Repetition and Loop Statements
Chapter 5

Problem Solving & Program Design in C

Eighth Edition

Jeri R. Hanly & Elliot B. Koffman

Chapter Objectives

- To understand why repetition is an important control structure in programming
- To learn about loop control variables and the three steps needed to control loop repetition
- To learn how to use the C for, while, and do-while statements for writing loops and when to use each statement type
- To learn how to accumulate a sum or a product within a loop body

Chapter Objectives

- To learn common loop patterns such as counting loops, sentinel-controlled loops, and flag-controlled loops
- To understand nested loops and how the outer loop control variable and inner loop control variable are changed in a nested loop
- To learn how to debug programs using a debugger
- To learn how to debug programs by adding diagnostic output statements

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

Comparison of Loop Kinds

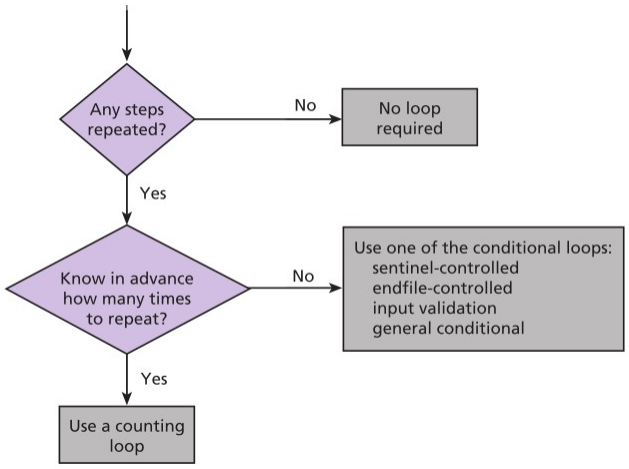
- counting loop
  - we can determine before loop execution exactly how many loop repetitions will be needed to solve the problem
    - while, for
- sentinel-controlled loop
  - input of a list of data of any length ended by a special value
    - while, for

Comparison of Loop Kinds

- endfile-controlled loop
  - input of a single list of data of any length from a data file
    - while, for
- input validation loop
  - repeated interactive input of a data value until a value within the valid range is entered
    - do-while
- general conditional loop
  - repeated processing of data until a desired condition is met
    - while, for

FIGURE 5.1

Flow Diagram of Loop Choice Process



Counting Loops

- counter-controlled loop
  - a.k.a. counting loop
  - a loop whose required number of iterations can be determined before loop execution begins
- loop repetition condition
  - the condition that controls loop repetition

Counting Loops

- loop control variable
  - the variable whose value controls loop repetition
- infinite loop
  - a loop that executes forever

while Statement Syntax

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

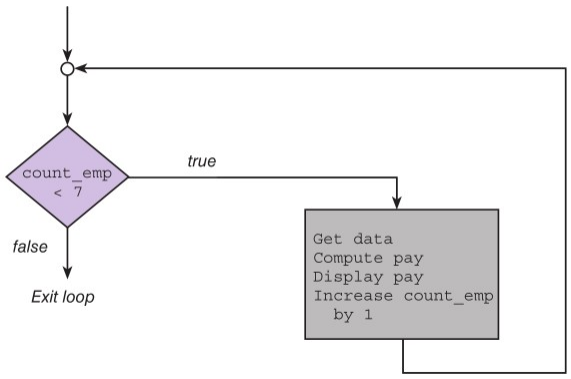
```
/* display N asterisks. */
count_star = 0
while (count_star  <  N) {
        printf("*");
        count_star = count_star + 1;
}
```

FIGURE 5.2 Program Fragment with a Loop

```
1.  count_emp = 0;              /* no employees processed yet      */
2.  while (count_emp < 7) {     /* test value of count_emp         */
3.      printf("Hours> ");
4.      scanf("%d", &hours);
5.      printf("Rate> ");
6.      scanf("%lf", &rate);
7.      pay = hours * rate;
8.      printf("Pay is $%6.2f\n", pay);
9.      count_emp = count_emp + 1; /* increment count_emp          */
10. }
11. printf("\nAll employees processed\n");
```


FIGURE 5.3

Flowchart for a while Loop



Computing a Sum or Product in a Loop

- accumulator
  - a variable used to store a value being computed in increments during the execution of a loop

FIGURE 5.4 Program to Compute Company Payroll

```
1.  /* Compute the payroll for a company */
2.
3.  #include <stdio.h>
4.
5.  int
6.  main(void)
7.  {
8.      double total_pay;     /* company payroll        */
9.      int    count_emp;     /* current employee       */
10.     int    number_emp;    /* number of employees    */
11.     double hours;         /* hours worked           */
12.     double rate;          /* hourly rate            */
13.     double pay;           /* pay for this period    */
```

(continued)

FIGURE 5.4 (continued)

```
14.
15.     /* Get number of employees. */
16.     printf("Enter number of employees> ");
17.     scanf("%d", &number_emp);
18.
19.     /* Compute each employee's pay and add it to the payroll. */
20.     total_pay = 0.0;
21.     count_emp = 0;
22.     while (count_emp < number_emp) {
23.         printf("Hours> ");
24.         scanf("%lf", &hours);
25.         printf("Rate > $");
26.         scanf("%lf", &rate);
27.         pay = hours * rate;
28.         printf("Pay is $%6.2f\n\n", pay);
29.         total_pay = total_pay + pay;              /* Add next pay. */
30.         count_emp = count_emp + 1;
31.     }
32.     printf("All employees processed\n");
33.     printf("Total payroll is $%8.2f\n", total_pay);
34.
35.     return (0);
36. }

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

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
```


TABLE 5.2 Trace of Three Repetitions of Loop in Fig. 5.4

Statement	hours	rate	pay	total_pay	count_emp	Effect
	?	?	?	0.0	0	
`count_emp < number_emp`						true
`scanf("%lf", &hours);`	50.0					get hours
`scanf("%lf", &rate);`		5.25				get rate
`pay = hours * rate;`			262.5			find pay
`total_pay = total_pay + pay;`				262.5		add to total_pay
`count_emp = count_emp + 1;`					1	increment count_emp
`count_emp < number_emp`						true
`scanf("%lf", &hours);`	6.0					get hours
`scanf("%lf", &rate);`		5.0				get rate
`pay = hours * rate;`			30.0			find pay
`total_pay = total_pay + pay;`				292.5		add to total_pay
`count_emp = count_emp + 1;`					2	increment count_emp
`count_emp < number_emp`						true
`scanf("%lf", &hours);`	15.0					get hours
`scanf("%lf", &rate);`		7.0				get rate
`pay = hours * rate;`			105.0			find pay
`total_pay = total_pay + pay;`				397.5		add pay to total_pay
`count_emp = count_emp + 1;`					3	increment count_emp

General Conditional Loop

1. Initialize loop control variable.
2. As long as exit condition hasn't been met
   3. Continue processing

TABLE 5.3 Compound Assignment Operators

Statement with Simple Assignment Operator	Equivalent Statement with Compound Assignment Operator
`count_emp = count_emp + 1;`	`count_emp += 1;`
`time = time - 1;`	`time -= 1;`
`total_time = total_time + times;`	`total time += time;`
`product = product * item;`	`product *= item;`
`n = n * (x + 1);`	`n *= x + 1;`

Loop Control Components

- initialization of the loop control variable
- test of the loop repetition condition
- change (update) of the loop control variable

- the for loop supplies a designated place for each of these three components

The for Statement Syntax

```
for   (initialization expression;
         loop repetition condition;
         update expression)
      statement;
```

```
/* Display N asterisks. */
for  (count_star = 0;
       count_star < N;
       count_star += 1)
    printf("*");
```

FIGURE 5.5 Using a for Statement in a Counting Loop

```
1.  /* Process payroll for all employees */
2.  total_pay = 0.0;
3.  for (count_emp = 0;                    /* initialization              */
4.       count_emp < number_emp;           /* loop repetition condition   */
5.       count_emp += 1) {                 /* update                      */
6.      printf("Hours> ");
7.      scanf("%lf", &hours);
8.      printf("Rate > $");
9.      scanf("%lf", &rate);
10.     pay = hours * rate;
11.     printf("Pay is $%6.2f\n\n", pay);
12.     total_pay = total_pay + pay;
13. }
14. printf("All employees processed\n");
15. printf("Total payroll is $%8.2f\n", total_pay);
```


Increment and Decrement Operators

- counter = counter + 1
  count += 1
  counter++

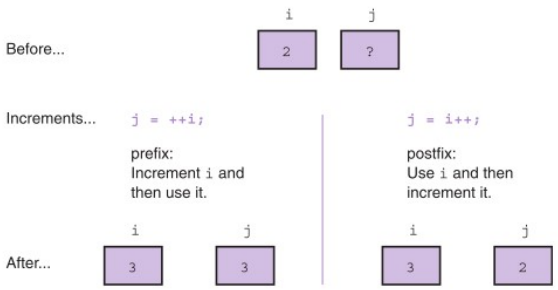
- counter = counter - 1
  count -= 1
  counter--

Increment and Decrement Operators

- side effect
  - a change in the value of a variable as a result of carrying out an operation

FIGURE 5.6

Comparison of Prefix and Postfix Increments



Computing Factorial

- loop body executes for decreasing value of i from n through 2
- each value of i is incorporated in the accumulating product
- loop exit occurs when i is 1

FIGURE 5.7 Function to Compute Factorial

```
1.  /*
2.   * Computes n!
3.   * Pre: n is greater than or equal to zero
4.   */
5.  int
6.  factorial(int n)
7.  {
8.      int i,          /* local variables */
9.          product;    /* accumulator for product computation */
10.
11.     product = 1;
12.     /* Computes the product n x (n-1) x (n-2) x . . . x 2 x 1 */
13.     for (i = n; i > 1; --i) {
14.          product = product * i;
15.     }
16.
17.     /* Returns function result */
18.     return (product);
19. }
```

Conversion of Celsius to Fahrenheit

- example shows conversions from 10 (CBEGIN) degree Celsius to -5 (CLIMIT) degrees Celsius
- loop update step subtracts 5 (CSTEP) from Celsius
  - accomplished by decreasing the value of the counter after each repetition
- loop exit occurs when Celsius becomes less than CLIMIT

FIGURE 5.8 Displaying a Celsius-to-Fahrenheit Conversion Table

```
1.  /* Conversion of Celsius to Fahrenheit temperatures */
2.
3.  #include <stdio.h>
4.
5.  /* Constant macros */
6.  #define CBEGIN 10
7.  #define CLIMIT -5
8.  #define CSTEP 5
9.
10. int
11. main(void)
12. {
13.         /* Variable declarations */
14.         int    celsius;
15.         double fahrenheit;
16.
17.         /* Display the table heading */
18.         printf("  Celsius   Fahrenheit\n");
19.
20.         /* Display the table */
21.  1      for  (celsius = CBEGIN;
22.  2            celsius >= CLIMIT;
23.  3            celsius -= CSTEP) {
24.  4          fahrenheit = 1.8 * celsius + 32.0;
25.  5          printf("%6c%3d%8c%7.2f\n", ' ', celsius, ' ', fahrenheit);
26.         }
27.
28.         return (0);
29. }
```

```
   Celsius     Fahrenheit
        10          50.00
         5          41.00
         0          32.00
        -5          23.00
```

TABLE 5.4 Trace of Loop in Fig. 5.8

	Statement	celsius	fahrenheit	Effect
1	`for (celsius = CBEGIN;`	10	?	Initialize `celsius` to 10
2	`celsius >= CLIMIT;`			`10 >= -5` is true
4	`fahrenheit = 1.8 * celsius + 32.0;`		50.0	Assign `50.0` to fahrenheit
5	`printf ...`			Display `10` and `50.0`
	Update and test `celsius`			
3	`... celsius -= CSTEP`	5		Subtract 5 from `celsius`, giving 5
2	`celsius >= CLIMIT;`			`5 >= -5` is true
4	`fahrenheit = 1.8 * celsius + 32.0;`		41.0	Assign `41.0` to `fahrenheit`
5	`printf ...`			Display `5` and `41.0`
	Update and test `celsius`			
3	`... celsius -= CSTEP`	0		Subtract 5 from `celsius`, giving 0
2	`celsius >= CLIMIT;`			`0 >= -5` is true
4	`fahrenheit = 1.8 * celsius + 32.0;`		32.0	Assign `32.0` to `fahrenheit`
5	`printf ...`			Display `0` and `32.0`
	Update and test `celsius`			
3	`... celsius -= CSTEP`	-5		Subtract 5 from `celsius`, giving `-5`
2	`celsius >= CLIMIT;`			`-5 >= -5` is true
4	`fahrenheit = 1.8 * celsius + 32.0;`		23.0	Assign `23.0` to `fahrenheit`
5	`printf ...`			Display `-5` and `23.0`
	Update and test `celsius`			
3	`... celsius -= CSTEP`	-10		Subtract 5 from `celsius`, giving `-10`
2	`celsius >= CLIMIT;`			`-10 >= -5` is false, so exit loop

Conditional Loops

- used when there are programming conditions when you will not be able to determine the exact number of loop repetitions before loop execution begins

FIGURE 5.9 Program to Monitor Gasoline Storage Tank

```
/*
 * Monitor gasoline supply in storage tank. Issue warning when supply
 * falls below MIN_PCT % of tank capacity.
 */

#include <stdio.h>

/* constant macros */
#define CAPACITY     80000.0  /* number of barrels tank can hold       */
#define MIN_PCT      10       /* warn when supply falls below this
                                 percent of capacity                   */
#define GALS_PER_BRL 42.0     /* number of U.S. gallons in one barrel  */

/* Function prototype */
double monitor_gas(double min_supply, double start_supply);

int
main(void)
{
        double start_supply, /* input - initial supply in barrels          */
               min_supply,   /* minimum number of barrels left without
                                warning                                   */
               current;      /* output - current supply in barrels        */

        /* Compute minimum supply without warning */
        min_supply = MIN_PCT / 100.0 * CAPACITY;

        /* Get initial supply */
        printf("Number of barrels currently in tank> ");
        scanf("%lf", &start_supply);

        /* Subtract amounts removed and display amount remaining
           as long as minimum supply remains.                            */
        current = monitor_gas(min_supply, start_supply);

        /* Issue warning                                                 */
        printf("only %.2f barrels are left.\n\n", current);
        printf("*** WARNING ***\n");
```

(continued)

FIGURE 5.9 (continued)

```
        printf("Available supply is less than %d percent of tank's\n",
               MIN_PCT);
        printf("%.2f-barrel capacity.\n", CAPACITY);

        return (0);
}

/*
 * Computes and displays amount of gas remaining after each delivery
 * Pre : min_supply and start_supply are defined.
 * Post: Returns the supply available (in barrels) after all permitted
 *       removals. The value returned is the first supply amount that is
 *       less than min_supply.
 */
double
monitor_gas(double min_supply, double start_supply)
{
        double remov_gals, /* input - amount of current delivery        */
               remov_brls, /*         in barrels and gallons            */
               current;    /* output - current supply in barrels        */

        for  (current = start_supply;
              current >= min_supply;
              current -= remov_brls) {
          printf("%.2f barrels are available.\n\n", current);
          printf("Enter number of gallons removed> ");
          scanf("%lf", &remov_gals);
          remov_brls = remov_gals / GALS_PER_BRL;

          printf("After removal of %.2f gallons (%.2f barrels),\n",
                 remov_gals, remov_brls);
        }

        return (current);
}
```

```
Number of barrels currently in tank> 8500.5
8500.50 barrels are available.
```

(continued)

FIGURE 5.9 (continued)

```
Enter number of gallons removed> 5859.0
After removal of 5859.00 gallons (139.50 barrels),
8361.00 barrels are available.

Enter number of gallons removed> 7568.4
After removal of 7568.40 gallons (180.20 barrels),
8180.80 barrels are available.

Enter number of gallons removed> 8400.0
After removal of 8400.00 gallons (200.00 barrels),
only 7980.80 barrels are left.

*** WARNING ***
Available supply is less than 10 percent of tank's
80000.00-barrel capacity.
```


Loop Design

- Sentinel-Controlled Loops
  - sentinel value: an end marker that follows the last item in a list of data
- Endfile-Controlled Loops
- Infinite Loops on Faulty Data

TABLE 5.5 Problem-Solving Questions for Loop Design

Question	Answer	Implications for the Algorithm

What are the inputs?	Initial supply of gasoline (barrels).
	Amounts removed (gallons).
		Input variables needed:
		`start_supply`
		`remov_gals`
		Value of `start_supply` must be input once, but amounts removed are entered many times.

What are the outputs?	Amounts removed in gallons and barrels, and the current supply of gasoline.	Values of `current` and `remov_gals` are echoed in the output. Output variable needed: `remov_brls`
Is there any repetition?	Yes. One repeatedly 1. gets amount removed 2. converts the amount to barrels 3. subtracts the amount removed from the current supply 4. checks to see whether the supply has fallen below the minimum.	Program variable needed: `min_supply`
Do I know in advance how many times steps will be repeated?	No.	Loop will not be controlled by a counter.
How do I know how long to keep repeating the steps?	As long as the current supply is not below the minimum.	The loop repetition condition is `current >= min_supply`

Sentinel Loop Design

- Correct 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

Sentinel Loop Design

- Incorrect Sentinel Loop
  1. Initialize sum to zero.
  2. while score is not the sentinel
     3. Get score
     4. Add score to sum.

FIGURE 5.10 Sentinel-Controlled while Loop

```
1.  /* Compute the sum of a list of exam scores. */
2.
3.  #include <stdio.h>
4.
5.  #define SENTINEL -99
6.
7.  int
8.  main(void)
9.  {
10.         int sum = 0,   /* output - sum of scores input so far              */
11.             score;     /* input - current score                            */
12.
13.         /* Accumulate sum of all scores.                                    */
14.         printf("Enter first score (or %d to quit)> ", SENTINEL);
15.         scanf("%d", &score);      /* Get first score.                       */
16.         while (score != SENTINEL) {
17.             sum += score;
18.             printf("Enter next score (%d to quit)> ", SENTINEL);
19.             scanf("%d", &score);  /* Get next score.                    */
20.         }
21.         printf("\nSum of exam scores is %d\n", sum);
22.
23.         return (0);
24. }
```

Endfile-Controlled Loop Design

1. Get the first *data value* 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*

FIGURE 5.11 Batch Version of Sum of Exam Scores Program

```
1.  /*
2.   *  Compute the sum of the list of exam scores stored in the
3.   *  file scores.txt
4.   */
5.  #include <stdio.h>
6.
7.  int
8.  main(void)
9.  {
10.       int sum = 0,      /* sum of scores input so far */
11.           score,        /* current score  */
12.           input_status; /* status value returned by scanf */
13.
14.       printf("Scores\n");
15.
16.       input_status = scanf("%d", &score);
17.       while (input_status != EOF) {
18.              printf("%5d\n", score);
19.              sum += score;
20.              input_status = scanf("%d", &score);
21.       }
22.
23.       printf("\nSum of exam scores is %d\n", sum);
24.
25.       return (0);
26. }

Scores
   55
   33
   77
Sum of exam scores is 165
```

Nested Loops

- Loops may be nested just like other control structures
- Nested loops consist of an outer loop with one or more inner loops
- Each time the outer loop is repeated, the inner loops are reentered, their loop control expressions are reevaluated, and all required iterations are performed

FIGURE 5.12 Program to Process Bald Eagle Sightings for a Year

```
/*
 * 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)

FIGURE 5.12 (continued)

```
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
. . .
```


FIGURE 5.13 Nested Counting Loop Program

```
1.  /*
2.   * Illustrates a pair of nested counting loops
3.   */
4.
5.  #include <stdio.h>
6.
7.  int
8.  main(void)
9.  {
10.       int i, j;   /* loop control variables */
11.
12.       printf("          i     j\n");        /* prints column labels          */
13.
14.       for (i = 1;   i < 4;   ++i) {         /* heading of outer for loop     */
15.          printf("Outer %6d\n", i);
16.          for   (j = 0;   j < i;   ++j) {    /* heading of inner loop         */
17.              printf("Inner %9d\n", j);
18.          }   /* end of inner loop */
19.       }  /*  end of outer loop */
20.
21.       return (0);
22. }
```

```
               i   j
Outer          1
  Inner            0
Outer          2
  Inner            0
  Inner            1
Outer          3
  Inner            0
  Inner            1
  Inner            2
```

do-while Statement

- For conditions where we know that a loop must execute at least one time

1. Get a *data value*
2. If *data value* isn't in the acceptable range, go back to step 1.

do-while Syntax

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

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

Flag-Controlled Loops for Input Validation

- Sometimes a loop repetition condition becomes so complex that placing the full expression in its usual spot is awkward
- Simplify the condition by using a **flag**

- flag
  - a type int variable used to represent whether or not a certain event has occurred
  - 1 (true) and 0 (false)

FIGURE 5.14 Validating Input Using do-while Statement

```
1.  /*
2.   * Returns the first integer between n_min and n_max entered as data.
3.   * Pre : n_min <= n_max
4.   * Post: Result is in the range n_min through n_max.
5.   */
6.  int
7.  get_int (int n_min, int n_max)
8.  {
9.          int   in_val,               /* input - number entered by user    */
10.               status;               /* status value returned by scanf    */
11.         char  skip_ch;              /* character to skip                 */
12.         int   error;                /* error flag for bad input          */
13.         /* Get data from user until in_val is in the range.              */
14.         do {
15.             /* No errors detected yet. */
16.             error = 0;
17.             /* Get a number from the user. */
18.             printf("Enter an integer in the range from %d ", n_min);
19.             printf("to %d inclusive> ", n_max);
20.             status = scanf("%d", &in_val);
21.
22.             /* Validate the number. */
23.             if (status != 1) { /* in_val didn't get a number */
24.                 error = 1;
25.                 scanf("%c", &skip_ch);
26.                 printf("Invalid character >>%c<<. ", skip_ch);
27.                 printf("Skipping rest of line.\n");
28.             } else if (in_val < n_min || in_val > n_max) {
29.                 error = 1;
30.                 printf("Number %d is not in range.\n", in_val);
31.             }
32.
33.             /* Skip rest of data line. */
34.             do
35.                 scanf("%c", &skip_ch);
36.             while (skip_ch != '\n');
37.         } while (error);
38.
39.         return (in_val);
40. }
```

Iterative Approximations

Numerical Analysis

FIGURE 5.15

Six Roots for the Equation $f(x) = 0$

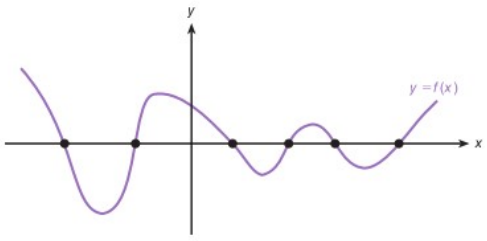


TABLE 5.6 Calls to Function evaluate and the Output Produced

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`

FIGURE 5.16 Using a Function Parameter

```
1.  /*
2.   * Evaluate a function at three points, displaying results.
3.   */
4.  void
5.  evaluate(double f(double f_arg), double pt1, double pt2, double pt3)
6.  {
7.        printf("f(%.5f) = %.5f\n", pt1, f(pt1));
8.        printf("f(%.5f) = %.5f\n", pt2, f(pt2));
9.        printf("f(%.5f) = %.5f\n", pt3, f(pt3));
10. }
```

Bisection Method for Finding Roots

Case Study

FIGURE 5.17

Change of Sign Implies an Odd Number of Roots

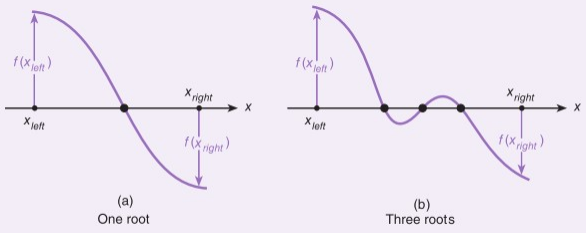
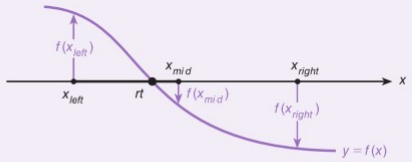
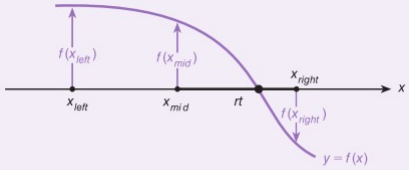


FIGURE 5.18

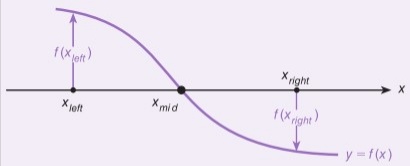
Three Possibilities That Arise When the Interval [x_{left}, x_{right}] Is Bisected



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

FIGURE 5.19 Finding a Function Root Using the Bisection Method

```
1.  /*
2.   *  Finds roots of the equations
3.   *       g(x) = 0   and    h(x) = 0
4.   *  on a specified interval [x_left, x_right] using the bisection method.
5.   */
6.
7.  #include <stdio.h>
8.  #include <math.h>
9.
10. #define FALSE 0
11. #define TRUE  1
12. #define NO_ROOT -99999.0
13.
14. double bisect(double x_left, double x_right, double epsilon,
15.                double f(double farg));
16. double g(double x);
17. double h(double x);
18.
19.
20. int
21. main(void)
22. {
23.       double x_left, x_right, /* left, right endpoints of interval  */
24.              epsilon,         /* error tolerance       */
25.              root;
26.
27.       /*  Get endpoints and error tolerance from user               */
28.       printf("\nEnter interval endpoints> ");
29.       scanf("%lf%lf", &x_left, &x_right);
30.       printf("\nEnter tolerance> ");
31.       scanf("%lf", &epsilon);
32.
33.       /*  Use bisect function to look for roots of g and h          */
34.       printf("\n\nFunction g");
35.       root = bisect(x_left, x_right, epsilon, g);
36.       if (root != NO_ROOT)
37.            printf("\n   g(%.7f) = %e\n", root, g(root));
38.
```

(continued)

FIGURE 5.19 (continued)

```
39.       printf("\n\nFunction h");
40.       root = bisect(x_left, x_right, epsilon, h);
41.       if (root != NO_ROOT)
42.             printf("\n   h(%.7f) = %e\n", root, h(root));
43.
44.       return (0);
45. }
46.
47. /*
48.  *  Implements the bisection method for finding a root of a function f.
49.  *  Returns a root if signs of fp(x_left) and fp(x_right) are different.
50.  *  Otherwise returns NO_ROOT.
51.  */
52. double
53. bisect(double x_left,          /* input  = endpoints of interval in */
54.        double x_right,         /*          which to look for a root */
55.        double epsilon,         /* input  = error tolerance          */
56.        double f(double farg))  /* input  = the function             */
57. {
58.       double x_mid,     /* midpoint of interval */
59.              f_left,    /* f(x_left)            */
60.              f_mid,     /* f(x_mid)             */
61.              f_right;   /* f(x_right)           */
62.
63.       int    root_found;  /* flag to indicate whether root is found */
64.
65.       /* Computes function values at initial endpoints of interval  */
66.       f_left = f(x_left);      f_right = f(x_right);
67.
68.       /* If no change of sign occurs on the interval there is not a
69.          unique root. Exit function and return NO_ROOT */
70.       if (f_left * f_right > 0) {     /* same sign */
71.           printf("\nMay be no root in [%.7f, %.7f]", x_left, x_right);
72.          return NO_ROOT;
73.       }
74.
75.       /*  Searches as long as interval size is large enough
76.           and no root has been found                          */
```

(continued)

FIGURE 5.19 (continued)

```
77.       root_found = FALSE;   /* no root found yet */
78.       while (fabs(x_right - x_left) > epsilon  &&  !root_found)
79.       {
80.            /* Computes midpoint and function value at midpoint */
81.            x_mid = (x_left + x_right) / 2.0;
82.            f_mid = f(x_mid);
83.
84.            if (f_mid == 0.0) {             /* Here's the root    */
85.                root_found = TRUE;
86.            } else if (f_left * f_mid < 0.0) {/* Root in [x_left,x_mid]*/
87.                x_right = x_mid;
88.            } else {                         /* Root in [x_mid,x_right]*/
89.                x_left = x_mid;
90.            }
91.
92.
93.            /* Trace loop execution - print root location or new interval */
94.            if (root_found)
95.                printf("\nRoot found at x = %.7f, midpoint of [%.7f, %.7f]",
96.                       x_mid, x_left, x_right);
97.            else
98.                printf("\nNew interval is [%.7f, %.7f]",
99.                       x_left, x_right);
100.      }
101.
102.      /*  If there is a root, it is the midpoint of [x_left, x_right]     */
103.      return ((x_left + x_right) / 2.0);
104. }
105.
106. /*  Functions for which roots are sought                             */
107.
108. /*    3     2
109.  *  5x  - 2x  + 3
110.  */
111. double
112. g(double x)
113. {
114.      return (5 * pow(x, 3.0) - 2 * pow(x, 2.0) + 3);
115. }
```

(continued)

FIGURE 5.19 (continued)

```
116.
117. /*   4     2
118.  *  x  - 3x  - 8
119.  */
120. double
121. h(double x)
122. {
123.       return (pow(x, 4.0) - 3 * pow(x, 2.0) - 8);
124. }
```


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]
```

Using Debugger Programs

- A debugger program can help you find defects in a C program
- It lets you execute your program one statement at a time (*single-step execution*).
- Use this to trace your program's execution and observe the effect of each C statement on variables you select.
- Separate your program into segments by setting *breakpoints*.

Debugging without a Debugger

- Insert extra *diagnostic* calls to printf that display intermediate results at critical points in your program.

```
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);  /* Get next score.                    */
}
```

Off-by-One Loop Errors

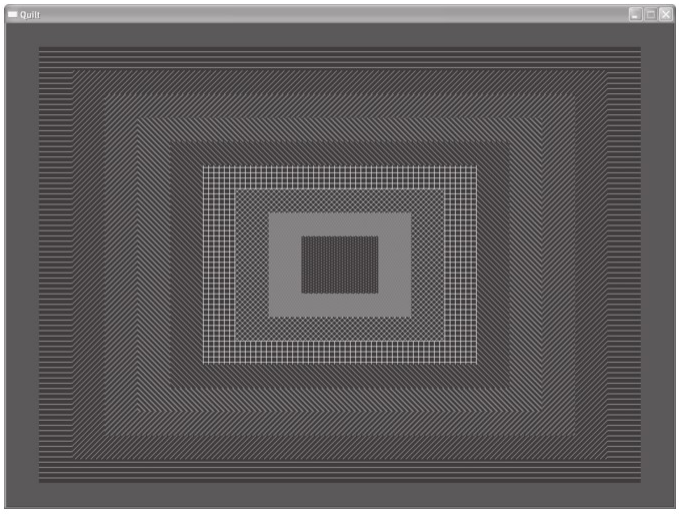
- A fairly common logic error in programs with loops is a loop that executes on more time or one less time than required.
- If a sentinel-controlled loop performs an extra repetition, it may erroneously process the sentinel value along with the regular data.
- loop boundaries
  - initial and final values of the loop control variable

Loops in Graphics Programs

Drawing a Quilt Example

FIGURE 5.21

Nested Rectangles for a Quilt Pattern



65

FIGURE 5.22 Program to Draw a Quilt

```
/*
 * Display a pattern for a quilt -- a set of nested rectangles
 */
#include <graphics.h>
#include <stdio.h>

int
main(void)
{
   int x1, y1, x2, y2;      /* coordinates of corner points */
   int stepX, stepY;        /* change in coordinate values */
   int foreColor;           /* foreground color */
   int numBars;             /* number of bars */
   int width, height;       /* screen width and height */

   printf("Enter number of bars> ");
   scanf("%d", &numBars);

   width = getmaxwidth();
   height = getmaxheight();

   initwindow(width, height, "Quilt");

   /* set corner points of outermost bar
      and increments for inner bars */
   x1 = 0;          y1 = 0;          /* top left corner */
   x2 = width;      y2 = height;     /* bottom right corner */
   stepX = width / (2 * numBars);    /* x increment */
   stepY = height / (2 * numBars);   /* y increment */

   for (int i = 1; i <= numBars; ++i)
   {
      foreColor = i % 16;            /* 0 <= foreColor <= 15 */
      setcolor(foreColor);
      setfillstyle(i % 12, foreColor);   /* Set fill style */
      bar(x1, y1, x2, y2);           /* Draw a bar */
      x1 = x1 + stepX;  y1 = y1 + stepY; /* Change top left corner */
      x2 = x2 - stepX;  y2 = y2 - stepY; /* Change bottom right */
   }

   getch();       /* pause for user */
   closegraph();
   return (0);
}
```


66

Loops in Graphics Programs

Moving a Ball Example

Animation

- graphics animation
  - motion achieved by displaying series of frames with object in a slightly different position from one frame to the next
- single buffering
  - the default case in which only one memory area is allocated
- buffer
  - an area of memory where data to be displayed or printed is temporarily stored

Animation

- double buffering
  - a technique used in graphics programming to reduce display flicker by allocating two buffers
  - the second buffer is filled while the contents of the first buffer is displayed and then the roles of the buffers are reversed

FIGURE 5.23 Program to Draw a Moving Ball

```
1.  /*
2.   * Draw a ball that moves around the screen
3.   */
4.  #include <graphics.h>
5.  #define TRUE  1
6.
7.  int
8.  main(void)
9.  {
10.     const int PAUSE = 20;          /* delay between frames */
11.     const int DELTA = 5;           /* change in x or y value */
12.     const int RADIUS = 30;         /* ball radius */
13.     const int COLOR = RED;
14.
15.     int width;                     /* width of screen */
16.     int height;                    /* height of screen */
17.     int x; int y;                  /* center of ball */
18.     int stepX;                     /* increment for x */
19.     int stepY;                     /* increment for y */
```

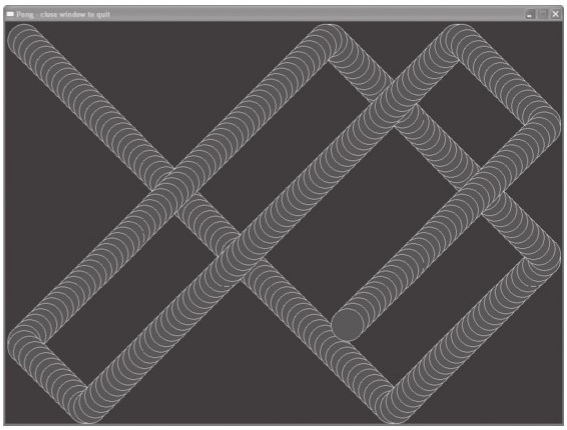
(continued)

FIGURE 5.23 (continued)

```
20.
21.    /* Open a full-screen window with double buffering */
22.     width = getmaxwidth();
23.     height = getmaxheight();
24.     initwindow(width, height,
25.                "Pong - close window to quit", 0, 0, TRUE);
26.     x = RADIUS;  y = RADIUS;       /* Start ball at top-left corner */
27.     stepX = DELTA;  stepY = DELTA; /* Move down and to the right */
28.
29.     /* Draw the moving ball */
30.     while (TRUE)    /* Repeat forever */
31.     {
32.         /* Clear the old frame and draw the new one. */
33.         clearviewport();
34.         setfillstyle(SOLID_FILL, COLOR);
35.         fillellipse(x, y, RADIUS, RADIUS);  /* Draw the ball */
36.
37.         /* After drawing the frame, swap the buffers */
38.         swapbuffers();
39.         delay(PAUSE);
40.
41.         /* If ball is too close to window edge, change direction          */
42.         if (x <= RADIUS)               /* Is ball too close to left edge?  */
43.            stepX = DELTA;              /* Move right                       */
44.         else if (x >= width - RADIUS)  /* Is ball too close to right edge? */
45.            stepX = -DELTA;             /* Move left                        */
46.
47.         if (y <= RADIUS)               /* Is ball too close to top?        */
48.            stepY = DELTA;              /* Move down                        */
49.         else if (y >= height - RADIUS) /* Is ball too close to bottom?     */
50.            stepY = -DELTA;             /* Move up                          */
51.
52.
53.        /* Move the ball */
54.        x = x + stepX;
55.        y = y + stepY;
56.     }
57.
58.    closegraph();
59.    return (0);
60. }
```


FIGURE 5.24

Trace of the Moving Ball



Wrap Up

- Use a loop to repeat steps in a program
- Frequently occuring loops
  - counter-controlled loop
  - sentinel-controlled loop
- Other useful loops
  - endfile-controlled loop
  - input validation loop
  - general conditional loop