3

Structured Program Development in C

Outline

3.1 Introduction
3.2 Algorithms
3.3 Pseudocode
3.4 Control Structures
3.5 The `if` Selection Statement
3.6 The `if...else` Selection Statement
3.7 The `while` Repetition Statement
3.8 Counter-Controlled Repetition (Loop)
3.9 Sentinel-Controlled Repetition (Loop)
3.10 Nested Control Structures
3.11 Assignment Operators
3.12 Increment and Decrement Operators

3.1 Introduction

- **Before writing a program:**
  - **Have a thorough understanding of the problem**
  - **Carefully plan an approach for solving it**
- **While writing a program:**
  - **Know what “building blocks” are available**
  - **Use good programming principles**

3.2 Algorithms

- Computing problems
  - All can be solved by executing a series of actions in a specific order
- Algorithm: procedure in terms of
  - Actions to be executed
  - The order in which these actions are to be executed
- Program control
  - Specify order in which statements are to be executed

3.3 Pseudocode

- Pseudocode
  - Artificial, informal language that helps us develop algorithms
  - Similar to everyday English
  - Not actually executed on computers
  - Helps us "think out" a program before writing it
    - Easy to convert into a corresponding C program
    - Consists only of executable statements

3.4 Control Structures

- **Sequential execution**
  - **Statements executed one after the other in the order written**
- **Transfer of control**
  - **When the next statement executed is not the next one in sequence**
  - **Overuse of `goto` statements led to many problems**
- **Bohm and Jacopini**
  - **All programs written in terms of 3 control structures**
    - **Sequence structures: Built into C. Programs executed sequentially by default**
    - **Selection structures: C has three types: `if`, `if…else`, and `switch`**
    - **Repetition structures: C has three types: `while`, `do…while` and `for`**

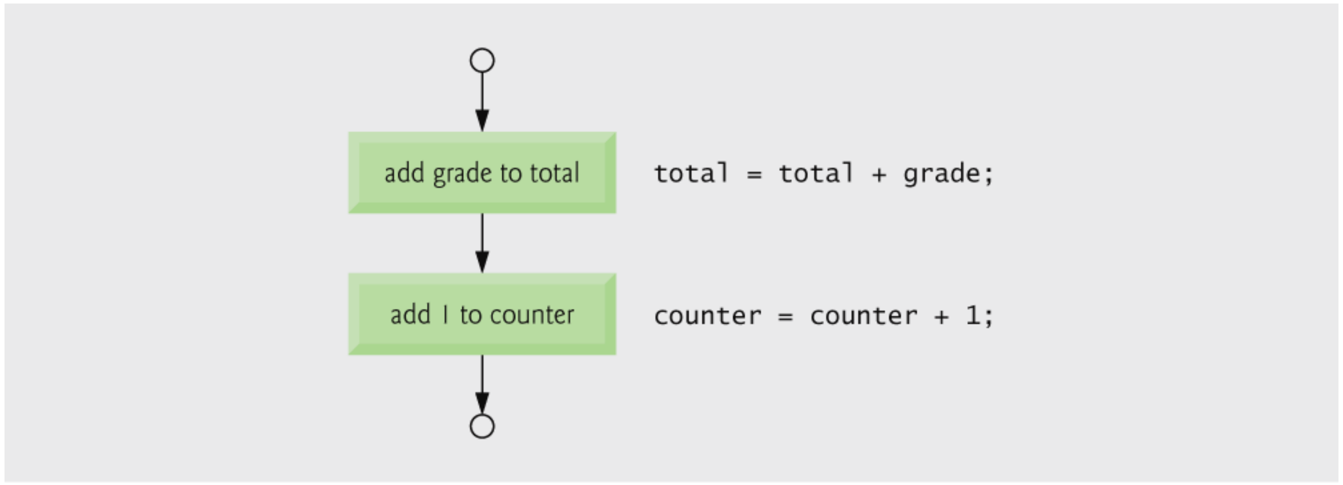

Fig. 3.1 | Flowcharting C’s **sequence** structure.

3.4 Control Structures

- **Flowchart**
  - **Graphical representation of an algorithm**
  - **Drawn using certain special-purpose symbols connected by arrows called flowlines**
  - **Rectangle symbol (action symbol):**
    - **Indicates any type of action**
  - **Oval symbol:**
    - **Indicates the beginning or end of a program or a section of code**
- **Single-entry/single-exit control structures**
  - **Connect exit point of one control structure to entry point of the next (control-structure stacking)**
  - **Makes programs easy to build**

3.5 The `if` selection statement

- Selection structure:
  - Used to choose among alternative courses of action
  - Pseudocode:

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

- If condition `true`
  - Print statement executed and program goes on to next statement
  - If `false`, print statement is ignored and the program goes onto the next statement
  - Indenting makes programs easier to read
    - C ignores whitespace characters

Good Programming Practice 3.1

- Consistently applying responsible indentation conventions greatly improves program readability. We suggest a fixed-size tab of about 1/4 inch or three blanks per indent.
In this book, we use three blanks per indent.

Good Programming Practice 3.2

- Pseudocode is often used to “think out” a program during the program design process. Then the pseudocode program is converted to C.

3.5 The `if` selection statement

- **Pseudocode statement in C:**

```
if ( grade >= 60 )
   printf( "Passed\n" );
```

  - **C code corresponds closely to the pseudocode**
- **Flowchart: Diamond symbol (decision symbol)**
  - **Indicates decision is to be made**
  - **Contains an expression that can be `true` or `false`**
  - **Test the condition, follow appropriate path**

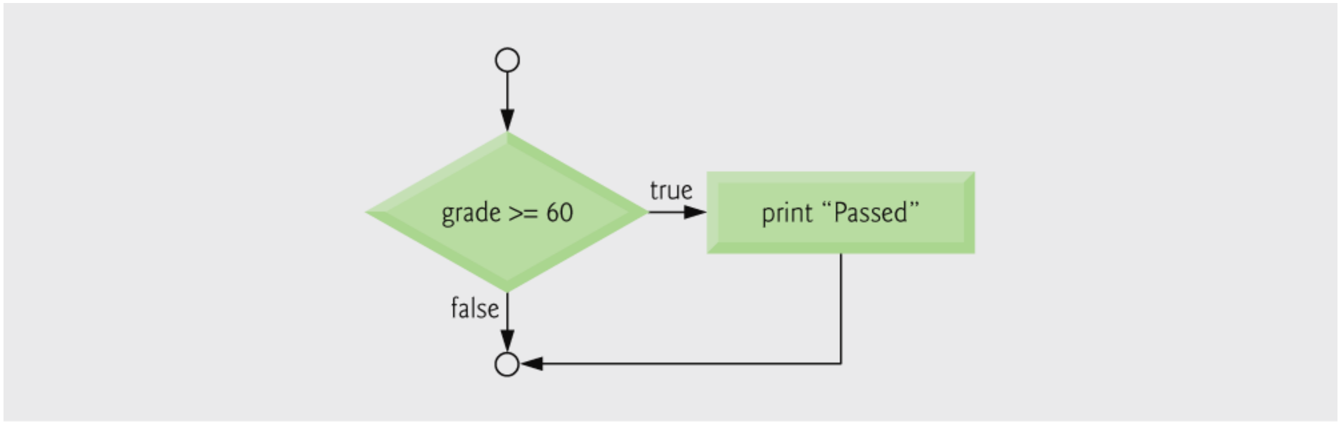

Fig. 3.2: Flowcharting the single-selection `if` statement.

3.6 The `if…else` selection statement

- `if`
  - Only performs an action if the condition is `true`
- `if…else`
  - Specifies an action to be performed both when the condition is `true` and when it is `false`
- **Pseudocode:**

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

  - Note spacing/indentation conventions

Good Programming Practice 3.3

- Indent both body statements of an `if...else` statement.

Good Programming Practice 3.4

- If there are several levels of indentation, each level should be indented the same additional amount of space.

3.6 The if…else selection statement

- C code:

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

- Ternary conditional operator (?:)
  - Takes three arguments (condition, value if `true`, value if `false`)
  - Our C Code could be written:

```
printf( "%s\n", grade >= 60 ? "Passed" : "Failed" );
```

  - Or it could have been written:

```
grade >= 60 ? printf(“Passed\n”) : printf(“Failed\n”);
```

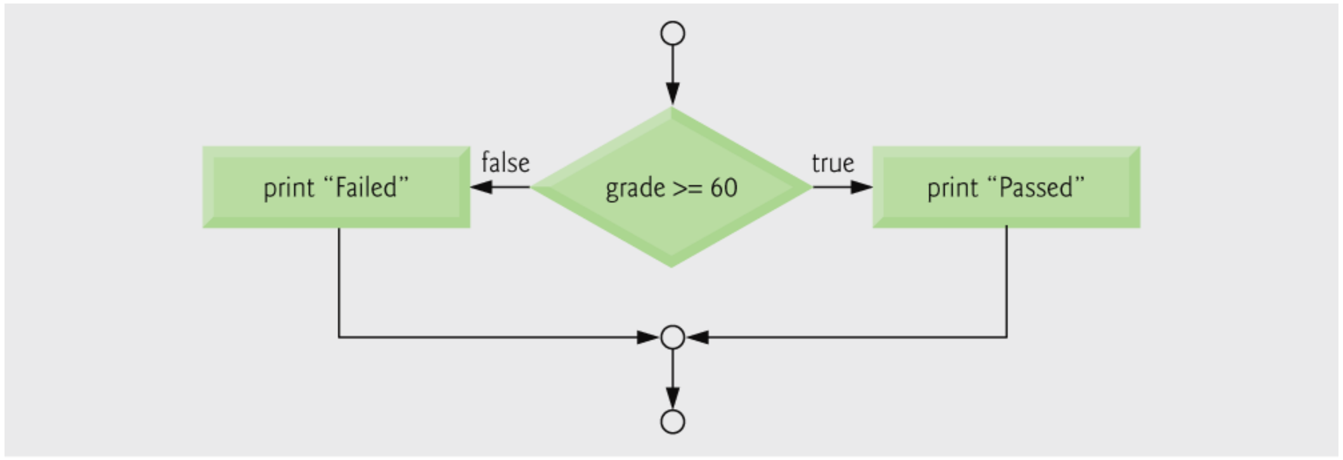

Fig. 3.3: Flowcharting the double-selection `if...else` statement.

3.6 The `if…else` selection statement

- **Nested `if…else` statements**
  - **Test for multiple cases by placing `if…else` selection statements inside `if…else` selection statement**
  - **Once condition is met, rest of statements skipped**
  - **Deep indentation usually not used in practice**

3.6 The `if…else` selection statement

– **Pseudocode for a nested `if…else` statement**

```
If student's grade is greater than or equal to 90
  Print "A"
else
  If student's grade is greater than or equal to 80
      Print "B"
  else
      If student's grade is greater than or equal to 70
          Print "C"
      else
          If student's grade is greater than or equal to 60
              Print "D"
          else
              Print "F"
```

3.6 The `if…else` selection statement

- Compound statement:
  - Set of statements within a pair of braces
  - Example:

```
if ( grade >= 60 )
   printf( "Passed.\n" );
else {
   printf( "Failed.\n" );
   printf( "You must take this course
      again.\n" );
 }
```

  - Without the braces, the statement

```
printf( "You must take this course
   again.\n" );
```

would be executed automatically

Software Engineering Observation 3.1

- A compound statement can be placed anywhere in a program that a single statement can be placed.

Common Programming Error 3.1

- Forgetting one or both of the braces that delimit a compound statement.

3.6 The `if…else` selection statement

- **Block:**
  - Compound statements with declarations
- **Syntax errors**
  - Caught by compiler
- **Logic errors:**
  - Have their effect at execution time
  - Non-fatal: program runs, but has incorrect output
  - Fatal: program exits prematurely

Common Programming Error 3.2

- Placing a semicolon after the condition in an `if` statement as in `if ( grade >= 60 );` leads to a logic error in single-selection `if` statements and a syntax error in double-selection `if` statements.

Error-Prevention Tip 3.1

- Typing the beginning and ending braces of compound statements before typing the individual statements within the braces helps avoid omitting one or both of the braces, preventing syntax errors and logic errors (where both braces are indeed required).

Software Engineering Observation 3.2

- Just as a compound statement can be placed anywhere a single statement can be placed, it is also possible to have no statement at all, i.e., the empty statement. The empty statement is represented by placing a semicolon (;) where a statement would normally be.

3.7 The `while` repetition statement

- Repetition structure
  - Programmer specifies an action to be repeated while some condition remains `true`
  - Pseudocode:

```
While there are more items on my shopping list
    Purchase next item and cross it off my list
```

  - `while` loop repeated until condition becomes `false`
  - Example:

```
int product = 2;
while ( product <= 1000 )
      product = 2 * product;
```

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.”

Common Programming Error 3.4

Spelling the keyword `while` with an uppercase `W` as in While (remember that C is a case-sensitive language). All of C’s reserved keywords such as `while`, `if` and `else` contain only lowercase letters.

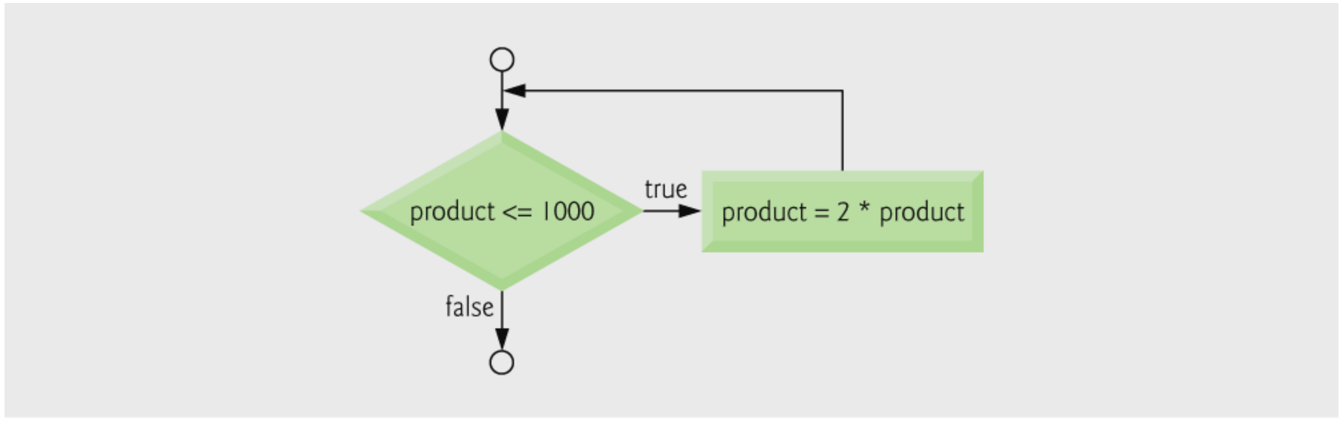

Fig. 3.4 | Flowcharting the `while` repetition statement.

3.8 Counter-Controlled Repetition

- **Counter-controlled repetition**
  - Loop repeated until counter reaches a certain value
  - Definite repetition: number of repetitions is known
  - Example: 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.

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

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

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

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

Outline

fig03_06.c

(1 of 2)

```
/* 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 */
```

Counter to control `while` loop

Initialize `counter` to 1

`while` loop iterates as long as `counter <= 10`

Increment the counter

Outline

```c

   /* 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 */
```

Calculate the average

fig03_06.c

(2 of 2)

```
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
```

Common Programming Error 3.5

If a counter or total is not initialized, the results of your program will probably be incorrect. This is an example of a logic error.

Error-Prevention Tip 3.2

Initialize all counters and totals.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement

- **Problem becomes:**

  Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.

  - **Unknown number of students**
  - **How will the program know to end?**
- **Use sentinel value**
  - **Also called signal value, dummy value, or flag value**
  - **Indicates “end of data entry.”**
  - **Loop ends when user inputs the sentinel value**
  - **Sentinel value chosen so it cannot be confused with a regular input (such as -1 in this case)**

Common Programming Error 3.6

Choosing a sentinel value that is also a legitimate data value.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement

- **Top-down, stepwise refinement**
  - **Begin with a pseudocode representation of the *top*:**

    Determine the class average for the quiz
  - **Divide *top* into smaller tasks and list them in order:**

    Initialize variables
    Input, sum and count the quiz grades
    Calculate and print the class average
- **Many programs have three phases:**
  - **Initialization: initializes the program variables**
  - **Processing: inputs data values and adjusts program variables accordingly**
  - **Termination: calculates and prints the final results**

Software Engineering Observation 3.3

Each refinement, as well as the top itself, is a complete specification of the algorithm; only the level of detail varies.

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement

- **Refine the initialization phase from *Initialize variables* to:**

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

- **Refine *Input, sum and count the quiz grades* to**

```
Input the first grade (possibly the sentinel)
While the user has not as yet entered the sentinel
    Add this grade into the running total
    Add one to the grade counter
    Input the next grade (possibly the sentinel)
```

3.9 Formulating Algorithms with Top-Down, Stepwise Refinement

- **Refine *Calculate and print the class average* to**

```
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”
```

Common Programming Error 3.7

An attempt to divide by zero causes a fatal error.

```
Initialize total to zero
Initialize counter to zero

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)

If the counter is not equal to zero
  Set the average to the total divided by the counter
  Print the average
else
  Print “No grades were entered”
```

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

Good Programming Practice 3.5

When performing division by an expression whose value could be zero, explicitly test for this case and handle it appropriately in your program (such as printing an error message) rather than allowing the fatal error to occur.

Software Engineering Observation 3.4

Many programs can be divided logically into three phases: an initialization phase that initializes the program variables; a processing phase that inputs data values and adjusts program variables accordingly; and a termination phase that calculates and prints the final results.

Software Engineering Observation 3.5

You terminate the top-down, stepwise refinement process when the pseudocode algorithm is specified in sufficient detail for you to be able to convert the pseudocode to C. Implementing the C program is then normally straightforward.

Outline

```
/* 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 */

```

fig03_08.c

(1 of 3)

`float` type indicates variable can be a non-integer

Outline

fig03_08.c

(2 of 3)

```
   /* 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 */
```

while loop repeats until user enters a value of -1

Ensures the user entered at least one grade

Converts **total** to **float** type

Prints result with 2 digits after decimal point

```
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
```

Outline

fig03_08.c

(3 of 3)

Good Programming Practice 3.6

In a sentinel-controlled loop, the prompts requesting data entry should explicitly remind the user what the sentinel value is.

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 are 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.

3.10 Nested Control Structures

- Problem
  - A college has a list of test results (1 = pass, 2 = fail) for 10 students
  - Write a program that analyzes the results
    - If more than 8 students pass, print "Raise Tuition"
- Notice that
  - The program must process 10 test results
    - Counter-controlled loop will be used
  - Two counters can be used
    - One for number of passes, one for number of fails
  - Each test result is a number—either a 1 or a 2
    - If the number is not a 1, we assume that it is a 2

3.10 Nested Control Structures

- **Top level outline**

  Analyze exam results and decide if tuition should be raised

- **First Refinement**

  Initialize variables

  Input the ten quiz grades and count passes and failures

  Print a summary of the exam results and decide if tuition should be raised

- **Refine *Initialize variables* to**

  Initialize passes to zero

  Initialize failures to zero

  Initialize student counter to one

3.10 Nested Control Structures

- Refine *Input the ten quiz grades and count passes and failures* to

```
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
```

- Refine *Print a summary of the exam results and decide if tuition should be raised* to

```
Print the number of passes
Print the number of failures
If more than eight students passed
    Print "Raise tuition"
```

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

While student counter is less than or equal to ten
    Input the next exam result

    If the student passed
      Add one to passes
    else
      Add one to failures

    Add one to student counter

Print the number of passes
Print the number of failures
If more than eight students passed
    Print “Raise tuition”
```

Fig. 3.9 | Pseudocode for examination results problem.

Outline

fig03_10.c

(1 of 3)

```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 */
```

while loop continues until 10 students have been processed

if and **else** statements are nested inside **while** loop

Outline

fig03_10.c

(2 of 3)

```c

   /* 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 "raise tuition" */
   if ( passes > 8 ) {
      printf( "Raise tuition\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
```

(continued on next slide…)

(continued from previous slide…)

```
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
Raise tuition
```

Outline

fig03_10.c

(3 of 3)

Performance Tip 3.1

Initializing variables when they are defined can help reduce a program’s execution time.

Performance Tip 3.2

Many of the performance tips we mention in this text result in nominal improvements, so the reader may be tempted to ignore them. Note that the cumulative effect of all these performance enhancements can make a program perform significantly faster. Also, significant improvement is realized when a supposedly nominal improvement is placed in a loop that may repeat a large number of times.

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.

Software Engineering Observation 3.7

Many programmers write programs without ever using program development tools such as pseudocode. They feel that their ultimate goal is to solve the problem on a computer and that writing pseudocode merely delays the production of final outputs.

3.11 Assignment Operators

- **Assignment operators abbreviate assignment expressions**

  ```
  c = c + 3;
  ```

  can be abbreviated as `c += 3;` using the addition assignment operator

- **Statements of the form**

  ***variable* = *variable operator expression*;**

  can be rewritten as

  ***variable operator*= *expression*;**

- **Examples of other assignment operators:**

  ```
  d -= 4      (d = d - 4)
  e *= 5      (e = e * 5)
  f /= 3      (f = f / 3)
  g %= 9      (g = g % 9)
  ```

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`

Fig. 3.11 | Arithmetic assignment operators.

3.12 Increment and Decrement Operators

- Increment operator (`++`)
  - Can be used instead of `c+=1`
- Decrement operator (`--`)
  - Can be used instead of `c-=1`
- Preincrement
  - Operator is used before the variable (`++c` or `--c`)
  - Variable is changed before the expression it is in is evaluated
- Postincrement
  - Operator is used after the variable (`c++` or `c--`)
  - Expression executes before the variable is changed

3.12 Increment and Decrement Operators

- If `c` equals 5, then

  ```
  printf( "%d", ++c );
  ```

  - Prints 6

  ```
  printf( "%d", c++ );
  ```

  - Prints 5
  - In either case, `c` now has the value of 6
- When variable not in an expression
  - Preincrementing and postincrementing have the same effect

  ```
  ++c;
  printf( “%d”, c );
  ```

  - Has the same effect as

  ```
  c++;
  printf( “%d”, c );
  ```

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.

Fig. 3.12 | Increment and decrement operators.

Outline

fig03_13.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 */
```

```
5
5
6

5
6
6
```

c is printed, then incremented

c is incremented, then printed

Good Programming Practice 3.7

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

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 will 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.

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 of the operators encountered so far in the text.