Chapter 3
Structured Program Development in C

C How to Program, 8/e

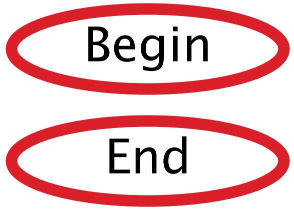
3.2 Algorithms

- An algorithm is a procedure for solving a problem in terms of
  - the actions to be executed, and
  - the order in which these actions are to be executed

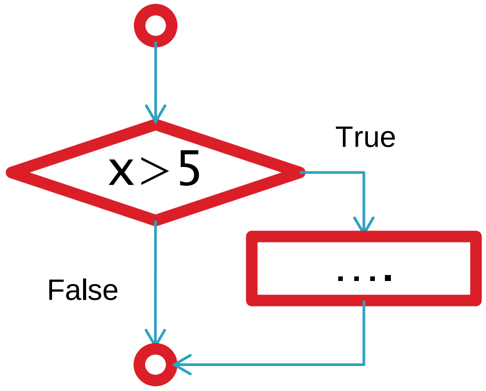
- Pseudocode is an artificial and informal language that helps you develop algorithms.

3.4 Control Structures (Cont.)

- Like pseudocode, flowcharts are useful for developing and representing algorithms, although pseudocode is preferred by most programmers.
- oval symbol:



- connector symbol:
- diamond symbol, also called the decision symbol: a decision is to be made.



3.4 Control Structures (Cont.)

- The `if` statement is called a single-selection statement because it selects or ignores a single action.
- The `if…else` statement is called a double-selection statement because it selects between two different actions.
- The `switch` statement is called a multiple-selection statement because it selects among many different actions.

Repetition Statements in C

- C provides three types of repetition structures in the form of statements, namely `while` (Section 3.7), `do…while`, and `for` (both discussed in Chapter 4).
- That's all there is.

3.4 Control Structures (Cont.)

- C has only seven control statements: sequence, three types of selection and three types of repetition.
- Each C program is formed by combining as many of each type of control statement as is appropriate for the algorithm the program implements.
- These single-entry/single-exit control statements make it easy to build clear programs.

3.5 The `if` Selection Statement (Cont.)

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

  - ```
    if ( grade >= 60 ) {
       printf( "Passed\n" );
    } /* end if */
    ```

- Notice that the C code corresponds closely to the pseudocode
    ```

3.5 The `if` Selection Statement (Cont.)

- if the expression evaluates to *zero*, it's treated as false
- if it evaluates to *nonzero*, it's treated as true.

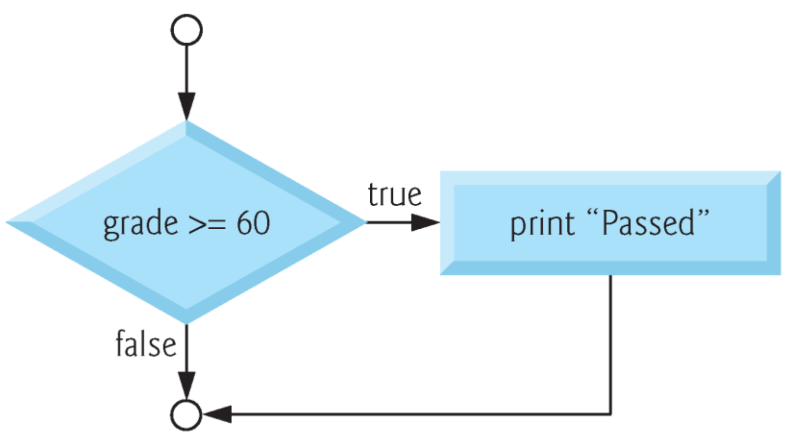


Fig. 3.2 | Flowcharting the single-selection `if` statement.

3.6 The `if...else` Selection Statement

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

- The preceding pseudocode *If…else* statement may be written in C as
  - ```
    if ( grade >= 60 ) {
       printf( "Passed\n" );
    } /* end if */
    else {
       printf( "Failed\n" );
    } /* end else */
    ```
    ```

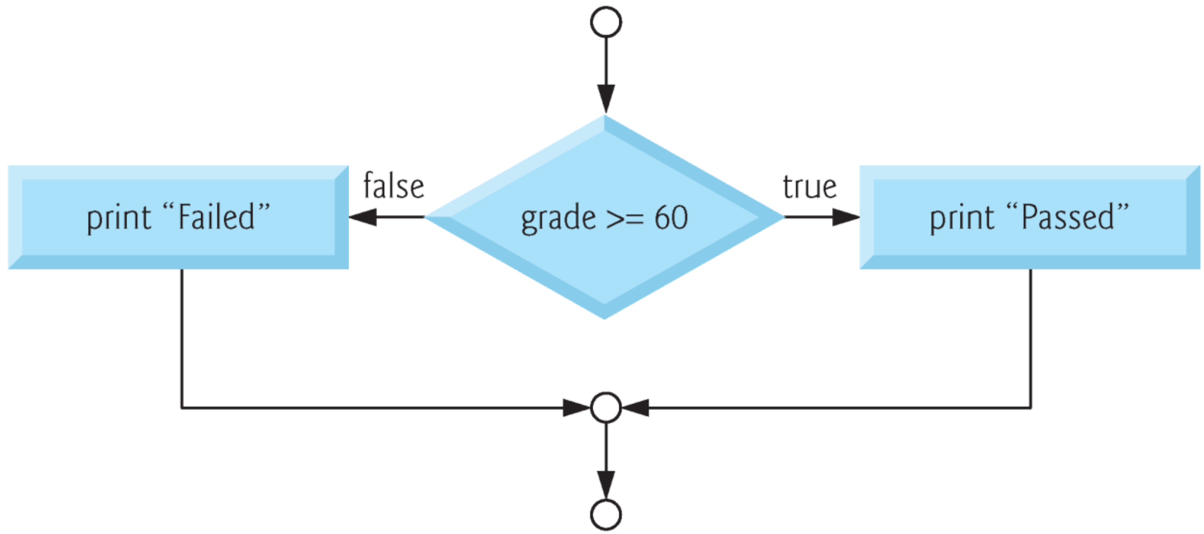


Fig. 3.3 | Flowcharting the double-selection `if...else` statement.

3.6 The `if...else` Selection Statement (Cont.)

- C provides the conditional operator (`?:`) which is closely related to the `if`…`else` statement.

- level = grade >=60 ? 2 : 1;

- The statement is equivalent to

```
 If (grade >= 60)
    level = 2;
 else
    level = 1;
```

3.6 The `if...else` Selection Statement (Cont.)

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

- if the condition `grade >= 60` is true, print the string `"Passed“`, and if the condition is false, print the string `"Failed"`
- The format control string of the `printf` contains the conversion specification **`%s`** for printing a character string.
- An alternative statement

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

3.6 The `if...else` Selection Statement (Cont.)

Nested if...else Statements

```
if (x > 0)
      num_pos = num_pos + 1;
else

      if (x < 0)
            num_neg = num_neg + 1;
      else /* x equals 0 */
            num_zero = num_zero + 1;
```

3.6 The `if...else` Selection Statement (Cont.)

```
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 (Cont.)

- This pseudocode may be written in C as
  - ```
    if ( grade >= 90 )
       puts( "A" );
    else
       if ( grade >= 80 )
          puts("B");
       else
          if ( grade >= 70 )
             puts("C");
          else
             if ( grade >= 60 )
                puts( "D" );
             else
                puts( "F" );
    ```
    ```

3.6 The `if...else` Selection Statement (Cont.)

- You may prefer to write the preceding `if` statement as
  - ```
    if ( grade >= 90 )
       puts( "A" );
    else if ( grade >= 80 )
       puts( "B" );
    else if ( grade >= 70 )
       puts( "C" );
    else if ( grade >= 60 )
       puts( "D" );
    else
       puts( "F" );
    ```
    ```

3.6 The `if...else` Selection Statement (Cont.)

- Compound statement: to include several statements in the body of an `if`, enclose the set of statements in braces (`{` and `}`).

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

3.7 The while Repetition Statement

- A repetition statement (also called an iteration statement) allows you to specify that an action is to be repeated while some condition remains true.

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

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

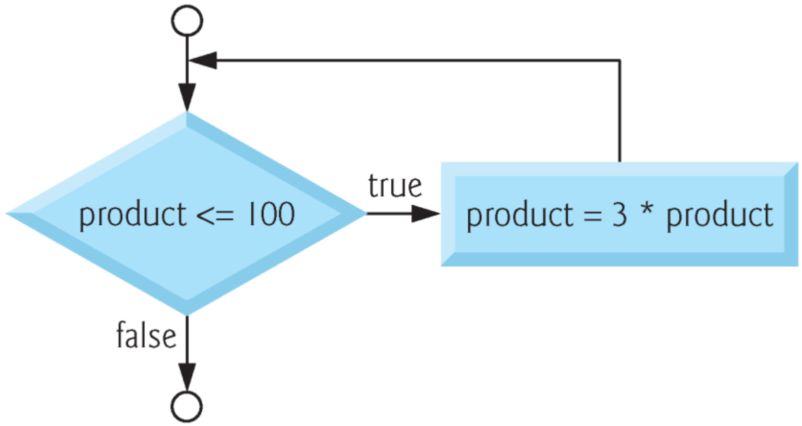


Fig. 3.4 | Flowcharting the `while` iteration statement.

3.8 Formulating Algorithms Case Study 1: Counter-Controlled Repetition

- 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.*

```
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.

We use counter-controlled repetition to input the grades one at a time.

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

// function main begins program execution
int main( void )
{
   unsigned int counter; // number of grade to be entered next
   int grade; // grade value
   int total; // sum of grades entered 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( "%s", "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

```

Initialize variables before using them

Fig. 3.6 | Class-average problem with counter-controlled iteration. (Part 1 of 2.)

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

   printf( "Class average is %d\n", average ); // display result
} // end function main
```

```
Enter grade: 98
Enter grade: 76
Enter grade: 71
Enter grade: 87
Enter grade: 83
Enter grade: 90
Enter grade: 57
Enter grade: 79
Enter grade: 82
Enter grade: 94
Class average is 81
```

The average should be 81.7
(**Type of variables**)

Fig. 3.6 | Class-average problem with counter-controlled iteration. (Part 2 of 2.)

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

- Consider the following problem:
  - *Develop a class-averaging program that will process an arbitrary number of grades each time the program is run.*
- How can the program determine when to stop the input of grades? How will it know when to calculate and print the class average?
  - sentinel value

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

- **Top-down** design
- The pseudocode statement
  - Input, sum, and count the quiz grades
  - Calculate and print the class average

- The refinement of "Input, sum, and count the quiz grades" 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

- Refinement:
  - 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

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

// function main begins program execution
int main( void )
{
   unsigned 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( "%s", "Enter grade, -1 to end: " ); // prompt for input
   scanf( "%d", &grade ); // read grade from user

```

type `float` to handle floating-point numbers

Fig. 3.8 | Class-average program with sentinel-controlled iteration. (Part 1 of 3.)

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

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

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

        // calculate average of all grades entered
        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
        puts( "No grades were entered" );
    } // end else
} // end function main
```

Cast

小數點下兩位

Fig. 3.8 | Class-average program with sentinel-controlled iteration. (Part 2 of 3.)

```
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 | Class-average program with sentinel-controlled iteration. (Part 3 of 3.)

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

- Line 38
  - `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.

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

Formatting Floating-Point Numbers

- Figure 3.8 uses the `printf` conversion specifier `%.2f` (line 41) 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.

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 the stored floating-point value can be only an *approximation*.

3.11 Assignment Operators

- The statement
  - `c = c + 3;`
- can be abbreviated with the addition assignment operator += as
  - `c += 3;`

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, ++, and decrement operator, --
  - x++ or x- -
  - x+=1;
  - x=x+1;
- Preincrement: ++ is in front of its operand
  - The expression's value is the variable's value after incrementing
  - Ex: y = ++x;
- Postincrement: ++ comes after the operand
  - The expression's value is the variable's value before incrementing
  - Ex: y = x++;

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

```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
} // end function main
```

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

```
5
5
6

5
6
6
```

Fig. 3.13 | Preincrementing and postincrementing. (Part 2 of 2.)

3.12 Increment and Decrement Operators (Cont.)

- The three assignment statements in Fig. 3.10
  - ```
    passes = passes + 1;
    failures = failures + 1;
    student = student + 1;
    ```

  can be written more concisely with *assignment operators* as
  - ```
    passes += 1;
    failures += 1;
    student += 1;
    ```

  with *preincrement operators* as
  - ```
    ++passes;
    ++failures;
    ++student;
    ```

  or with *postincrement operators* as
  - ```
    passes++;
    failures++;
    student++;
    ```

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.

3.13 Secure C Programming

Arithmetic Overflow

- Figure 2.5 presented an addition program which calculated the sum of two int values (line 18) with the statement

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

- Even this simple statement has a potential problem—adding the integers could result in a value that’s *too large* to store in an int variable.
- This is known as arithmetic overflow and can cause undefined behavior, possibly leaving a system open to attack.

3.13 Secure C Programming (Cont.)

- The maximum and minimum values that can be stored in an `int` variable are represented by the constants `INT_MAX` and `INT_MIN`, respectively, which are defined in the header `<limits.h>`.
- You can see your platform's values for these constants by opening the header `<limits.h>` in a text editor.
- It's considered a good practice to ensure that before you perform arithmetic calculations like the one in line 18 of Fig. 2.5, they will not overflow.
- The code for doing this is shown on the CERT website `www.securecoding.cert.org`—just search for guideline "INT32-C."
- The code uses the && (logical AND) and `||` (logical OR) operators, which are introduced in the Chapter 4.

3.13 Secure C Programming (Cont.)

Unsigned Integers

- In Fig. 3.6, line 8 declared as an `unsigned int` the variable counter because it's used to count only *non-negative values*.
- In general, counters that should store only non-negative values should be declared with `unsigned` before the integer type.
- Variables of `unsigned` types can represent values from 0 to approximately twice the positive range of the corresponding signed integer types.
- You can determine your platform's maximum unsigned `int` value with the constant `UINT_MAX` from `<limits.h>`.

3.13 Secure C Programming (Cont.)

- The class-averaging program in Fig. 3.6 could have declared as `unsigned int` the variables `grade`, `total` and `average`.
- Grades are normally values from 0 to 100, so the total and average should each be greater than or equal to 0.
- We declared those variables as `int`s because we can't control what the user actually enters—the user could enter negative values.
- Worse yet, the user could enter a value that's not even a number. (We'll show how to deal with such inputs later in the book.)

3.13 Secure C Programming (Cont.)

- Sometimes sentinel-controlled loops use invalid values to terminate a loop.
- For example, the class-averaging program of Fig. 3.8 terminates the loop when the user enters the sentinel `-1` (an invalid grade), so it would be improper to declare variable grade as an `unsigned int`.
- As you'll see, the end-of-file (`EOF`) indicator—which is introduced in the next chapter and is often used to terminate sentinel-controlled loops—is also a negative number.

3.13 Secure C Programming (Cont.)

scanf_s and printf_s

- The C11 standard's Annex K introduces more secure versions of `printf` and `scanf` called `printf_s` and `scanf_s`. Annex K is designated as optional, so not every C vendor will implement it.
- Microsoft implemented its own versions of `printf_s` and `scanf_s` prior to the publication of the C11 standard and immediately began issuing warnings for every `scanf` call.
- The warnings say that `scanf` is deprecated—it should no longer be used—and that you should consider using `scanf_s` instead.

3.13 Secure C Programming (Cont.)

- There are two ways to eliminate Visual C++'s `scanf` warnings—you can use `scanf_s` instead of `scanf` or you can disable these warnings.
- For the input statements we've used so far, Visual C++ users can simply replace `scanf` with `scanf_s`. You can disable the warning messages in Visual C++ as follows:

1. Type *Alt F7* to display the Property Pages dialog for your project.
2. In the left column, expand Configuration Properties > C/C++ and select Preprocessor.
3. In the right column, at the end of the value for Preprocessor Definitions, insert

   `;_CRT_SECURE_NO_WARNINGS`
4. Click OK to save the changes.