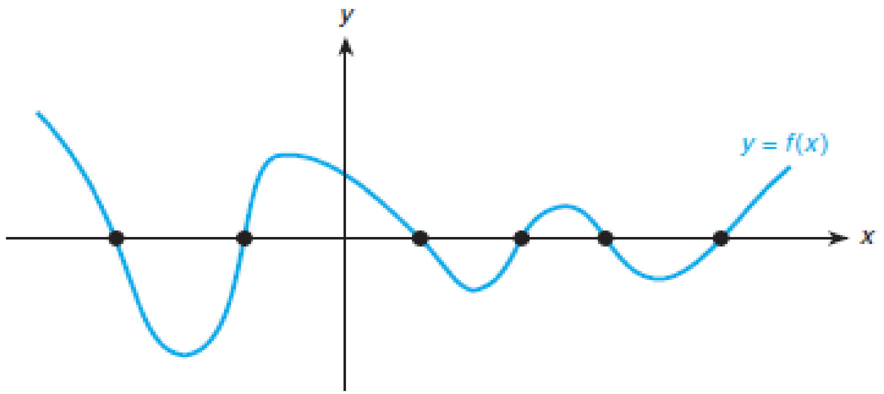


Figure 5.15 Six Root for the Equation *f*(x)=0

Function Parameters

- Include a function in a parameter list of another function

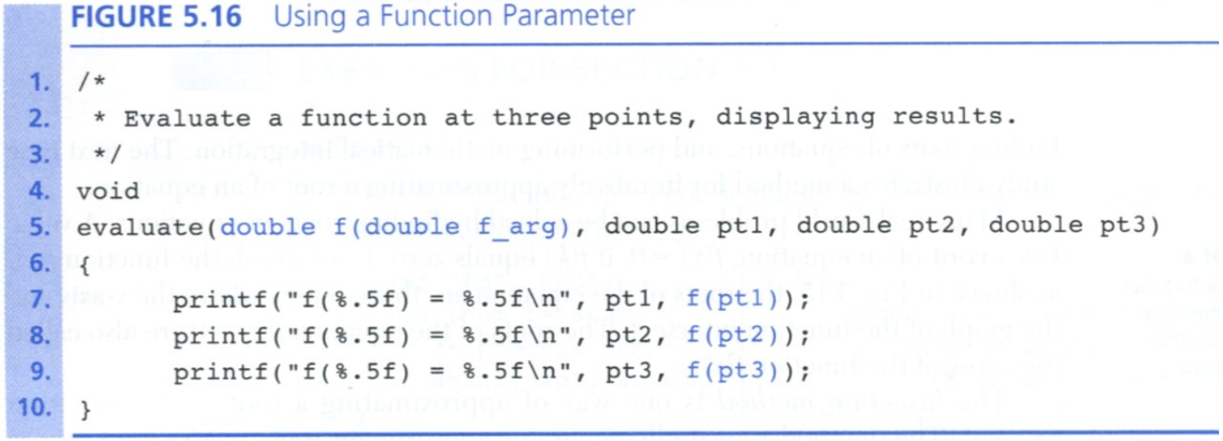


FIGURE 5.16 Using a Function Parameter

```
/*
 * Evaluate a function at three points, displaying results.
 */
void
evaluate(double f(double f_arg), double pt1, double pt2, double pt3)
{
        printf("f(%.5f) = %.5f\n", pt1, f(pt1));
        printf("f(%.5f) = %.5f\n", pt2, f(pt2));
        printf("f(%.5f) = %.5f\n", pt3, f(pt3));
}
```

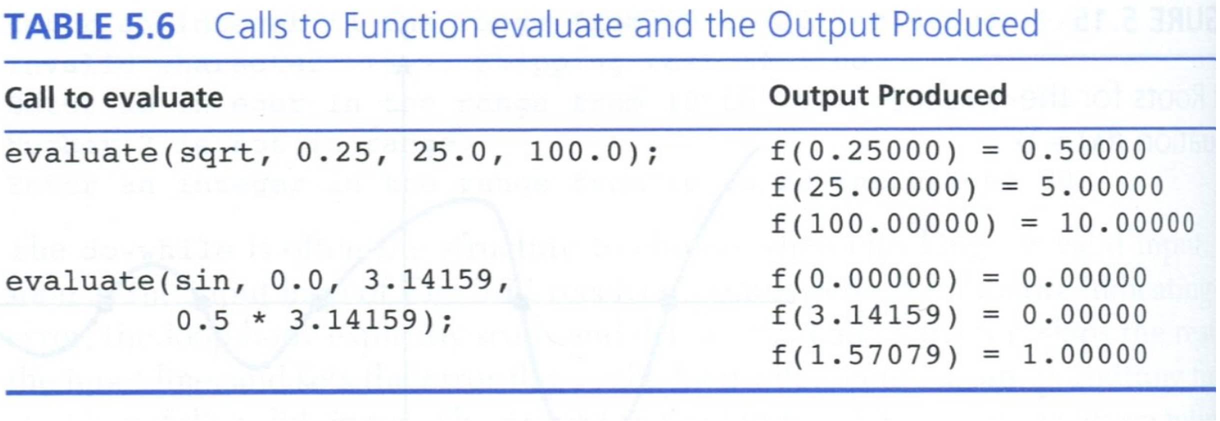


TABLE 5.6 Calls to Function evaluate and the Output Produced

Call to evaluate	Output Produced

`evaluate(sqrt, 0.25, 25.0, 100.0);`	`f(0.25000) = 0.50000`
	`f(25.00000) = 5.00000`
	`f(100.00000) = 10.00000`

`evaluate(sin, 0.0, 3.14159,`
`         0.5 * 3.14159);`
	`f(0.00000) = 0.00000`
	`f(3.14159) = 0.00000`
	`f(1.57079) = 1.00000`

Case study: Bisection method for finding roots

<Step 1> Problem

Develop a function **bisect** that approximates a root of a function f on an interval that contains an odd number of roots.

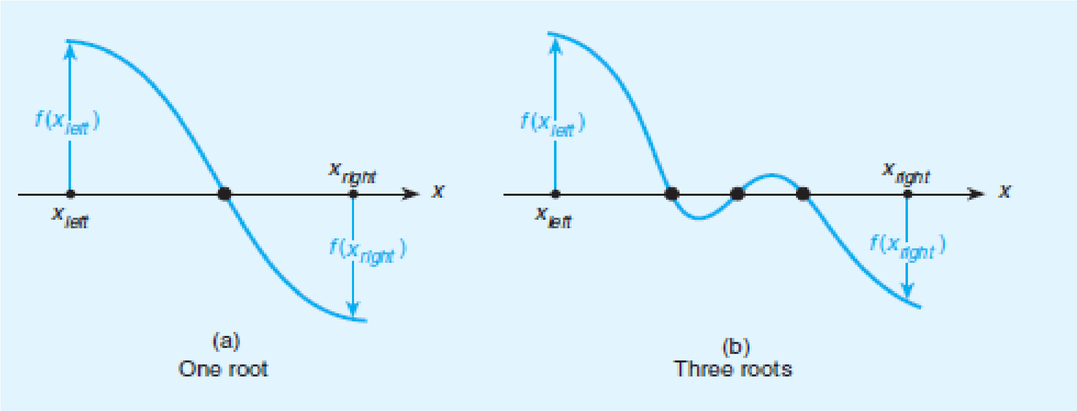
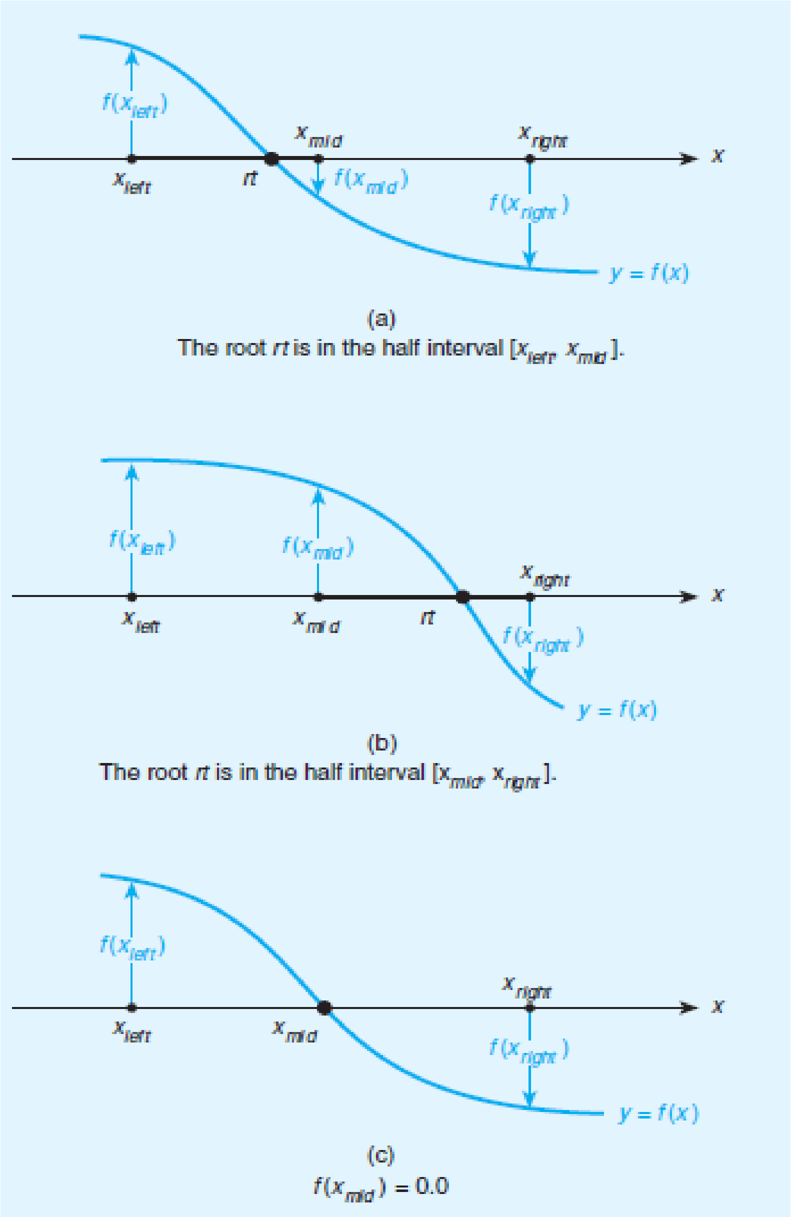


Figure 5.17 Change of Sign Implies an Odd Number of Roots

Figure 5.18

Three Possibilities That Arise When the Interval $[X_{left}, X_{right}]$ Is Bisected



(a)
The root rt is in the half interval $[x_{left}, x_{mid}]$.

(b)
The root rt is in the half interval $[x_{mid}, x_{right}]$.

(c)
$f(x_{mid}) = 0.0$

Case study: Bisection method for finding roots (cont)

<Step 2> Analysis (Additions to data requirements)

- Problem Inputs
  - double x_left /* left endpoint of interval */
  - double x_right /* right endpoint of interval */
  - double epison /* error tolerance */
  - double f (double farg) /* function whose root is sought */
- Problem output
  - double root /* approximate root of f */

Case study: Bisection method for finding roots (cont)

- Design (Initial Algorithm)

```
1. if the interval contains an even number of roots
        2. Display “not root” message.
        3. Return NO_ROOT and exit the function.
4. Repeat as long as interval is greater than tolerance and a root is not found
        5. Compute the function value at the midpoint of the interval.
        6. if the function value at midpoint is zero
            7. Set root to the midpoint.
            else
            8. Choose the left or right half of the interval and continue the search.
9. Return the midpoint of the final interval as the root.
```

Case study: Bisection method for finding roots (cont)

- Program variables
  - int root_found /* whether root is found */
  - double x_mid /* interval midpoint */
  - double f_left, /* values of function at left and */
f_mid, /* right endpoints and at midpoint */
f_right /* of the interval */

Case study: Bisection method for finding roots (cont)

- Refinement

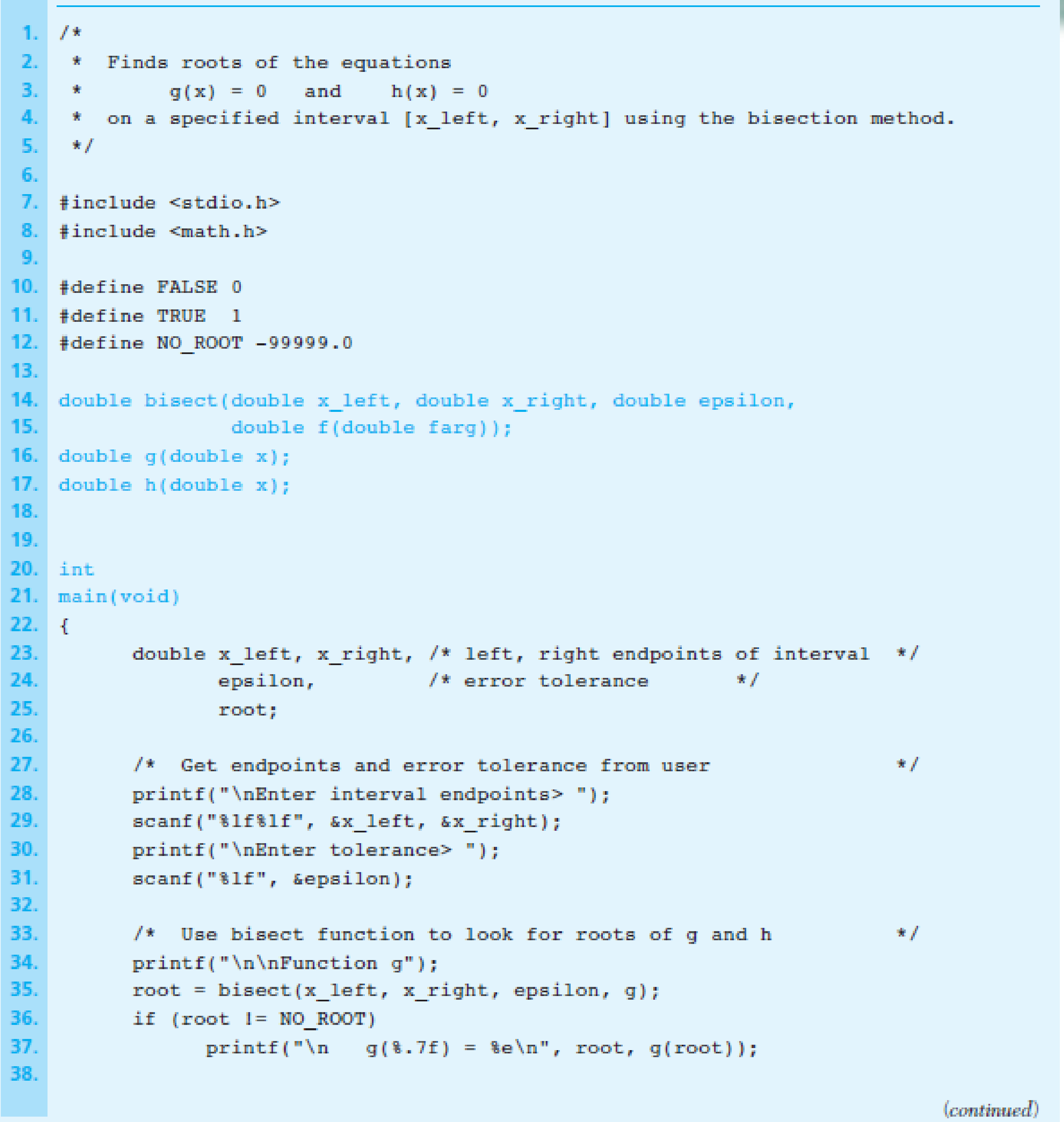
1. if the interval contains an even number of roots
   - 1.1 f_left = f(x_left);
   - 1.2 f_right = f(x_right);
   - 1.3 if signs of f_left and f_right are the same (i.e., if their product is nonnegative)

4. Repeat as long as interval is greater than tolerance and a root is not found
   - 4.1 while x_right – x_left > epsilon and !root_found

8. Choose the left or right half of the interval and continue the search.
   - 8.1 if root is in left half of interval (f_left*f_mid <0.0)
   - 8.2 Change right end to midpoint

     else
   - 8.3 Change left end to midpoint

Figure 5.19 **Finding a Function Root using Bisection Method**

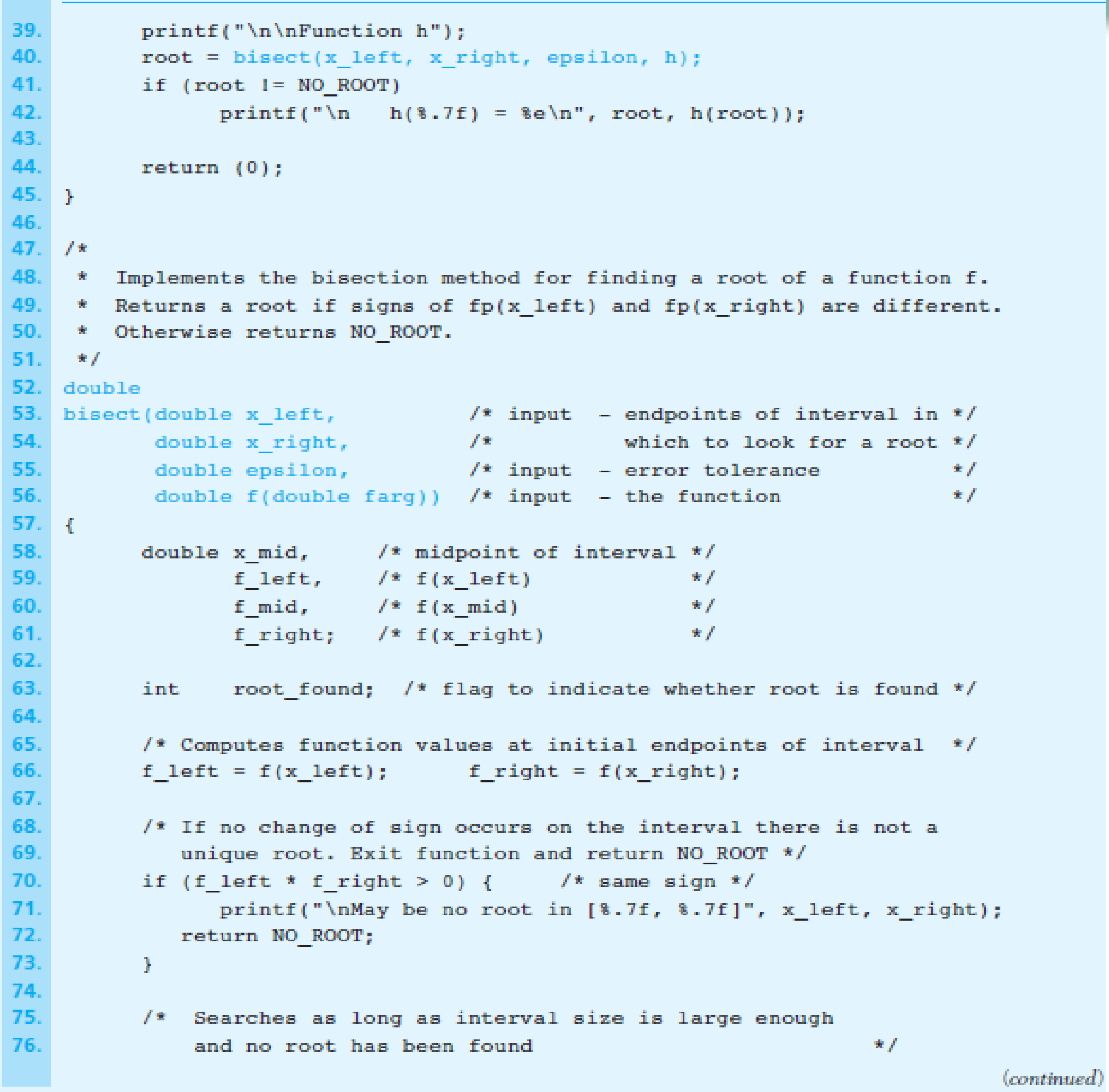


```c
/*
 *  Finds roots of the equations
 *        g(x) = 0    and    h(x) = 0
 *  on a specified interval [x_left, x_right] using the bisection method.
 */

#include <stdio.h>
#include <math.h>

#define FALSE 0
#define TRUE  1
#define NO_ROOT -99999.0

double bisect(double x_left, double x_right, double epsilon,
              double f(double farg));
double g(double x);
double h(double x);


int
main(void)
{
      double x_left, x_right, /* left, right endpoints of interval  */
             epsilon,         /* error tolerance       */
             root;

      /*  Get endpoints and error tolerance from user                */
      printf("\nEnter interval endpoints> ");
      scanf("%lf%lf", &x_left, &x_right);
      printf("\nEnter tolerance> ");
      scanf("%lf", &epsilon);

      /*  Use bisect function to look for roots of g and h           */
      printf("\n\nFunction g");
      root = bisect(x_left, x_right, epsilon, g);
      if (root != NO_ROOT)
            printf("\n   g(%.7f) = %e\n", root, g(root));

```

(continued)

Figure 5.19 **Finding a Function Root using Bisection Method**

(cont’d)



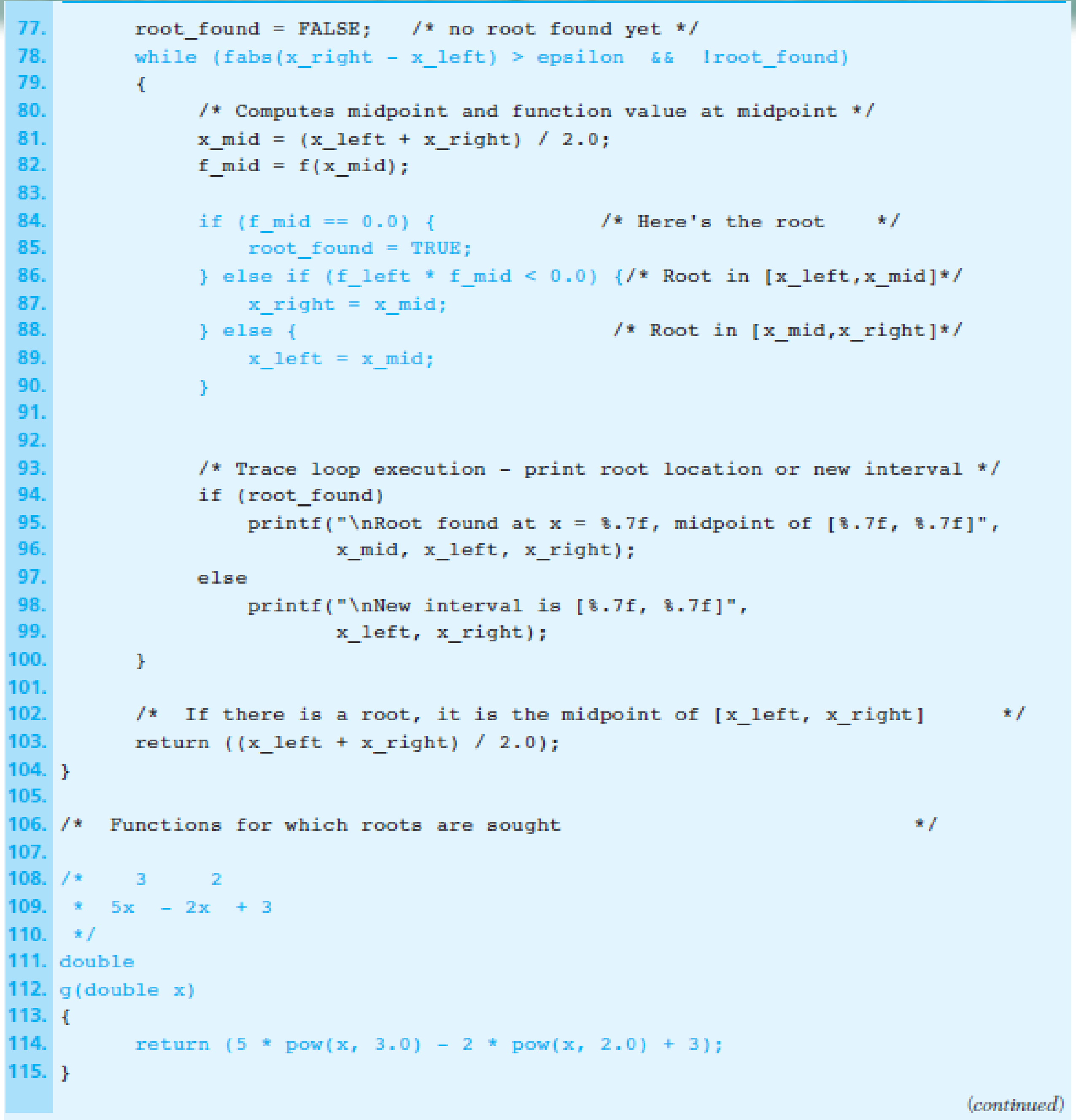
```
      printf("\n\nFunction h");
      root = bisect(x_left, x_right, epsilon, h);
      if (root != NO_ROOT)
            printf("\n   h(%.7f) = %e\n", root, h(root));

      return (0);
}

/*
 *  Implements the bisection method for finding a root of a function f.
 *  Returns a root if signs of fp(x_left) and fp(x_right) are different.
 *  Otherwise returns NO_ROOT.
 */
double
bisect(double x_left,          /* input   - endpoints of interval in */
       double x_right,         /*           which to look for a root */
       double epsilon,         /* input   - error tolerance          */
       double f(double farg))  /* input   - the function             */
{
      double x_mid,     /* midpoint of interval */
             f_left,    /* f(x_left)            */
             f_mid,     /* f(x_mid)             */
             f_right;   /* f(x_right)           */

      int    root_found;  /* flag to indicate whether root is found */

      /* Computes function values at initial endpoints of interval   */
      f_left = f(x_left);      f_right = f(x_right);

      /* If no change of sign occurs on the interval there is not a
         unique root. Exit function and return NO_ROOT */
      if (f_left * f_righ > 0) {      /* same sign */
            printf("\nMay be no root in [%.7f, %.7f]", x_left, x_right);
         return NO_ROOT;
      }

      /*  Searches as long as interval size is large enough
          and no root has been found                          */
```

(continued)

Figure 5.19

Finding a Function Root using Bisection Method

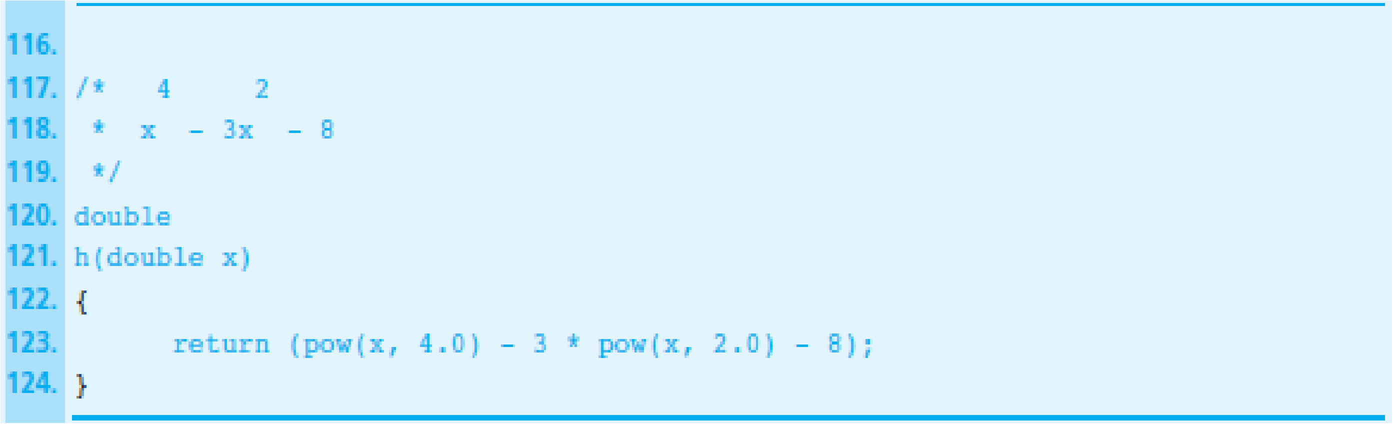
(cont’d)



```
      root_found = FALSE;   /* no root found yet */
      while (fabs(x_right - x_left) > epsilon  &&  !root_found)
      {
           /* Computes midpoint and function value at midpoint */
           x_mid = (x_left + x_right) / 2.0;
           f_mid = f(x_mid);

           if (f_mid == 0.0) {             /* Here's the root    */
               root_found = TRUE;
           } else if (f_left * f_mid < 0.0) {/* Root in [x_left,x_mid]*/
               x_right = x_mid;
           } else {                         /* Root in [x_mid,x_right]*/
               x_left = x_mid;
           }


           /* Trace loop execution - print root location or new interval */
           if (root_found)
               printf("\nRoot found at x = %.7f, midpoint of [%.7f, %.7f]",
                      x_mid, x_left, x_right);
           else
               printf("\nNew interval is [%.7f, %.7f]",
                      x_left, x_right);
      }

      /*  If there is a root, it is the midpoint of [x_left, x_right]      */
      return ((x_left + x_right) / 2.0);
}

/*  Functions for which roots are sought                            */

/*     3     2
 *  5x  - 2x  + 3
 */
double
g(double x)
{
      return (5 * pow(x, 3.0) - 2 * pow(x, 2.0) + 3);
}
```

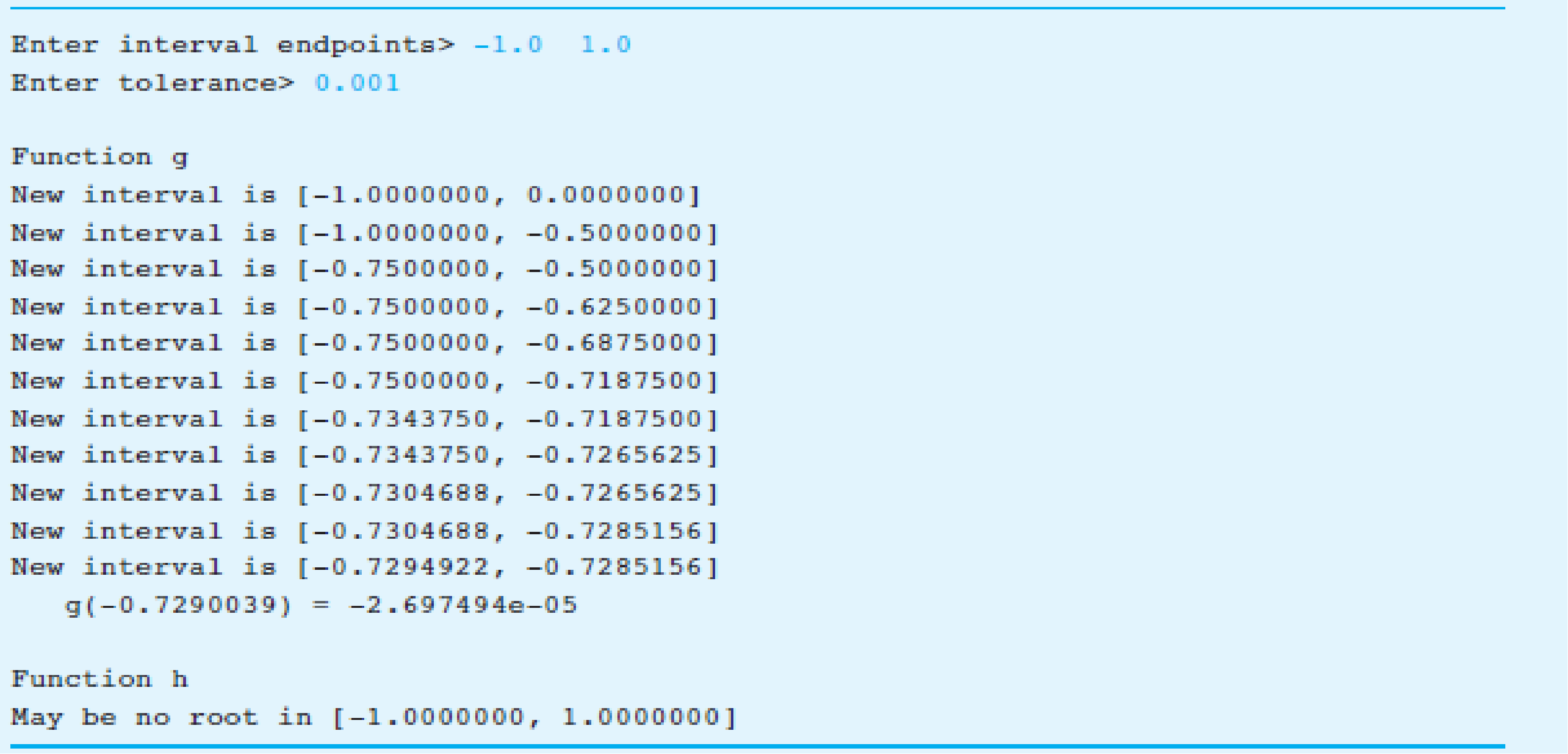
(*continued*)

Figure 5.19 **Finding a Function Root using Bisection Method** (cont’d)



```

/*    4      2
 *  x  - 3x  - 8
 */
double
h(double x)
{
      return (pow(x, 4.0) - 3 * pow(x, 2.0) - 8);
}
```

Figure 5.20 **Sample Run of Bisection Program with Trace Code Included**



```
Enter interval endpoints> -1.0  1.0
Enter tolerance> 0.001

Function g
New interval is [-1.0000000, 0.0000000]
New interval is [-1.0000000, -0.5000000]
New interval is [-0.7500000, -0.5000000]
New interval is [-0.7500000, -0.6250000]
New interval is [-0.7500000, -0.6875000]
New interval is [-0.7500000, -0.7187500]
New interval is [-0.7343750, -0.7187500]
New interval is [-0.7343750, -0.7265625]
New interval is [-0.7304688, -0.7265625]
New interval is [-0.7304688, -0.7285156]
New interval is [-0.7294922, -0.7285156]
   g(-0.7290039) = -2.697494e-05

Function h
May be no root in [-1.0000000, 1.0000000]
```

5.10 How to Debug and Test Programs

- Using debugger programs
  - Execute programs one statement at a time (single-step execution)
  - Set breakpoints at selected statements when a program is very long
- Debugging without a debugger
  - Insert *extra diagnostic calls* to printf that display intermediate results at critical points
  - #define DEBUG 1
  - #define DEBUG 0

Example: debug using printf

```
 while (score != SENTINEL) {
        sum += score;
        if   (DEBUG)
             printf (“***** score is %d, sum is %d\n, score, sum);
        printf (“Enter next score (%d to quit)> “, SENTINEL);
        scanf(“%d”, &score);
}
```

```
#define DEBUG  1     turn on diagnostics
#define DEBUG  0     turn off
```

Off-by-One Loop Errors

- A common logic error
  - – A loop executes one more time or one less time

```
for (count = 0; count <= n; ++count)
    sum += count;
```

execute n+1 time

- Loop boundaries
  - – Initial and final values of the loop control variable

5.11 Common Programming Errors(1/3)

- Do not confuse *if* and *while* statements
  - *if* statement implement a decision step
  - *while/for* statement implement a loop
- Remember to end the initialization expression and the loop repetition condition with semicolons.
- Remember to use braces around a loop body consisting of multiple statements.
- Remember to provide a prompt for the users, when using a sentinel-controlled loop.
- Make sure the sentinel value cannot be confused with a normal data item.

Common Programming Errors (2/3)

- Use *do-while* only when there is no possibility of zero loop iterations.
- Replace the segment with a while or for statement when adding an if statement.
  - if (condition$_1$)

```
    do {
            …
    } while(condition₁);
```

Common Programming Errors(3/3)

- Do not use increment, decrement, or compound assignment operators as subexpressions in complex expressions.
- Be sure that the operand of an increment or decrement operator is a variable and that this variable is referenced after executing the increment or decrement operation.

Chapter Review (1)

- Two kinds of loops occur frequently in programming：
  - Counting loops
    - The number of iterations required can be determined before the loop is entered.
  - Sentinel-controlled loops
    - Repetition continues until a special data value is scanned.

Chapter Review (2)

- Pseudocode for each form
  - Counter-controlled loop

```
Set loop control variable to an initial value of 0.
While loop control variable<final value
        …..
        Increase loop control variable by 1
```

  - Sentinel-controlled loop

```
Get a line of data
While the sentinel value has not been encountered
        Process the data line
        Get another line of data
```

Chapter Review (3)

- Pseudocode for each form
  - Endfile-controlled loop

```
Get first data value and save input status
While input status does not indicate that end of the file has been reached
        Process data value
    Get next data value and save input status
```

  - Input validation loop

```
Get a data value
if data value isn’t in the acceptable range,
                go back to first step
```

Chapter Review (4)

- Pseudocode for each form
  - General condition loop

```
Initialize loop control variable
As long as exit condition hasn’t been met,
                    continue processing
```

- C provides three statements for implementing loops: while, for, do-while
- The loop control variable must be initialized, tested, and updated.