مبانی برنامه نویسی
به زبان سی

۲۵، ۲۷ و ۲۹ آبان ۱۳۹۹

جلسه پنجم، ششم و هفتم

ملکی مجد

مباحث این هفته:

- حلقه تکرار while
  - نمونه موردی برای محاسبه میانگین نمرات یک کلاس ده نفره
  - نمونه موردی برای محاسبه میانگین نمرات یک کلاس با تعداد متغیر و استفاده از sentinel
- کنترل های تو در تو
- تعداد عملگر جدید
  - -- ++ =- =+

- حلقه های با تکرار مشخص
  - for

3.7 The `while` Repetition Statement

- A repetition statement allows you to specify that an action is to be repeated while some condition remains true.
  - The pseudocode statement
    - While there are more items on my shopping list
      Purchase next item and cross it off my list
  - The condition, "there are more items on my shopping list" may be true or false.
    - If it's true, then the action, "Purchase next item and cross it off my list" is performed.
  - This action will be performed repeatedly while the condition remains true.
  - 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 repetition terminates, and the first pseudocode statement after the repetition structure is executed.

Common Programming Error 3.3

Not providing the body of a `while` statement with an action that eventually causes the condition in the `while` to become false. Normally, such a repetition structure will never terminate—an error called an "infinite loop."

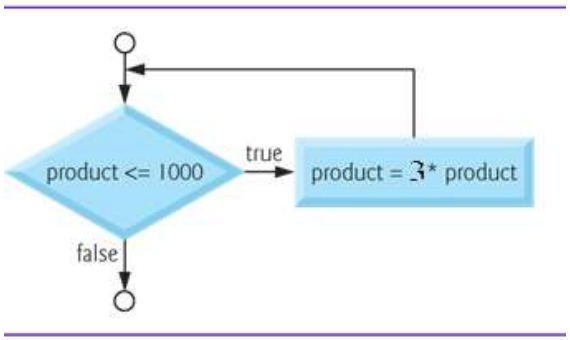
3.7 The `while` Repetition Statement (Cont.)

- *the body of the while :*
  - The statement(s) contained in the *while repetition statement constitute.*
  - The *while statement body may be a single statement or a compound statement.*

3.7 The `while` Repetition Statement (Cont.)

- As an example of `while`,
  - consider a program segment designed to find the first power of 3 larger than 100.

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



3.7 The `while` Repetition Statement (Cont.)

- When the `while` statement is entered, the value of `product` is 3.
  - The variable `product` is repeatedly multiplied by 3, taking on the values 9, 27 and 81 successively.
  - When `product` becomes 243, the condition in the `while` statement, `product <= 100`, becomes false.
  - This terminates the repetition, and the final value of `product` is 243.
  - Program execution continues with the next statement after the `while`.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition

- To illustrate how algorithms are developed, we solve several variations of a class averaging problem.

- Consider the following problem statement:
  - *A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.*

  - The class average is equal to the sum of the grades divided by the number of students.
  - The algorithm for solving this problem on a computer must input each of the grades, perform the averaging calculation, and print the result.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition (Cont.)

- Let's use pseudocode
  - to list the actions to execute and specify the order in which these actions should execute.

- We use counter-controlled repetition 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, repetition terminates when the counter exceeds 10.
  - Counter-controlled repetition is often called definite repetition because the number of repetitions is known before the loop begins executing.

```
1  Set total to zero
2  Set grade counter to one
3
4  While grade counter is less than or equal to ten
5      Input the next grade
6      Add the grade into the total
7      Add one to the grade counter
8
9  Set the class average to the total divided by ten
10 Print the class average
```

Fig. 3.5 | Pseudocode algorithm that uses counter-controlled repetition to solve the class average problem.

```c
/* Fig. 3.6: fig03_06.c
   Class average program with counter-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter; /* number of grade to be entered next */
   int grade; /* grade value */
   int total; /* sum of grades input by user */
   int average; /* average of grades */

   /* initialization phase */
   total = 0; /* initialize total */
   counter = 1; /* initialize loop counter */

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

   /* termination phase */
   average = total / 10; /* integer division */

   printf( "Class average is %d\n", average ); /* display result */
   return 0; /* indicate program ended successfully */
} /* 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. 3.6 | C program and sample execution for the class average problem with counter-controlled repetition.

A total is a variable used to accumulate the sum of a series of values.
A counter is a variable used to count—in this case, to count the number of grades entered.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition (Cont.)

- Variables used to store totals should normally 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.
- Counter variables are normally initialized to zero or one, depending on their use.

- An uninitialized variable contains a "garbage" value—the value last stored in the memory location reserved for that variable.

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

- **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, the number of grades (10) was known in advance. In this example, the program must process an **arbitrary number of grades**.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?

- One way to solve this problem is to use a **special value called a sentinel value**
  - (also called a signal value, a dummy value, or a flag value) to indicate "end of data entry."

- The user types in grades until all legitimate grades have been entered.
- The user then types the sentinel value to indicate that the last grade has been entered.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Clearly, the sentinel value must be chosen so that it cannot be confused with an acceptable input value.
  - Since grades on a quiz are normally nonnegative integers, –1 is an acceptable sentinel value for this problem.
- Thus, a run of the class average program might process a stream of inputs such as

  95, 96, 75, 74, 89 and –1.

  - The program would then compute and print the class average for the grades 95, 96, 75, 74, and 89 (–1 is the sentinel value, so it should not enter into the averaging calculation).

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

Top-Down, Stepwise Refinement

- We approach the class average program with a technique called top-down, stepwise refinement, a technique that is essential to the development of 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.
  - the top rarely conveys a sufficient amount of detail for writing the C program.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- begin the refinement process:
  - We divide the top into a series of smaller tasks and list these 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 are to be executed in order, one after the other.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

the second refinement, we commit to specific variables:

- We need a running total of the numbers,
- a count of how many numbers have been processed,
- a variable to receive the value of each grade as it's input and
- a variable to hold the calculated average.

- The pseudocode statement
  - Initialize variables
- may be refined as follows:
  - Initialize total to zero
    Initialize counter to zero

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Only total and counter need to be initialized; the variables average and grade (for the calculated average and the user input, respectively) need not be initialized because their values will be written over by the process of destructive.

- The pseudocode statement
  - Input, sum, and count the quiz grades
- requires a repetition structure (i.e., a loop) that successively inputs each grade.
- Since we do not know in advance how many grades are to be processed, we'll use sentinel-controlled repetition.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- The user will type legitimate grades in one at a time.
- After the last legitimate grade is typed, 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

    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)

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- 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"

- Notice that 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 "bombing" or "crashing").

```
1   Initialize total to zero
2   Initialize counter to zero
3
4   Input the first grade
5   While the user has not as yet entered the sentinel
6       Add this grade into the running total
7       Add one to the grade counter
8       Input the next grade (possibly the sentinel)
9
10  If the counter is not equal to zero
11      Set the average to the total divided by the counter
12      Print the average
13  else
14      Print "No grades were entered"
```

Fig. 3.7 | Pseudocode algorithm that uses sentinel-controlled repetition to solve the class average problem.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- We use the data type `float` to handle numbers with decimal points (called floating-point numbers) and
- introduces a special operator called a cast operator to handle the averaging calculation.

```c
/* Fig. 3.8: fig03_08.c
   Class average program with sentinel-controlled repetition */
#include <stdio.h>

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

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

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

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

   /* loop while sentinel value not yet read from user */
   while ( grade != -1 ) {
      total = total + grade; /* add grade to total */
      counter = counter + 1; /* increment counter */

      /* get next grade from user */
      printf( "Enter grade, -1 to end: " ); /* prompt for input */
      scanf("%d", &grade); /* read next grade */
   } /* end while */
   /* termination phase */
   /* if user entered at least one grade */
   if ( counter != 0 ) {

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

      /* display average with two digits of precision */
      printf( "Class average is %.2f\n", average );
   } /* end if */
   else { /* if no grades were entered, output message */
      printf( "No grades were entered\n" );
   } /* end else */

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

```
Enter grade, -1 to end: 75
Enter grade, -1 to end: 94
Enter grade, -1 to end: 97
Enter grade, -1 to end: 88
Enter grade, -1 to end: 70
Enter grade, -1 to end: 64
Enter grade, -1 to end: 83
Enter grade, -1 to end: 89
Enter grade, -1 to end: -1
Class average is 82.50
```

```
Enter grade, -1 to end: -1
No grades were entered
```

Fig. 3.8 | C program and sample execution for the class average problem with sentinel-controlled repetition.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- What is the problem of following code?

- ```
  while ( grade != -1 )
     total = total + grade; /* add grade to total */
     counter = counter + 1; /* increment counter */
     printf( "Enter grade, -1 to end: " ); /* prompt for input */
     scanf( "%d", &grade ); /* read next grade */
  ```
  ```

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- What is the problem of following code?

```
while ( grade != -1 )
   total = total + grade; /* add grade to total */
   counter = counter + 1; /* increment counter */
   printf( "Enter grade, -1 to end: " ); /* prompt for input */
   scanf( "%d", &grade ); /* read next grade */
```

- This would cause an infinite loop if the user did not input `-1` for the first grade.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Averages do not always evaluate to integer values.
  - such as 7.2 or –93.5 that contains a fractional part.
- These values are referred to as **floating-point numbers** and are represented by the data type `float`.
  - The variable `average` is defined to be of type `float` (line 12) to capture the fractional result of our calculation.

- However, the result of the calculation `total / counter` is an integer because `total` and `counter` are both integer variables.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Dividing two integers results in integer division in which any fractional part of the calculation is lost (i.e., truncated).
- Since the calculation is performed first, the fractional part is lost before the result is assigned to `average`.
- To produce a floating-point calculation with integer values, we must create temporary values that are floating-point numbers.

- C provides the unary **cast operator** to accomplish this task.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- CAST
  - `average = ( float ) total / counter;`
  - includes the cast operator `(float)`, which creates a temporary floating-point copy of its operand, `total`.
- The value stored in `total` is still an integer.
- Using a cast operator in this manner is called explicit conversion.

- To ensure that the operands are of the same type, the compiler performs an operation called promotion (also called implicit conversion) on selected operands.
- For example, in an expression containing the data types `int` and `float`, copies of `int` operands are made and promoted to `float`.
- The calculation now consists of a floating-point value (the temporary `float` version of `total`) divided by the integer value stored in `counter`.
- After a copy of `counter` is made and promoted to `float`, the calculation is performed and the result of the floating-point division is assigned to `average`.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Cast operators are available for most data types.
- The cast operator is formed by placing parentheses around a data type name.
- The cast operator is a unary operator, i.e., an operator that takes only one operand.
- Cast operators associate from right to left and have the same precedence as other unary operators such as unary + and unary -.
- This precedence is one level higher than that of the multiplicative operators *, / and %.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- conversion specifier `%.2f` (to print the value of `average`):
  - The `f` specifies that a floating-point value will be printed.
  - The `.2` is the precision with which the value will be displayed—with 2 digits to the right of the decimal point.
  - If the `%f` conversion specifier is used (without specifying the precision), the default precision of 6 is used—exactly as if the conversion specifier `%.6f` had been used.
  - When floating-point values are printed with precision, the printed value is rounded to the indicated number of decimal positions.
    - The value in memory is unaltered.
  - When the following statements are executed, the values 3.45 and 3.4 are printed.

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

Common Programming Error 3.8

Using precision in a conversion specification in the format control string of a `scanf` *statement is wrong. Precisions are used only in* `printf` *conversion specifications.*

Common Programming Error 3.9

Using floating-point numbers in a manner that assumes they're represented precisely can lead to incorrect results. Floating-point numbers are represented only approximately by most computers.

Error-Prevention Tip 3.3

Do not compare floating-point values for equality.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Despite the fact that floating-point numbers are not always "100% precise," they have numerous applications.

- For example, when we speak of a "normal" body temperature of 98.6, we do not need to be precise to a large number of digits.
- When we view the temperature on a thermometer and read it as 98.6, it may actually be 98.5999473210643.
- The point here is that calling this number simply 98.6 is fine for most applications.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 2: Sentinel-Controlled Repetition (Cont.)

- Another way floating-point numbers develop is through division.
- When we divide 10 by 3, the result is 3.3333333… with the sequence of 3s repeating infinitely.
- The computer allocates only a fixed amount of space to hold such a value, so clearly the stored floating-point value can be only an **approximation**.

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

- Another complete problem.
  - We'll once again formulate the algorithm using pseudocode and top-down, stepwise refinement, and write a corresponding C program.

- We've seen that control statements may be stacked on top of one another (in sequence) just as a child stacks building blocks.

- In this case study we'll see the only other structured way control statements may be connected in C, namely through nesting of one control statement within another.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- A problem:
  - A college offers a course that prepares students for the state licensing exam for real estate brokers.
    - Last year, 10 of the students who completed this course took the licensing examination.
    - Naturally, the college wants to know how well its students did on the exam.
  - You have been asked to write a program to **summarize the results**. You have been given a list of these 10 students. Next to each name a 1 is written if the student passed the exam and a 2 if the student failed.
- Your program should analyze the results of the exam as follows:
  - Input each test result (i.e., a 1 or a 2). D
    - isplay the prompting message "Enter result" each time the program requests another test result.
  - Count the number of test results of each type.
  - Display a summary of the test results indicating the number of students who passed and the number who failed.
    - If more than eight students passed the exam, print the message "Bonus to instructor!"

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- After **reading** the problem statement carefully,
- we make the following observations:
  - The program must process 10 test results. A **counter-controlled loop** will be used.
  - Each test result is a number—either a 1 or a 2. Each time the program reads a test result, the program must determine if the number is a 1 or a 2.
    - We test for a 1 in our algorithm. If the number is not a 1, we assume that it's a 2.
  - **Two counters** are used—one to count the number of students who passed the exam and one to count the number of students who failed the exam.
  - After the program has processed all the results, it must **decide** if more than 8 students passed the exam.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- Let's proceed with top-down, stepwise refinement.

- We begin with a pseudocode representation of the top:
  - **Analyze exam results and decide if instructor should receive a bonus**

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

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- Our first refinement is
  - Initialize variables
    Input the ten quiz grades and count passes and failures
    Print a summary of the exam results and decide if
    instructor should receive a bonus

  - we have a complete representation of the entire program,
  - further refinement is necessary.

  - Counters are needed to record the passes and failures,
    - a counter will be used to control the looping process, and
  - a variable is needed to store the user input.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- The pseudocode statement
  - Initialize variables

- may be refined as follows:
  - Initialize passes to zero
    Initialize failures to zero
    Initialize student to one

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- The pseudocode statement
  - Input the ten quiz grades and count passes and failures

requires a loop that successively inputs the result of each exam.

Here it's known in advance that there are precisely ten exam results, so counter-controlled looping is appropriate.

Inside the loop (i.e., nested within the loop) a double-selection statement will determine whether each exam result is a pass or a failure,

and will increment the appropriate counters accordingly.

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- 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

3.10 Formulating Algorithms with Top-Down, Stepwise Refinement Case Study 3: Nested Control Structures (Cont.)

- The pseudocode statement
  - Print a summary of the exam results and decide if instructor should receive a bonus
- may be refined as follows:
  - Print the number of passes
    Print the number of failures
    If more than eight students passed
    Print “Bonus to instructor!”

```
1  Initialize passes to zero
2  Initialize failures to zero
3  Initialize student to one
4
5  While student counter is less than or equal to ten
6          Input the next exam result
7
8          If the student passed
9              Add one to passes
10         else
11             Add one to failures
12
13         Add one to student counter
14
15 Print the number of passes
16 Print the number of failures
17 If more than eight students passed
18     Print "Bonus to instructor!"
```

Fig. 3.9 | Pseudocode for examination results problem.

```c
/* Fig. 3.10: fig03_10.c
   Analysis of examination results */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   /* initialize variables in definitions */
   int passes = 0; /* number of passes */
   int failures = 0; /* number of failures */
   int student = 1; /* student counter */
   int result; /* one exam result */

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

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

      /* if result 1, increment passes */
      if ( result == 1 ) {
         passes = passes + 1;
      } /* end if */
      else {/* otherwise, increment failures */
         failures = failures + 1;
      } /* end else */

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

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

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

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

```c
/* Fig. 3.10: fig03_10.c
   Analysis of examination results */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   /* initialize variables in definitions */
   int passes = 0; /* number of passes */
   int failures = 0; /* number of failures */
   int student = 1; /* student counter */
   int result; /* one exam result */

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

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

      /* if result 1, increment passes */
      if ( result == 1 ) {
         passes = passes + 1;
      } /* end if */
      else {/* otherwise, increment failures */
         failures = failures + 1;
      } /* end else */

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

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

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

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

```
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
```

```
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!
```

Software Engineering Observation 3.6

Experience has shown that the most difficult part of solving a problem on a computer is developing the algorithm for the solution. Once a correct algorithm has been specified, the process of producing a working C program is normally straightforward.

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 right of the operator to the value of the variable on the left of the operator and stores the result in the variable on the left of the operator.

3.11 Assignment Operators (Cont.)

- Any statement of the form
  - *variable = variable operator expression;*
- where *operator is one of the binary operators +, -, *, / or %, can be written in the form*
  - *variable operator= expression;*
- Thus the assignment c += 3 adds 3 to c.

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

3.12 Increment and Decrement Operators

- C also provides the unary increment operator, `++`, and the unary decrement operator, `--`,
- If a variable `c` is incremented by 1, the increment operator `++` can be used rather than the expressions `c = c + 1` or `c += 1`.

- If increment or decrement operators are placed before a variable (i.e., prefixed),
  - they're referred to as the preincrement or predecrement operators, respectively.
- If increment or decrement operators are placed after a variable (i.e., postfixed),
  - they're referred to as the postincrement or postdecrement operators, respectively.

3.12 Increment and Decrement Operators (Cont.)

- Preincrementing (predecrementing) a variable causes the variable to be incremented (decremented) by 1**, then the new value of the variable is used** in the expression in which it appears.
- Postincrementing (postdecrementing) the variable causes the current value of the variable to be used in the expression in which it appears, **then the variable value is incremented** (decremented) by 1.

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.

Unary operators should be placed directly next to their operands with no intervening spaces.

3.12 Increment and Decrement Operators (Cont.)

- Figure 3.13 demonstrates the difference between the preincrementing and the postincrementing versions of the ++ operator.
- Postincrementing the variable `c` causes it to be incremented after it's used in the `printf` statement.
- Preincrementing the variable `c` causes it to be incremented before it's used in the `printf` statement.

```c
/* Fig. 3.13: fig03_13.c
   Preincrementing and postincrementing */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int c; /* define variable */

   /* demonstrate postincrement */
   c = 5; /* assign 5 to c */
   printf( "%d\n", c ); /* print 5 */
   printf( "%d\n", c++ ); /* print 5 then postincrement */
   printf( "%d\n\n", c ); /* print 6 */

   /* demonstrate preincrement */
   c = 5; /* assign 5 to c */
   printf( "%d\n", c ); /* print 5 */
   printf( "%d\n", ++c ); /* preincrement then print 6 */
   printf( "%d\n", c ); /* print 6 */
   return 0; /* indicate program ended successfully */
} /* end function main */
```

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.

```
5
5
6
 s
5
6
6
```

Fig. 3.13 | Preincrementing vs. postincrementing. (Part 1 of 2.)

3.12 Increment and Decrement Operators (Cont.)

- The three assignment statements

```
passes = passes + 1;
failures = failures + 1;
student = student + 1;
```

can be written :

```
passes += 1;
failures += 1;
student += 1;
```

or

```
++passes;
++failures;
++student;
```

or

```
passes++;
failures++;
student++;
```

Common Programming Error 3.10

Attempting to use the increment or decrement operator on an expression other than a simple variable name is a syntax error, e.g., writing `++(x + 1)`.

Error-Prevention Tip 3.4

C generally does not specify the order in which an operator's operands will be evaluated (although we'll see exceptions to this for a few operators in Chapter 4). Therefore you should avoid using statements with increment or decrement operators in which a particular variable being incremented or decremented appears more than once.

3.12 Increment and Decrement Operators (Cont.)

- The operators are shown top to bottom in decreasing order of precedence.
- The second column describes the associativity of the operators at each level of precedence.

The operators are shown top to bottom in decreasing order of precedence.
The second column describes the associativity of the operators at each level of precedence.

Operators	Associativity	Type
++ *(postfix)* -- *(postfix)*	right to left	postfix
+ - *(type)* ++ *(prefix)* -- *(prefix)*	right to left	unary
* / %	left to right	multiplicative
+ -	left to right	additive
< <= > >=	left to right	relational
== !=	left to right	equality
?:	right to left	conditional
= += -= *= /= %=	right to left	assignment

Fig. 3.14 | Precedence and associativity of the operators encountered so far in the text.

4.2 Repetition Essentials

- A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.

- We have discussed two means of repetition:
  - Counter-controlled repetition
  - Sentinel-controlled repetition
- Counter-controlled repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed.
- Sentinel-controlled repetition is sometimes called indefinite repetition because it's not known in advance how many times the loop will be executed.

4.2 Repetition Essentials (Cont.)

- In counter-controlled repetition, a control variable is used to count the number of repetitions.
  - The control variable is incremented (usually by 1) each time the group of instructions is performed.
- When the value of the control variable indicates that the correct number of repetitions has been performed,
  - the loop terminates and the computer continues executing with the statement after the repetition statement.

4.2 Repetition Essentials (Cont.)

- Sentinel values are used to control repetition when:
  - The precise number of repetitions is not known in advance, and
  - The loop includes statements that obtain data each time the loop is performed.
- The sentinel value indicates “end of data.”
- The sentinel is entered after all regular data items have been supplied to the program.
  - Sentinels must be distinct from regular data items.

4.3 Counter-Controlled Repetition

- Counter-controlled repetition requires:
  - The name of a control variable (or loop counter).
  - The initial value of the control variable.
  - The increment (or decrement) by which the control variable is modified each time through the loop.
  - The condition that tests for the final value of the control variable (i.e., whether looping should continue).

```c
/* Fig. 4.1: fig04_01.c
   Counter-controlled repetition */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int counter = 1; /* initialization */

   while ( counter <= 10 ) { /* repetition condition */
      printf ( "%d\n", counter ); /* display counter */
      ++counter; /* increment */
   } /* end while */

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

Fig. 4.1 | Counter-controlled repetition. (Part 1 of 2.)

```c
while ( ++counter <= 10 )
   printf( "%d\n", counter );
```

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

Fig. 4.1 | Counter-controlled repetition. (Part 2 of 2.)

Error-Prevention Tip 4.1

Control counting loops with integer values.

Good Programming Practice 4.1

Too many levels of nesting can make a program difficult to understand. As a rule, try to avoid using more than three levels of nesting.

Good Programming Practice 4.2

The combination of vertical spacing before and after control statements and indentation of the bodies of control statements within the control-statement headers gives programs a two-dimensional appearance that greatly improves program readability.

4.4 for Repetition Statement

- The **for** repetition statement handles all the details of counter-controlled repetition.

```
1  /* Fig. 4.2: fig04_02.c
2     Counter-controlled repetition with the for statement */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     int counter; /* define counter */
9
10    /* initialization, repetition condition, and increment
11       are all included in the for statement header. */
12    for ( counter = 1; counter <= 10; counter++ ) {
13       printf( "%d\n", counter );
14    } /* end for */
15
16    return 0; /* indicate program ended successfully */
17 } /* end function main */
```

Fig. 4.2 | Counter-controlled repetition with the `for` statement.

4.4 `for` Repetition Statement (Cont.)

- When the `for` statement begins executing, the control variable `counter` is initialized to `1`.
- Then, the loop-continuation condition `counter <= 10` is checked.
- Because the initial value of `counter` is `1`, the condition is satisfied, so the `printf` statement (line 13) prints the value of `counter`, namely `1`.
- The control variable `counter` is then incremented by the expression `counter++`, and the loop begins again with the loop-continuation test.

```
6  int main( void )
7  {
8     int counter; /* define counter */
9
10    /* initialization, repetition condition, and increment
11       are all included in the for statement header. */
12    for ( counter = 1; counter <= 10; counter++ ) {
13       printf( "%d\n", counter );
14    } /* end for */
15
16    return 0; /* indicate program ended successfully */
17 } /* end function main */
```

4.4 `for` Repetition Statement (Cont.)

- Since the control variable is now equal to `2`, the final value is not exceeded, so the program performs the `printf` statement again.
- This process continues until the control variable `counter` is incremented to its final value of 11—
  - this causes the loop-continuation test to fail, and repetition terminates.
- Notice that the `for` statement "does it all"—it specifies each of the items needed for counter-controlled repetition with a control variable.
- If there is more than one statement in the body of the `for`, braces are required to define the body of the loop.

```
6  int main( void )
7  {
8     int counter; /* define counter */
9
10    /* initialization, repetition condition, and increment
11       are all included in the for statement header. */
12    for ( counter = 1; counter <= 10; counter++ ) {
13       printf( "%d\n", counter );
14    } /* end for */
15
16    return 0; /* indicate program ended successfully */
17 } /* end function main */
```

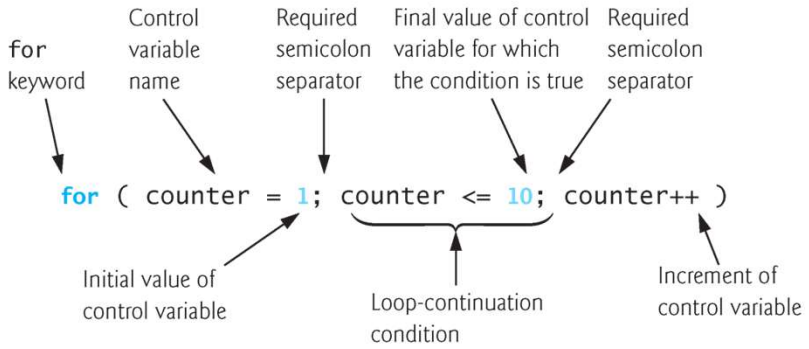


Fig. 4.3 | `for` statement header components.

4.4 `for` Repetition Statement (Cont.)

- What if you wrote `counter < 10`?

```
6  int main( void )
7  {
8     int counter; /* define counter */
9
10    /* initialization, repetition condition, and increment
11       are all included in the for statement header. */
12    for ( counter = 1; counter <= 10; counter++ ) {
13       printf( "%d\n", counter );
14    } /* end for */
15
16    return 0; /* indicate program ended successfully */
17 } /* end function main */
```

4.4 `for` Repetition Statement (Cont.)

- What if you wrote `counter < 10`?
  - then the loop would be executed only 9 times.
- If you do it by mistake:
  - This is a common logic error called an off-by-one error.

4.4 `for` Repetition Statement (Cont.)

- The general format of the `for` statement is
  - `for ( expression1; expression2; expression3)`
    `statement`

  where *expression1* initializes the loop-control variable,
  expression2 is the loop-continuation condition, and
  expression3 increments the control variable.

- In most cases, the `for` statement can be represented with an **equivalent** `while` statement as follows:

```
expression1;
while ( expression2 ) {
   statement
   expression3;
}
```

4.4 `for` Repetition Statement (Cont.)

- Often, *expression1* and *expression3* are comma-separated lists of expressions.
- The commas as used here are actually comma operators that guarantee that lists of expressions evaluate from left to right.
- The value and type of a comma-separated list of expressions are the value and type of the right-most expression in the list.
- The comma operator is most often used in the `for` statement.
- Its primary use is to enable you to use multiple initialization and/or multiple increment expressions.
- For example, there may be two control variables in a single `for` statement that must be initialized and incremented.

4.4 `for` Repetition Statement (Cont.)

- The two semicolons in the `for` statement are **required**.
- The three expressions in the `for` statement are **optional**.
  - If *expression2* is omitted, C assumes that the condition is true, thus creating an infinite loop.
  - One may omit *expression1* if the control variable is initialized elsewhere in the program.
  - *expression3* may be omitted if the increment is calculated by statements in the body of the `for` statement or if no increment is needed.
    - The increment expression in the `for` statement acts like a stand-alone C statement at the end of the body of the `for`.

4.4 `for` Repetition Statement (Cont.)

- Therefore, the expressions

```
counter = counter + 1
counter += 1
++counter
counter++
```

  are all equivalent in the increment part of the `for` statement.

4.5 `for` Statement: Notes and Observations

- The initialization, loop-continuation condition and increment can contain arithmetic expressions. For example, if `x = 2` and `y = 10`, the statement

```
for ( j = x; j <= 4 * x * y; j += y / x )
```

  is (at first) equivalent to the statement

```
for ( j = 2; j <= 80; j += 5 )
```

- The "increment" may be negative (in which case it's really a decrement and the loop actually counts downward).
- If the loop-continuation condition is initially false, the loop body does not execute. Instead, execution proceeds with the statement following the `for` statement.

```
for ( counter = 1; counter <= 10; counter++ )
   printf( "%d", counter );
```

This flowchart makes it clear that the **initialization occurs only once** and that **incrementing occurs after the body statement is performed**.

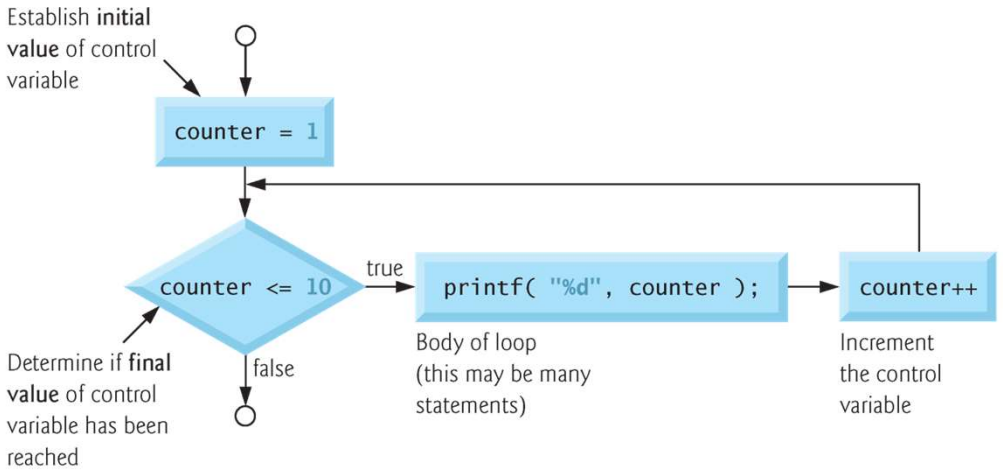


Fig. 4.4 | Flowcharting a typical `for` repetition statement.