Chapter 4
C Program Control

C How to Program, 8/e

4.1 Introduction

- repetition control statement
  - `For`
  - `do`…`while`,
- multiple-selection statement
  - `switch`
- exiting from certain control statement
  - `break`
- skipping the remainder of the body of a repetition statement and proceeding with the next iteration of the loop
  - `continue`

4.2 Repetition Essentials

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

- We've discussed two means of repetition:
  - Counter-controlled repetition
  - Sentinel-controlled repetition

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 iteration.
#include <stdio.h>

int main(void)
{
   unsigned int counter = 1; // initialization

   while (counter <= 10) { // iteration condition
      printf ("%u\n", counter);
      ++counter; // increment
   }
}
```

宣告變數可順便初始化

counter加1

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

Fig. 4.1 | Counter-controlled iteration.

4.3 Counter-Controlled Repetition (Cont.)

- You could make the program in Fig. 4.1 more concise by initializing `counter` to `0` and by replacing the `while` statement with

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

- 難懂、易出錯
  - 少用

4.4 for Repetition Statement

- `for` repetition: counter-controlled repetition.

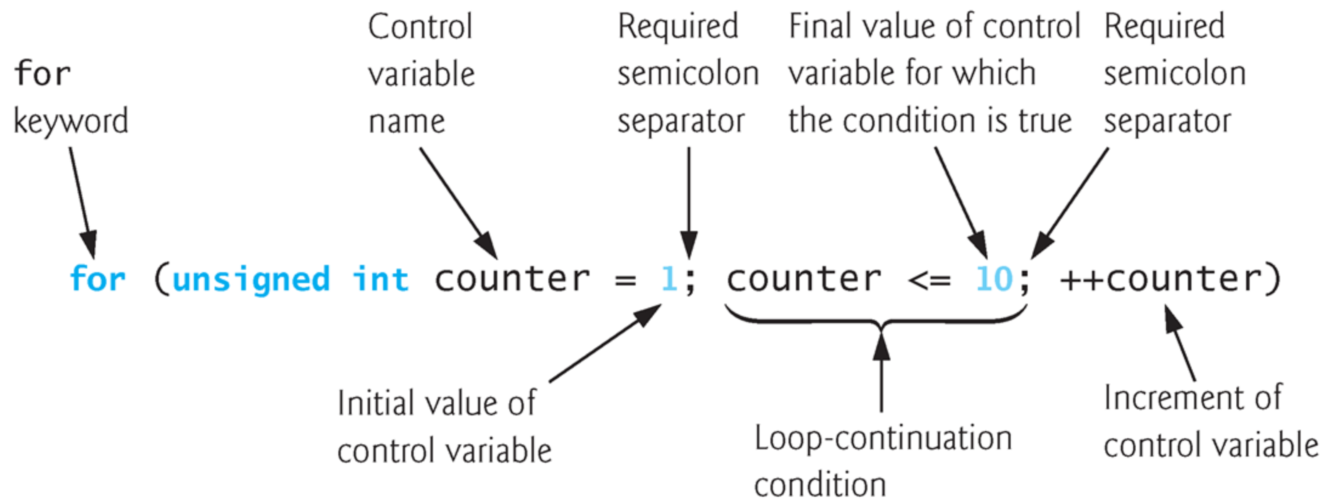

Fig. 4.3 | `for` statement header components.

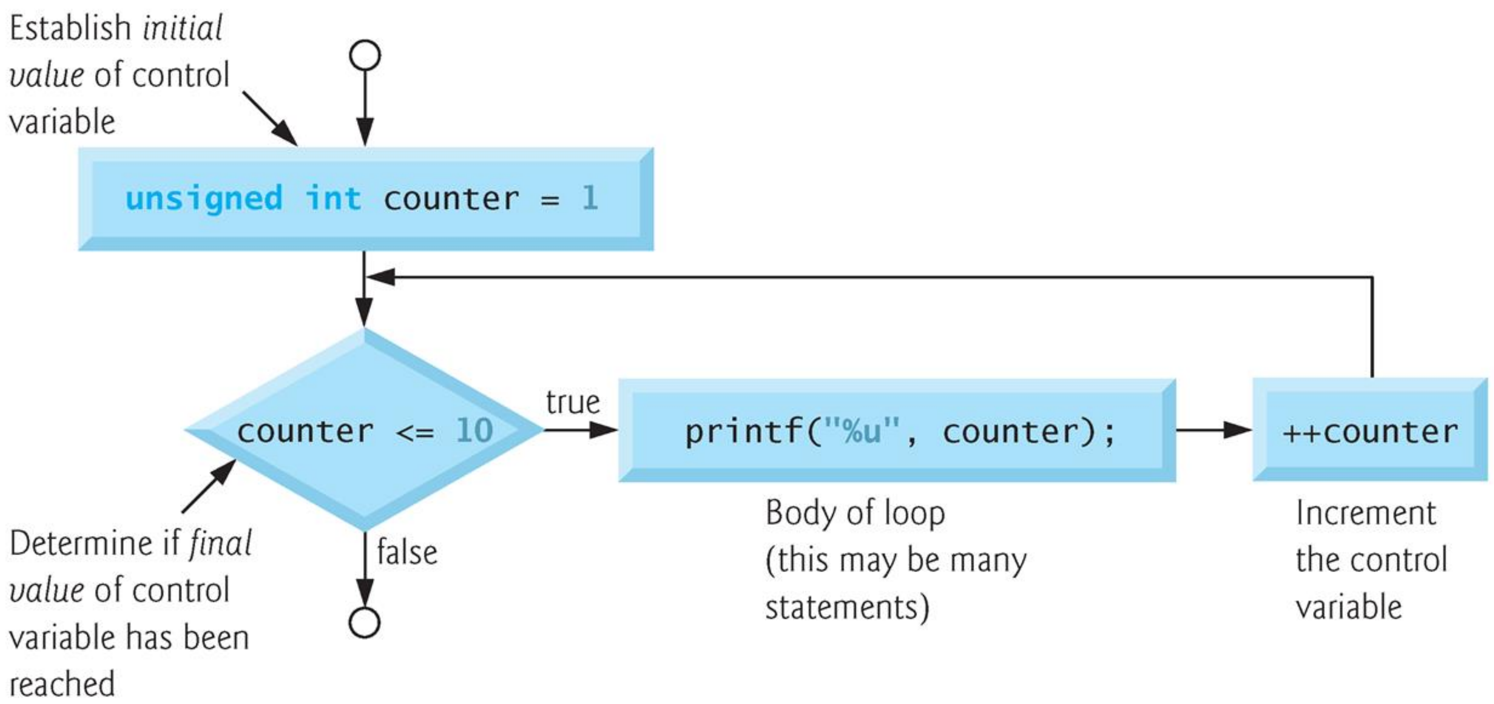

Fig. 4.4 | Flowcharting a typical `for` iteration statement.

```c
// Fig. 4.2: fig04_02.c
// Counter-controlled iteration with the for statement.
#include <stdio.h>

int main(void)
{
   // initialization, iteration condition, and increment
   //  are all included in the for statement header.
   for (unsigned int counter = 1; counter <= 10; ++counter) {
      printf("%u\n", counter);
   }
}
```

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

4.4 for Repetition Statement (Cont.)

Off-By-One Errors

- Notice that Fig. 4.2 uses the loop-continuation condition `counter <= 10`.
- If you incorrectly wrote `counter < 10`, then the loop would be executed only 9 times.
- This is a common logic error called an off-by-one error.

4.4 for Repetition Statement (Cont.)

General Format of a `for` Statement

- 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.)

Expressions in the `for` Statement's Header Are Optional

- 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.
- You 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.

4.4 for Repetition Statement (Cont.)

- ```
  counter = counter + 1
  counter += 1
  ++counter
  counter++
  ```
  are all equivalent in the increment part of the `for` statement.

- 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 equivalent to the statement
  ```
  for ( j = 2; j <= 80; j += 5 )
  ```
- The "increment" may be negative.
  ```

4.6 Examples Using the `for` Statement

- The following examples show methods of varying the control variable in a `for` statement.
  - Vary the control variable from `1` to `100` in increments of `1`.

    `for ( i = 1; i <= 100; ++ i )`
  - Vary the control variable from `100` to `1` in increments of `-1` (decrements of `1`).

    `for ( i = 100; i >= 1; --i )`
  - Vary the control variable from `7` to `77` in steps of `7`.

    `for ( i = 7; i <= 77; i += 7 )`
  - Vary the control variable from `20` to `2` in steps of `-2`.

    `for ( i = 20; i >= 2; i -= 2 )`
  - Vary the control variable over the following sequence of values: `2`, `5`, `8`, `11`, `14`, `17`.

    `for ( j = 2; j <= 17; j += 3 )`
  - Vary the control variable over the following sequence of values: `44`, `33`, `22`, `11`, `0`.

    `for ( j = 44; j >= 0; j -= 11 )`

4.6 Examples Using the for Statement (Cont.)

- 請用for loop計算小於100的所有奇數之和

4.6 Examples Using the `for` Statement (Cont.)

Application: Compound-Interest Calculations

- A person invests $1000.00 in a savings account yielding 5% interest. Assuming that all interest is left on deposit in the account, calculate and print the amount of money in the account at the end of each year for 10 years. Use the following formula for determining these amounts:

  $a = p(1 + r)^n$

  where

  p is the original amount invested (i.e., the principal)
  r is the annual interest rate
  n is the number of years
  a is the amount on deposit at the end of the n^{th} year.

```
Year     Amount on deposit
   1               1050.00
   2               1102.50
   3               1157.63
   4               1215.51
   5               1276.28
   6               1340.10
   7               1407.10
   8               1477.46
   9               1551.33
  10               1628.89
```

```c
// Fig. 4.6: fig04_06.c
// Calculating compound interest.
#include <stdio.h>
#include <math.h>

int main(void)
{
   double principal = 1000.0; // starting principal
   double rate = .05; // annual interest rate

   // output table column heads
   printf("%4s%21s\n", "Year", "Amount on deposit");

   // calculate amount on deposit for each of ten years
   for (unsigned int year = 1; year <= 10; ++year) {

      // calculate new amount for specified year
      double amount = principal * pow(1.0 + rate, year);

      // output one table row
      printf("%4u%21.2f\n", year, amount);
   }
}
```

Fig. 4.6 | Calculating compound interest. (Part 1 of 2.)

```
Year    Amount on deposit
   1              1050.00
   2              1102.50
   3              1157.63
   4              1215.51
   5              1276.28
   6              1340.10
   7              1407.10
   8              1477.46
   9              1551.33
  10              1628.89
```

Fig. 4.6 | Calculating compound interest. (Part 2 of 2.)

4.6 Examples Using the for Statement (Cont.)

- The header `<math.h>` (line 4) should be included whenever a math function such as `pow` is used.

- Function `pow` requires two `double` arguments, but variable `year` is an integer.

4.6 Examples Using the `for` Statement (Cont.)

Formatting Numeric Output

- The conversion specifier `%21.2f`
  - The `21` denotes the *field width*
  - The `2` specifies the *precision* (i.e., the number of decimal positions).

`%-6d`

- To *left justify* a value in a field, place a `-` (minus sign) between the `%` and the field width.
  - `%-6d`

Exercise

- 寫一程式製作一攝氏與華氏溫度對照表

Celsius	Fahrenheit
10	50.00
5	41.00
0	32.00
-5	23.00

列印到小數點下兩位

- F = 1.8*C + 32

Nested Loop

```
for ( i=0;  i< 5;  i++){
        for(j = 0;  j<5;  j++) {
                printf("*");
        }
        printf("\n");
}
```

Exercise

- Write nests of loops that cause the following output to be displayed:

0

0 1

0 1 2

0 1 2 3

0 1 2 3 4

4.7 switch Multiple-Selection Statement

```
int  score;
scanf("%d", &score);
switch( score) {
case  1:
            printf("score = 1\n");
            break;
case  2:
            printf("score = 2\n");
            break;
default:
            printf("other cases\n");
}
```

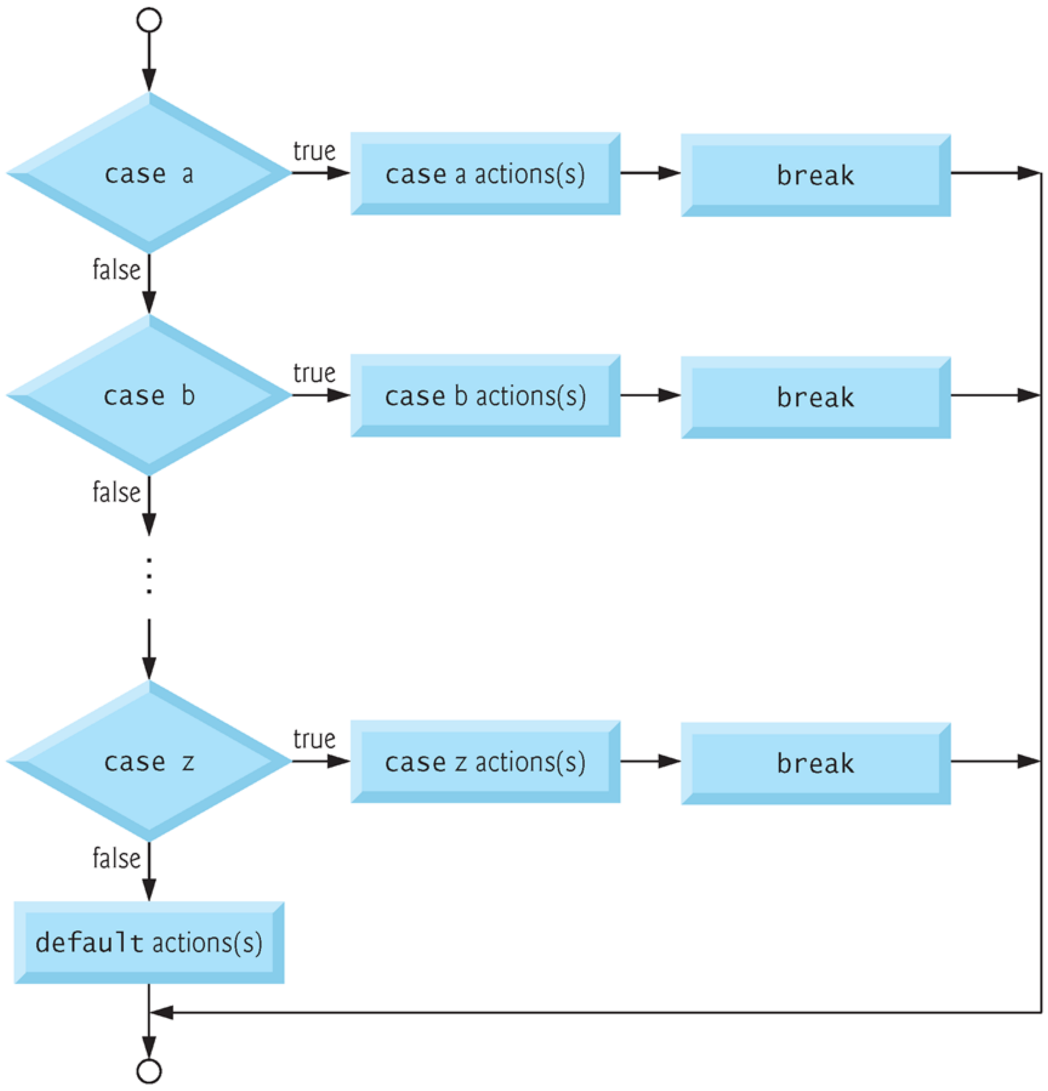

Fig. 4.8 | `switch` multiple-selection statement with `break`s.

計算成績A~F個別有幾人

```c
// Fig. 4.7: fig04_07.c
// Counting letter grades with switch.
#include <stdio.h>

int main(void)
{
   unsigned int aCount = 0;
   unsigned int bCount = 0;
   unsigned int cCount = 0;
   unsigned int dCount = 0;
   unsigned int fCount = 0;

   puts("Enter the letter grades.");
   puts("Enter the EOF character to end input.");
   int grade; // one grade

```

Fig. 4.7 | Counting letter grades with `switch`. (Part 1 of 5.)

```c
    // loop until user types end-of-file key sequence
    while ((grade = getchar()) != EOF) {

        // determine which grade was input
        switch (grade) { // switch nested in while

            case 'A': // grade was uppercase A
            case 'a': // or lowercase a
                ++aCount;
                break; // necessary to exit switch

            case 'B': // grade was uppercase B
            case 'b': // or lowercase b
                ++bCount;
                break;

            case 'C': // grade was uppercase C
            case 'c': // or lowercase c
                ++cCount;
                break;

```

getchar() reads a character from keyboard

Fig. 4.7 | Counting letter grades with `switch`. (Part 2 of 5.)

```c
         case 'D': // grade was uppercase D
         case 'd': // or lowercase d
            ++dCount;
            break;

         case 'F': // grade was uppercase F
         case 'f': // or lowercase f
            ++fCount;
            break;

         case '\n': // ignore newlines,
         case '\t': // tabs,
         case ' ': // and spaces in input
            break;

         default: // catch all other characters
            printf("%s", "Incorrect letter grade entered.");
            puts(" Enter a new grade.");
            break; // optional; will exit switch anyway
      }
   } // end while

```

Fig. 4.7 | Counting letter grades with `switch`. (Part 3 of 5.)

```
    // output summary of results
    puts("\nTotals for each letter grade are:");
    printf("A: %u\n", aCount);
    printf("B: %u\n", bCount);
    printf("C: %u\n", cCount);
    printf("D: %u\n", dCount);
    printf("F: %u\n", fCount);
}
```

Fig. 4.7 | Counting letter grades with `switch`. (Part 4 of 5.)

```
Enter the letter grades.
Enter the EOF character to end input.
a
b
c
C
A
d
f
C
E
Incorrect letter grade entered. Enter a new grade.
D
A
b
^Z
```

Not all systems display a representation of the EOF character

```
Totals for each letter grade are:
A: 3
B: 2
C: 3
D: 2
F: 1
```

Windows 用 Ctrl-Z
Linux 用 Ctrl-D

Fig. 4.7 | Counting letter grades with `switch`. (Part 5 of 5.)

4.7 `switch` Multiple-Selection Statement (Cont.)

- We can treat a character as either an integer or a character, depending on its use.
- For example, the statement

```
printf( "The character (%c) has the value %d.\n", 'a', 'a' );
```

- The result is

```
The character (a) has the value 97.
```

- ASCII (American Standard Code for Information Interchange) character set
  - 97 is the ASCII code for ‘a’

4.7 `switch` Multiple-Selection Statement (Cont.)

`switch` Statement Details

- switch(grade)
  - This is called the controlling expression.

- The value of this expression is compared with each of the `case` labels.
  - switch(grade)
  - case ‘a’:
  - case ‘A’:
  - …
  - break:
  - case ‘b’:

4.7 `switch` Multiple-Selection Statement (Cont.)

- The `break` statement causes program control to continue with the first statement after the `switch` statement.
- If `break` is not used anywhere in a `switch` statement, then each time a match occurs in the statement, the statements for all the remaining `case`s will be executed.
- If no match occurs, the `default` case is executed, and an error message is printed.

4.7 `switch` Multiple-Selection Statement (Cont.)

Ignoring Newline, Tab and Blank Characters in Input

- In the `switch` statement of Fig. 4.7, the lines

```
case '\n': // ignore newlines,
case '\t': // tabs,
case ' ': // and spaces in input
   break; // exit switch
```

cause the program to skip newline, tab and blank characters.

4.7 `switch` Multiple-Selection Statement (Cont.)

- Reading characters one at a time can cause some problems.
  - the newline character (Enter key)
- Often, this newline character must be specially processed to make the program work correctly.
- By including the preceding cases in our `switch` statement, we prevent the error message in the `default` case from being printed each time a newline, tab or space is encountered in the input.

4.7 `switch` Multiple-Selection Statement (Cont.)

Notes on Integral Types

- C provides several data types to represent integers.
  - `int`
  - `char`
  - `short int` (which can be abbreviated as `short`)
  - `long int` (which can be abbreviated as `long`).

- For `short int`s the minimum range is –32767 to +32767.
- The minimum range of values for `long int`s is –2147483647 to +2147483647.
- For most integer calculations, `long int`s are sufficient.

4.8 do…while Repetition Statement

- The `do…while` repetition statement is similar to the `while` statement.

```
do {
   statement
} while ( condition );
```

- The `do…while` statement tests the loop-continuation condition *after* the loop body is performed.

- Therefore, the loop body will be executed at least once.

4.8 do…while Repetition Statement (Cont.)

- Figure 4.9 uses a `do…while` statement to print the numbers from 1 to 10.
- The control variable `counter` is preincremented in the loop-continuation test.

```c
// Fig. 4.9: fig04_09.c
// Using the do...while iteration statement.
#include <stdio.h>

int main(void)
{
   unsigned int counter = 1; // initialize counter

   do {
      printf("%u  ", counter);
   } while (++counter <= 10);
}
```

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

Fig. 4.9 | Using the do…while iteration statement.

4.8 do...while Repetition Statement (Cont.)

***do...while* Statement Flowchart**

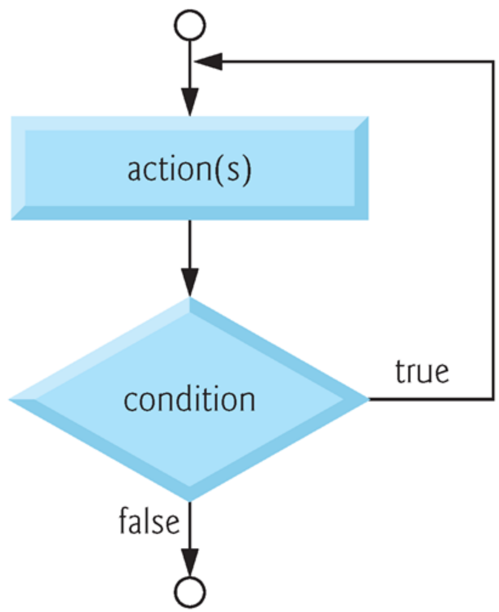

Fig. 4.10 | Flowcharting the `do...while` iteration statement.

4.9 break and continue Statements

- The `break` and `continue` statements are used to alter the flow of control.

***break* Statement**

- The `break` statement causes an immediate exit from that statement, when executed in a `while`, `for`, `do`…`while` or `switch` statement,

```c
// Fig. 4.11: fig04_11.c
// Using the break statement in a for statement.
#include <stdio.h>

int main(void)
{
   unsigned int x; // declared here so it can be used after loop

   // loop 10 times
   for (x = 1; x <= 10; ++x) {

      // if x is 5, terminate loop
      if (x == 5) {
         break; // break loop only if x is 5
      }

      printf("%u ", x);
   }

   printf("\nBroke out of loop at x == %u\n", x);
}
```

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.11 | Using the `break` statement in a `for` statement.

4.9 break and continue Statements (Cont.)

***continue* Statement**

- The `continue` statement skips the remaining statements in the body of that control statement and performs the next iteration of the loop.

```c
// Fig. 4.11: fig04_11.c
// Using the break statement in a for statement.
#include <stdio.h>

int main(void)
{
   unsigned int x; // declared here so it can be used after loop

   // loop 10 times
   for (x = 1; x <= 10; ++x) {

      // if x is 5, terminate loop
      if (x == 5) {
         break; // break loop only if x is 5
      }

      printf("%u ", x);
   }

   printf("\nBroke out of loop at x == %u\n", x);
}
```

```
1 2 3 4
Broke out of loop at x == 5
```

Fig. 4.11 | Using the `break` statement in a `for` statement.

4.10 Logical Operators

- C provides *logical operators* that may be used to form more complex conditions by combining simple conditions.
- && (logical AND)
- || (logical OR)
- ! (logical NOT also called logical negation).

4.10 Logical Operators (Cont.)

Logical AND (&&) Operator

- Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution.
  - Ex:

```
if ( gender == 1 && age >= 65 )
   ++seniorFemales;
```

4.10 Logical Operators (Cont.)

- truth table
- &&
- (logical AND)

expression1	expression2	expression1 && expression2
0	0	0
0	nonzero	0
nonzero	0	0
nonzero	nonzero	1

Fig. 4.13 | Truth table for the logical AND (&&) operator.

- *C evaluates all expressions that include relational operators, equality operators, and/or logical operators to 0 or 1.*
- Although C sets a true value to 1, it accepts any nonzero value as true.

4.10 Logical Operators (Cont.)

- || (logical OR) operator.

```
if ( semesterAverage >= 90 || finalExam >= 90 )
   printf( "Student grade is A\n" );:
```

expression1	expression2	expression1 || expression2
0	0	0
0	nonzero	1
nonzero	0	1
nonzero	nonzero	1

Fig. 4.14 | Truth table for the logical OR (||) operator.

4.10 Logical Operators (Cont.)

- short-circuit evaluation
  - evaluation of the condition
    `gender == 1 && age >= 65`
  - will stop if `gender` is not equal to `1` (i.e., the entire expression is false), and continue if `gender` is equal to `1` (i.e., the entire expression could still be true if `age >= 65`).

4.10 Logical Operators (Cont.)

- C provides ! (logical negation) to enable a programmer to “reverse” the meaning of a condition.

```
if ( !( grade == sentinelValue ) )
   printf( "The next grade is %f\n", grade );
```

  - The parentheses around the condition `grade == sentinelValue` are needed because the logical negation operator has a higher precedence than the equality operator.

expression	!expression
0	1
nonzero	0

Fig. 4.15 | Truth table for operator ! (logical negation).

4.10 Logical Operators (Cont.)

- In most cases, you can avoid using logical negation by expressing the condition differently with an appropriate relational operator.

- For example, the preceding statement may also be written as follows:

```
if ( grade != sentinelValue )
   printf( "The next grade is %f\n", grade );
```

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

Fig. 4.16 | Operator precedence and associativity.

4.11 Confusing Equality (==) and Assignment (=) Operators (Cont.)

- Suppose we intend to write

```
if ( payCode == 4 )
   printf( “%s“, "You get a bonus!" );
```

  but we accidentally write

```
if ( payCode = 4 )
   printf( “%s“, "You get a bonus!" );
```

- Result?

4.12 Structured Programming Summary

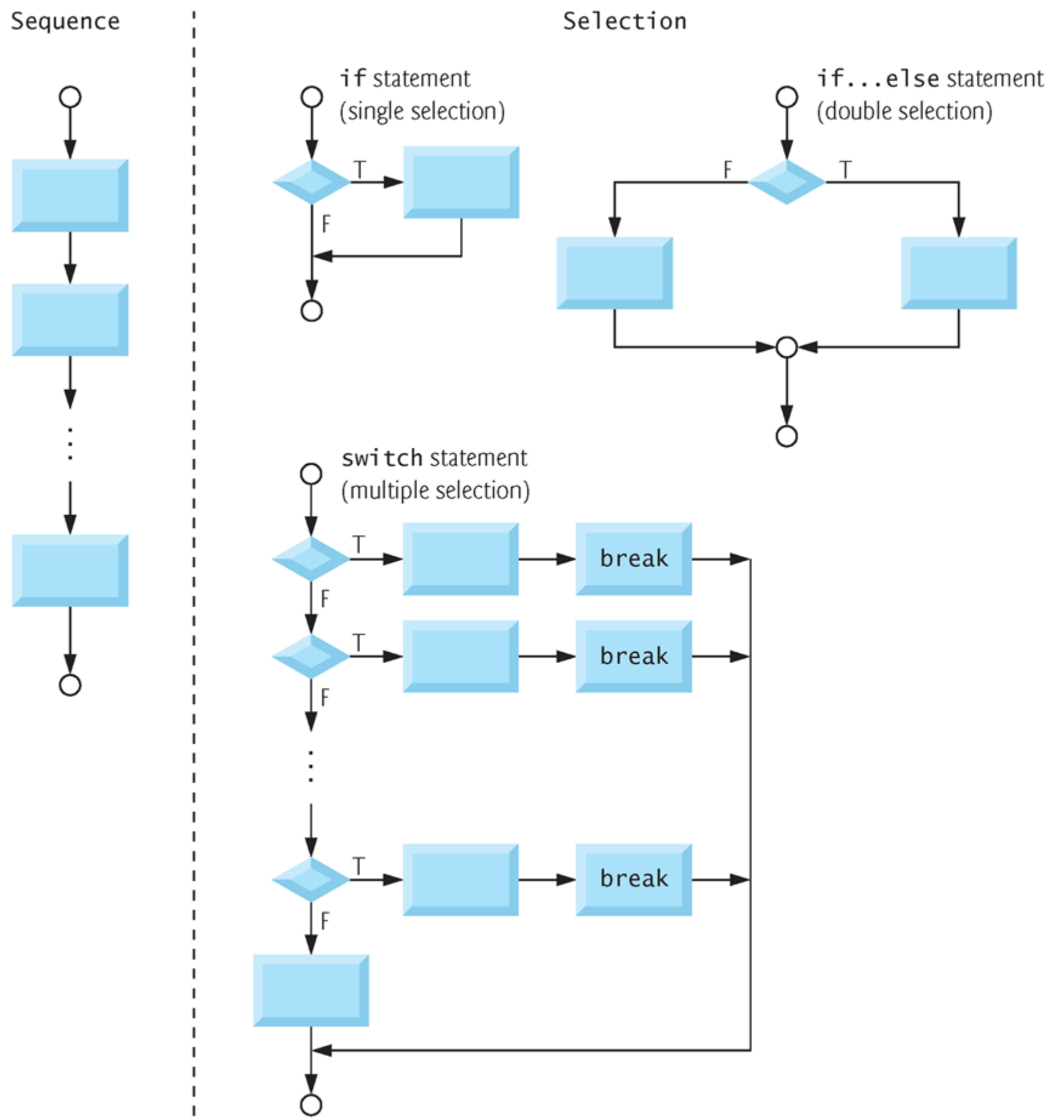

Fig. 4.17 | C's single-entry/single-exit sequence, selection and iteration statements. (Part 1 of 2.)

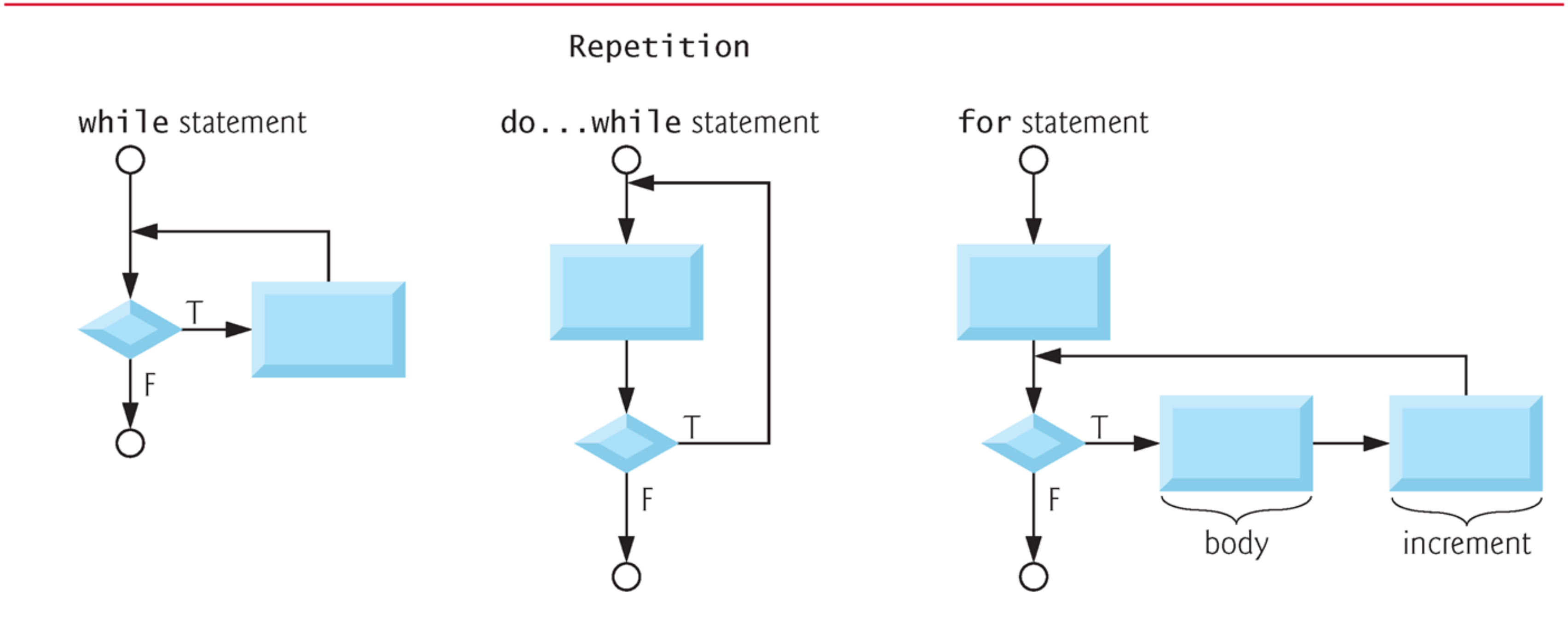

Fig. 4.17 | C's single-entry/single-exit sequence, selection and iteration statements. (Part 2 of 2.)

4.12 Structured Programming Summary

- Figure 4.17 summarizes the control statements discussed in Chapters 3 and 4.
- For simplicity, only single-entry/single-exit control statements are used—there is only one way to enter and only one way to exit each control statement.
- The beauty of the structured approach is that we use only a small number of simple single-entry/single-exit pieces, and we assemble them in only two simple ways.

Rules for forming structured programs
1. Begin with the "simplest flowchart" (Fig. 4.19). 2. ("Stacking" rule) Any rectangle (action) can be replaced by *two* rectangles (actions) in sequence. 3. ("Nesting" rule) Any rectangle (action) can be replaced by *any* control statement (sequence, `if`, `if...else`, `switch`, `while`, `do...while` or `for`). 4. Rules 2 and 3 may be applied as often as you like and in *any* order.

Fig. 4.18 | Rules for forming structured programs.

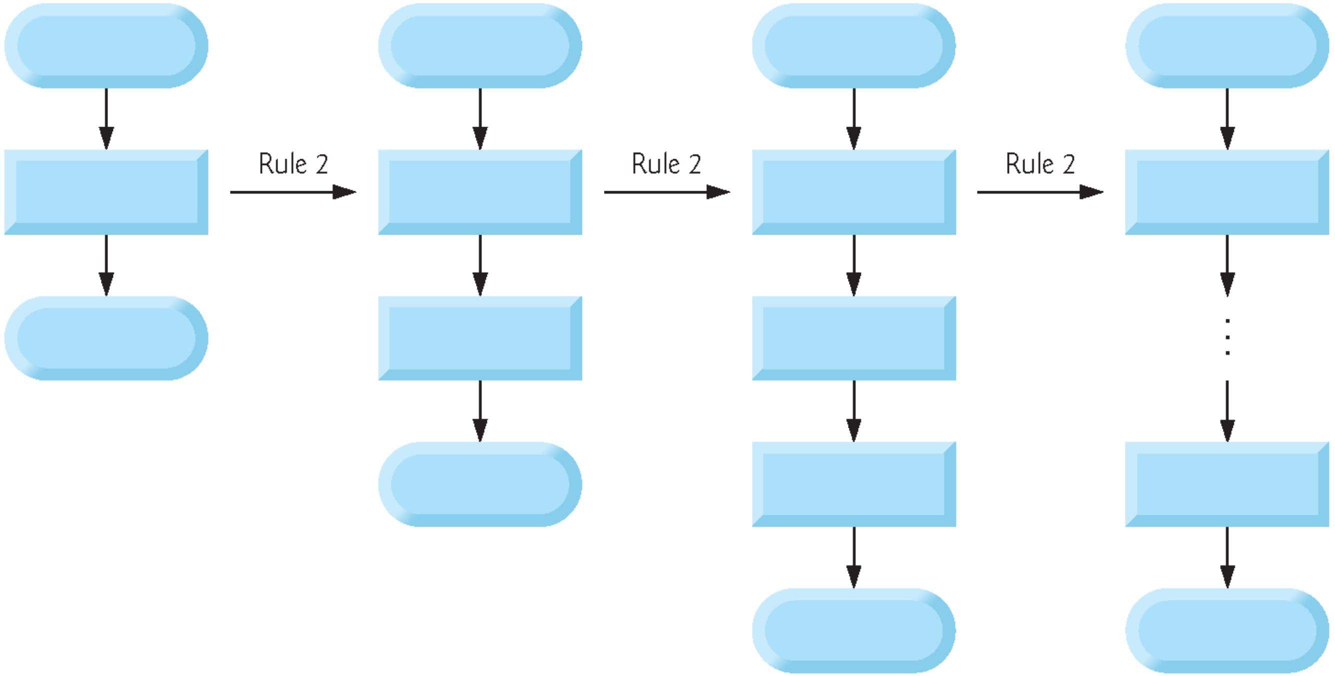

Fig. 4.20 | Repeatedly applying Rule 2 of Fig. 4.18 to the simplest flowchart.

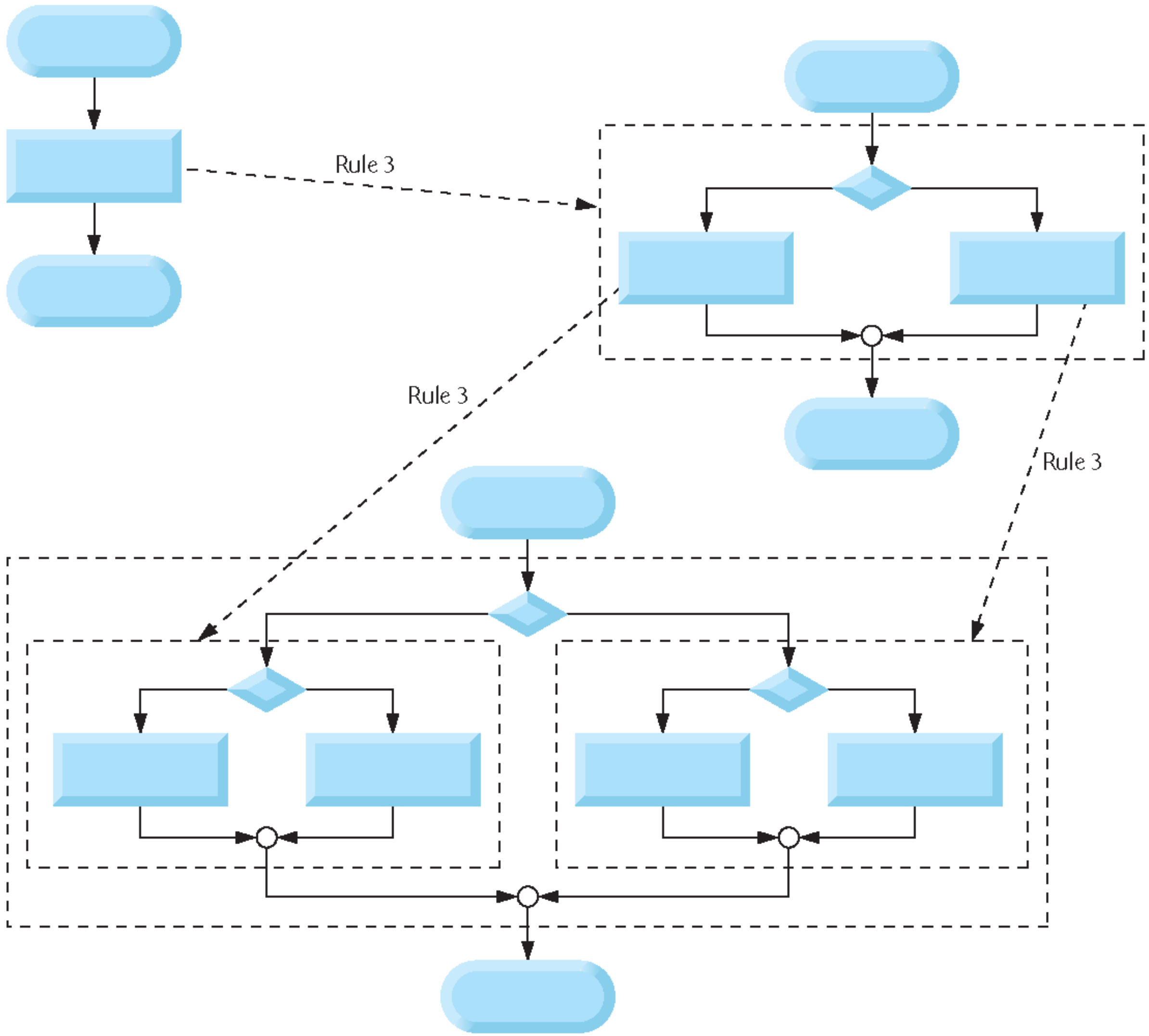

Fig. 4.21 | Applying Rule 3 of Fig. 4.18 to the simplest flowchart.

4.12 Structured Programming Summary (Cont.)

- Structured programming promotes simplicity.
- Bohm and Jacopini showed that only three forms of control are needed:
  - Sequence
  - Selection
  - Repetition