Agenda

- The for repetition statement handles all the details of counter-controlled repetition. To illustrate its power, let's write the program of Fig. 1.1.

```
1   // Counter-controlled repetition with the for statement.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      unsinged int counter; // number of grade to be entered next
8
9      // initialization, repetition condition, and increment
10     // are all included in the for statement header.
11     for( counter = 1; counter <= 10; ++counter ) {
12        printf( "%u\n", counter );
13     } //end for
14  } //end function main
```

Fig. 1.1 | Counter-controlled repetition with the for statement

- 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 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.

- 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.

Figure 1.2 takes a closer look at the for statement. 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's more than one statement in the body of the for, braces are required to define the body of the loop.

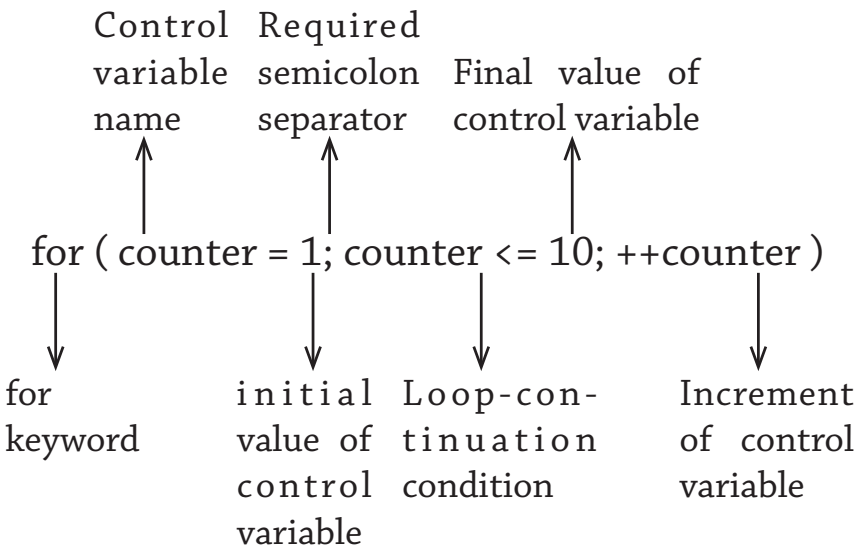

Fig 1.2 | for statement header components.

The C standard allows you to declare the control variable in the initialization section of the for header as in int counter = 1.

- Notice that 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.

Error-Prevention Tip
Using the final value in the condition of a while or for statement and using the <= relational operator can help avoid off-by-one errors. For a loop used to print the values 1 to 10, for example, the loop-continuation condition should be counter <= 10 rather than counter < 11 or counter < 10.

C Program Control
for Repetition Statement--General Format of a for Statement

- The general format of the for statement is

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

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

- Often, expression1 and expression3 are comma-separated lists of expressions. The commas as used here are actually comma operatos that guarantee that list of expression evaluate from left to right.

- 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.

Software Engineering Observation
Place only expressions involving the control variables in the initialization and increment sections of a for statement. Manipulations of other variables should appear either before the loop or in the loop body.

for Repetition Statement--Expression in the for Statement's Header Are Optional

- The three expressions in the for statements 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 in calculated by statements in the body of the for statement of if no increment is needed.

for Repetition Statement--Increment Expression Acts Like a Standalone Statement

- The increment expression in the for statement acts like a stand-alone C statement at the end of the body of the for. Therefore the expression counter = counter + 1, counter += 1, ++counter, counter++ are all equivalent in the increment part of the for statement.

- Some C programmers prefer the form counter++ because the increment occurs after the loop body is executed, and the postincrementing for seems more natural.

Common Programming Error

Placing a semicolon immediately to the right of a for header makes the body of that for statement an empty statement. This is normally a logic error.

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 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.

- The control variable is frequently printed or used in calculations in the body of a loop, but it need not be. It's common to use the control variable for controlling repetition while never mentioning it in the body of the loop.

for Statement: Notes and Observations

- The for statement is flowcharted much like the while statement. For example, Fig. 1.4 shows the flowchart of the for statement

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

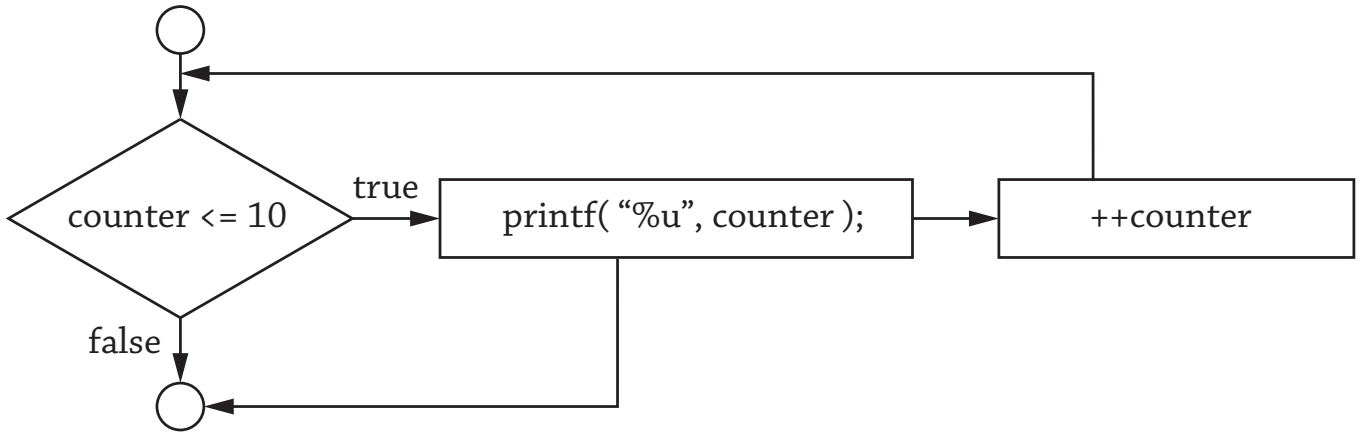

Fig. 1.4 | Flowcharting a typical for repetition statement

Error-Prevention Tip

Although the value of the control variable can be changed in the body of a for loop, this can lead to subtle errors. It's best no to change it.

for Statement: Examples Using the 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: 44, 33, 22, 11, 0.

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

switch Multiple-Selection Statement

Occasionally, an algorithm will contain a series of decisions in which a variable or expresion is tested separately for each of the constant integral values it may assume, and different actions are taken. This is called multiple selection. C provides the switch multiple-selection statement to handle such decision making.

The switch statement consists of a series of case labels, an optional default case and statements to execute for each case. Figure 1.5 uses switch to count the number of each different letter grade students earned on an exam.

```
1   // Counting letter grades with switch.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      int grade; // one grade
8      unsingned int aCount = 0; // number of As
9      unsingned int bCount = 0; // number of Bs
10     unsingned int cCount = 0; // number of Cs
11     unsingned int dCount = 0; // number of Ds
12     unsingned int fCount = 0; // number of Fs
13
14     puts( “Enter the letter grades.” );
15     puts( “Enter the EOF character to end input.” );
16
```

```
17  // loop until user types end-of-file key sequence
18  while( ( grade = getchar() ) != EOF ){
19    // Counting letter grades with switch.
20    switch( grade ){ // switch nested in while
21
22      case 'A': // grade was uppercase A
23      case 'a': // or lowercase a
24      ++aCount // increment aCount
25      break; //necessary to exit switch
26
27      case 'B': // grade was uppercase B
28      case 'b': // or lowercase b
29      ++bCount // increment bCount
30      break; //necessary to exit switch
31
32      case 'C': // grade was uppercase C
33      case 'c': // or lowercase c
34      ++cCount // increment cCount
35      break; //necessary to exit switch
36
37      case 'D': // grade was uppercase D
38      case 'd': // or lowercase d
39      ++dCount // increment dCount
40      break; //necessary to exit switch
41
42       case 'F': // grade was uppercase F
43       case 'f': // or lowercase f
44       ++fCount // increment fCount
45       break; //necessary to exit switch
46
```

```
47          case '\n': // ignore newlines,
48          case '\t': // tabs,
49          case ' ': // and spaces in input
50          break; // exit switch
51
52          default: // catch all other characters
53          printf( "%s", "Incorrect letter grade entered." );
54          puts( "Enter a new grade." );
55          break; // optional, will exit switch anyway
56       } // end switch
57    } // end while
58
59       //output summary of results
60       puts( "\nTotals for each letter grade are:" );
61       printf( "A: %u\n", aCount  ); //display number of A grades
62       printf( "B: %u\n", bCount  ); //display number of B grades
63       printf( "C: %u\n", cCount  ); //display number of C grades
64       printf( "D: %u\n", dCount  ); //display number of D grades
65       printf( "F: %u\n", dCount  ); //display number of F grades
66    } // end function main
```

Fig. 1.5 | Counter letter grades with switch.

switch Multiple-Selection Statement--Reading Character Input

- In the program, the user enters letter grades for a class. In the while header while((grade = getchar()) != EOF) the parenthesized assignment (grade = getchar()) executes first. The getchar function (from <stdio.h>) reads one character from the keyboard and stores that character in the integer variable grade.

- Characters are normally stored in variables of type char. However, an important feature of C is that characters can be stored in any integer data type because they're usually represented as one-byte integers in the computer. Thus, 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');

  uses the conversion specifier %c and %d to print the character a and its integer value, respectively.

- The result is

  The character (a) has the value 97.

  The integer 97 is the character's numerical representation in the computer. Many computers today use the ASCII /(American Standard Code for Information Interchange) character set in which 97 represent the lowercase letter 'a'.

- In the program, the value of the assignment grade = getchar() is compared with the value of EOF (a symbol whose acronym stands for “end of file”). We use EOF (which normally has the value -1) as the sentinel value. The user types a system-dependent keystroke combination to mean “end of file”--i.e., “I have no more data to enter.”.

- EOF is a symbolic integer constant defined in the <stdio.h> header. If the value assigned to grade is equal to EOF, the program terminates. We’ve chosen to represent characters in this program as ints because EOF has an integer value.

Portability Tip
The keystroke combinations for entering EOF (end of file) are system depended.

Portability Tip
Testing for the symbolic constant EOF (rather than -1) makes programs more portable. The C standard states that EOF is a negative integral value (but not necessarily -1). Thus, EOF could have different values on different system.

switch Multiple-Selection Statement--Entering the EOF Indicator

- On Linux/UNIX/MAC OS X systems, the EOF indicator is entered by typing <Ctrl> D on a line by itself. This notation <Ctrl> D means to press the Enter key and then simultaneously press both the Ctrl key and the d key. On other systems, such as Microsoft Windwos, the EOF indicator can be entered by typing <Ctrl> Z.

- The user enters grades at the keyboard. When the Enter key is pressed, the characters are read by function getchar one character at a time. If the character entered is not equal to EOF, the switch statement is entered.

- Keyword switch is followed by the variable name grade in parentheses. This is called the controlling expression. The value of this expression is compared with each of the case labels. Assume the user has entered the letter C as a grade. C is automatically compared to each case in the switch. If a match occurs, the statements for that case are executed. In the case of the letter C, cCount is incremented by 1, and the switch statement is exited immediately with the break statement.

- The break statement causes program control to continue with the first statement after the switch statement. The break statement is used because the cases in a switch statement would otherwise run together. 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 cases will be executed. If no match occurs, the default case is executed, and an error message is printed.

switch Multiple-Selection Statement--switch Statement Flowchart

- Each case can have one or more actions. The switch statement is different from all other control statements in that braces are not required around multiple actions in a case of a switch. The general switch multiple-seelction statement (using a break in each case) is flowcharted in Fig. 1.6. The flowchart makes it clear that each break statement at the end of a case causes control to immediately exit the switch statement.

Common Programming Error
Forgetting a break statement when one is needed in a switch statement is a logic error.

Software Engineering Observation
Provide a default case in switch statements. Cases not explicitly tested in a switch are ignored. The default case helps prevent this by focusing you on the need to precess exceptional conditions. Sometimes no default prcessing is needed.

Good Programming Practive
Although the case clauses and the default case clause in a switch statement can occur in any order, it's common to place the default clause last.

Good Programming Practive
In a switch statement when the default clause is last, the break statement isn't required. You may prefer to include this break for clarity and symmetry with other cases.

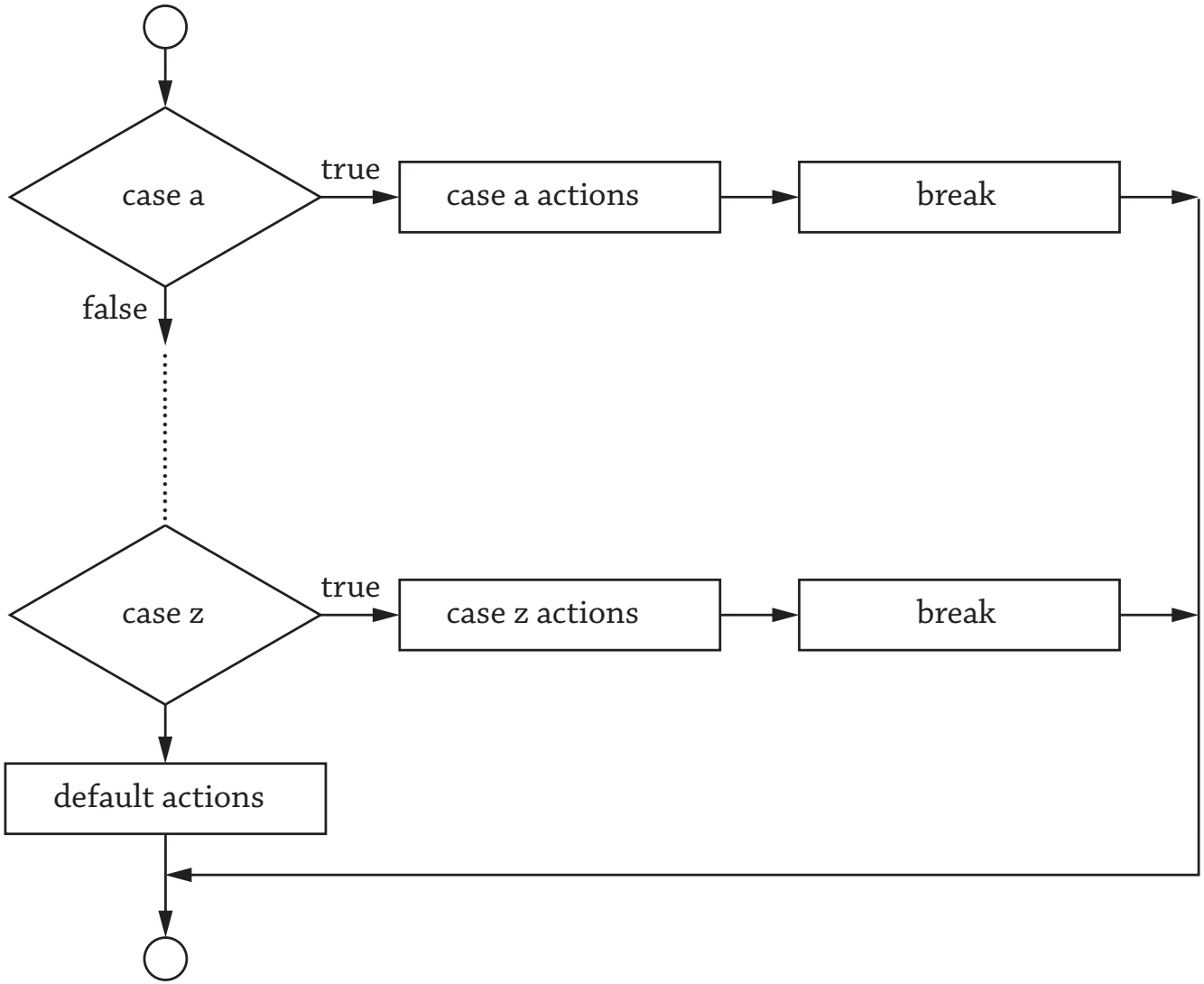

Fig. 1.6 | switch multiple-selection statement with breaks

switch Multiple-Selection Statement--Ignoring Newline, Tab and Blank Charcters in Input

- In the switch statement, the lines

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

cause the problem to skip newline, tab and blank characters. Reading characters one at a time can cause some problems. To have the program read the characters, you must send them to the computer by pressing the Enter key. This causes the new line character to be placed in the input after the character we wish to process. 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.

Error-Preventing Tip

Remember to provide processing capabilities for newline (any possibly other white-space) characters in the input when processing characters one at a time.

switch Multiple-Selection Statement--Constant Integral Expressions

- When using the switch statement, remember that each individual case can test only a constant integral expression--i.e., any combination of character constant and integer constants that evaluates to a constant integer value. A character constant can be represented as the specific character in single quotes such as ‘A’.

- Characters must be enclosed within single quotes to be recognized as character constants--characters in double qotes are recognized as strings. Integer constants are simply integer values. In our example, we’ve used character constraints.

- Portable languages like C must have flexible data type sizes. Different applications may need integers of different sizes. C provides several data types to represent integer. In addition int and char, C provides types short int (which can be abbreviated as short) and long int (which can be abbreviated as long).

- The C standard specifies the minimum range of values for each integer type, but the actual range may be greater and depends on the implementation. For short ints the minimum range -32767 to +32767. For most integer calculations, long ints are sufficient. The minimum range of values for long ints -2147483647 to +2147483647.

- The range of values for an int greater than or equal to that of a short int and less than or equal to that of a long int. On many of today's platforms, ints and long ints represent the same range of values. The data type signed char can be used to represent integer in the range -127 to +127 or any of the characters in the computer's character set.

do...while Repetition Statement

The do...while repetition statement is similar to the while statement. In the while statement, the loop-continuation condition is tested at the beginning of the loop before the body of the loop is performed. The do...while statement tests the loop-continuation condition after the loop body is performed. Therefore, the loop body will be executed at least one.

When a do...while terminates, execution continues with the statement after the while clause. It's not necessary to use braces in the do...while statement if there's only one statement in the body. However, the braces are usually included to avoid confision between the while and do...while statements. For example,

```
while ( condition )
```

is normally regarded as the header to a while statement. A do...while with no braces around the single-statement appears as

```
do
  Statement;
while ( condition );
```

which can be confusing. The last line--while (condition);--may be misinterpreted as a while statement containing an empty statement. Thus, to avoid confusion, the do...while with one statement is often written as follows:

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

do…while Repetition Statement

- Fig 1.7 uses a do…while statement to print the numbers from 1 to 10. The control variable counter is preincremented in the loop-continuation test.

```
1   // Using the do…while repetition statement.
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      unsigned int counter = 1; // initialize counter
8      do {
9         printf( "%u ", counter ); // display counter
10     } while ( ++counter <= 10 ); // end do…while
11  } // end function main
```

Fig. 1.6 | Using the do…while repetition statement

do...while Repetition Statement--do...while Statement Flowchart

- Figure 1.7 shows the do...while statement flowchart, which makes it clear that the loop-continuation condition does not execute until after the action is performed at least once.

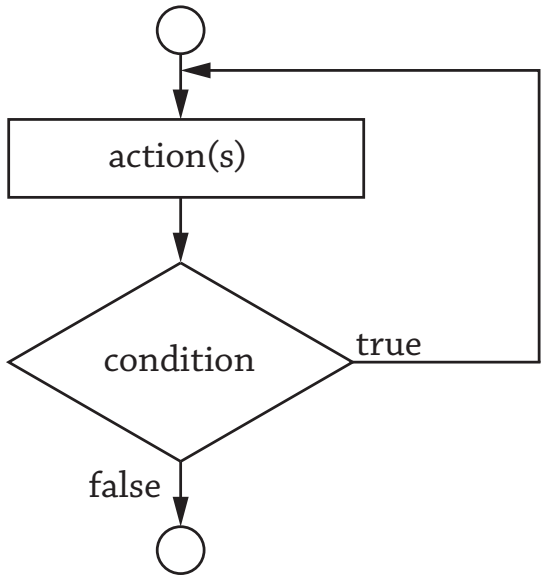

Fig. 1.7 | Flowcharting the do...while repetition statement.

break and continue Statements--break Statement

- The break and continue statements are used to alter the flow of control. It is showed how break can be used to terminate a switch statement's execution.

- The break statement, when executed in a while, for, do...while or switch statement, causes an immediate exit from that statement. Program execution continues with the next statement. Common uses of the break statement are to escape early from a loop or to skip the remainder or a switch statement. Figure 1.8 demontrates the break statement in a for repetition statement.

```
1   // Using the break statement in a for statement
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      unsigned int x; // counter
8      for ( x = 1; x <= 10; ++x ) {
9         if ( x == 5 ) {
10           break; // break loop only if x is 5
11        } // end if
12        printf( "%u ", x ); // display value of x
13     } // end for
14     printf( "\nBroke out of loop a x == %u\n", x );
15  } // end function main
```

Fig. 1.8 | Using the break statement in a for statemet

break and continue Statements--continue Statement

- The continue statement, when executed in a while, for, or do...while statement, skips the remaining statements in the body of that control statement and performs the next iteration of the loop.

- In while and do...while statements, the loop-continuation test is evaluated immediately after the continue statement is executed. In the for statement, the increment expression is executed, then the loop-continuation test is evaluated.

- Fig. 1.9 uses the continue statement in a for statement to skip the printf statement and begin the next iteration of the loop.

```
1   // Using the continue statement in a for statement
2   #include <stdio.h>
3
4   // function main begins program execution
5   int main( void )
6   {
7      unsigned int x; // counter
8      for ( x = 1; x <= 10; ++x ) {
9         if ( x == 5 ) {
10           continue; // break loop only if x is 5
11        } // end if
12        printf( "%u ", x ); // display value of x
13     } // end for
14     puts( "\nUsed continue to skip printing the value 5" );
15  } // end function main
```

Fig. 1.9 | Using the continue statement in a for statement

Software Engineering Observation

Some programmers feel that break and continue violate the norms of structured programming. The effects of these statements can be achieved by structured programming techniques, so these programmers do not use break and continue.

Logical Operators

- So far, we've studied only simple conditions, such as counter <= 10, total > 1000, and number != sentinelValue. We've expressed these conditions in terms of the relational operators, >, <, <= and <=, and the equality operators, == and !=. Each decision tested precisely one condition.

- To test multiple conditions in the process of making a decision, we had to perform these tests in separate statements or in nested if or if...else statements. C provides logical operators that may be used to form more complex conditions by combining simple conditions.

- The logical operators are && (logical AND), || (logical OR) and ! (logical NOT, also called logical negation).

Logical Operators--Logical AND (&&) Operator

- Suppose we wish to ensure that two conditions are both true before we choose a certain path of execution. In this case, we can use the logical operator && as follows:

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

- Two simple conditions are evaluated first because == and >= are have higher precedence that &&. The if statement then considers the combined condition gener == 1 && age >= 65, which is true if and only if both of the simple conditions are true. Finally, if this combined condition is true, then the count of seniorFemales is incremented by 1.

- If either or both of the simple conditions are false, then the program skips the incrementing and proceeds to the statement following the if.

Logical Operators--Logical OR (||) Operator

- Suppose we wish to ensure at some point in a program that either or both of two conditions are true before we choose a certain path of execution. In this case, we use the ||operator, as in the following program segment:

```
if ( semesterAverage >= 90 || finalExam >= 90 )
  puts( "Student grade is A" );
```

- The if statement then considers the combined condition semesterAverage >= 90 || finalExam >= 90 and awards the student an "A" if either or both of the simple conditions are true.

- The && operator has a higher precedence than ||. Both operators associate from left to right. An expression containing && or ||operators is evaluated only until truth or falsehood is known. Thus, evaluation of the condition

```
gender == 1 && age >= 65
```

will stop if gender is not equal to 1, and continue if gender is equal to 1. This performance feature for the evaluation of logical AND and logical OR expression is called short-circuit evaluation.

Performance Tip

In expression using operator &&, make the conditions that's most likely to be false the left-most condition. In expression using operator ||, make the conditions that's most likely to be ture the left-most condition. This can reduce a program's execution time.

Logical Operators--Logical Negation (!) Operator

- C provides ! (logical negation) to enable you to "reverse" the meaning of a condition. Unlike operators && and ||, which combine two conditions (and are therefore binary operators), the logical negation operator has only a single condition as an operand (and is therefore a unary operator).

- The logical negation operator is placed before a condition when we're interested in choosing a path of execution if the original condition (without the logical negation operator) is false, such as in the following program segment:

```
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.

- In most cases, you can avoid using logical negation by expressing the condition differently with and 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 );
```

Logical Operators--Summary of Operator and Associativity

Figure 1.10 shows the precedence and associativity of the operators introduced to this point. The operators are shown from top to bottom in decreasing order of precedence.

Operators	Associativity	Type
++ (postfix) --(postfix)	right to left	postfix
+, -, !, ++(prefix) --(prefix), (type)	right to left	unary
*, /, %	left to right	multiplication
+, -	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. 1.10 | Operator precedence and associativity

Logical Operators--The _Bool Data Type

- The C standard includes a boolean type--represented by the keyword _Bool--which can hold only the values 0 and 1. Recall C's convention of using zero and nonzero values to represent false and true--the value 0 in a condition evaluates to false, while any nonzero value evaluates to true.

- Assigning any non-zero value to a _Bool sets it to 1. The standard also includes the <stdbool.h> header, which defines bool as a shorthand for the type _Bool, and true and false as named representations of 1 and 0, respectively.

Confusing Equality (==) and Assignment (=) Operators

There's one type of error that C programmers, no matter how experienced, tend to make so frequently that we felt it was worth a separate section. That error is accidentally swapping the operators == and =. What makes these swaps so damaging is the fact that they do not ordinarily cause compilation errors. Rather, statements with these errors ordinarily compile correctly, allowing programs to run to completion while likely generating incorrect results through runtime logic errors.

Two aspects of C cause these problems. One is that any expression in C that produces a value can be used in the decision portion of any control statement. If the value is 0, it's treated as false, and if the value is nonzero, it's treated as true.

The second is that assignment in C produce a value, namely the value that's assigned to the variable on the left side of the assignment operator. For example, 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!" );
```

The first if statement properly awards a bonus to the person whose paycode is equal to 4. The second if statement--the one with error--evaluates the assignment expression in the if condition. This expression is a simple assignment whose value is the contant 4. Because any nonzero value is interpreted as ture, the condition in this if statement is always true, and not only is the value of payCode inadvertantly set to 4, but the person always receives a bonus regardless of what the actual paycode is!

Confusing Equality (==) and Assignment (=) Operators--lvalues and rvalues

- You'll probably be inclined to write conditions such as x==7 with the variable name on the left and the constant on the right. By reversing these terms so that the constant is on the left and the variable name is on the right, as in 7==x, then if you accidentally replace the == operator with =, you'll be protected by the compiler.

- The compiler will treat this assignment a syntax error, because only a variable name can be placed on the left-hand side of an assignment expression. This will prevent the potential devastation of a runtime logic error.

- Variable names are said to be lvalues (for "left values") because they can be used on the left side of an assignment operator. Constants are said to be rvalues (for "right values") because they can be used on only the right side of an assignment operator. lvalues can also be used as rvalues, but not vice versa.

Confusing Equality (==) and Assignment (=) Operators--Confusing == and = in Standalone

- The other side of the coin can be equally unpleasant. Suppose you want to assign a value to a variable with a simple statement such as

  x = 1;

  but instead write

  x == 1;

- Here, too, this is not a syntax error. Rather the compiler simply evaluates the conditional expression. If x is equal to 1, the condition is true and the expression returns the value 1. If x is not equal to 1, the condition is false the the expression returns the value 0.

- Regardless of what value is returned, there's no assignment operator, so the value is simple lost, and the value of x remains unaltered, probably causing an execution-time logical error. Unfortunately, we do not have a handy trick available to help you with this problem. Many compilers, however, will issue a warning on such a statement.

Error-Prevention Tip
After you write a program, text search is for every = and check that it's used properly.

- Connecting control statements in sequence to form structured programs is simple--the exit point of one control statement is connected directly to the entry point of the next--i.e., the control statements are simply placed one after another in program--we've called this "control statement stacking". The rules for forming structured programs also allow for control statement to be nested.

- We shows the rules for forming structured programs. The rules assume that the rectangle flowchart symbol may be used to indicate any action including input/output. Figure 4.11 shows the simplest flowchart.

1) Begin with the simplest flowchart. (Fig. 4.11)
2) Any rectange (action) can be replaced by two rectangles (actions) in sequence.
3) 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.

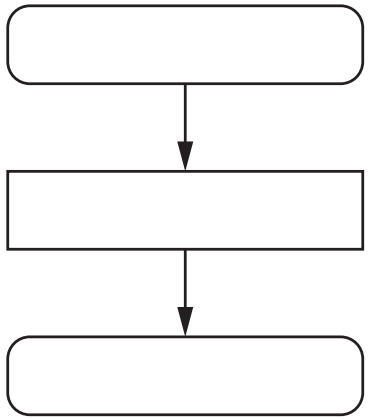

Fig. 4.11 | Simplest flowchart

- Applying the rules always results in a structured flowchart with a neat, building-block appearance. Repeadly applying Rule 2 to the simplest flowchart results in a structured flowchart containing many rectangles in sequence. Rule 2 generates a stack of control statements; so we call Rule 2 the stacking rule.

- Rule 3 is called the nesting rule. Repeatedly applying Rule 3 to the simplest flowchart results in a flowchart with neatly nested control statements. For example, in Fig 4.11, the rectangle in the simplest flowchart is first replaced with a double selection statement.

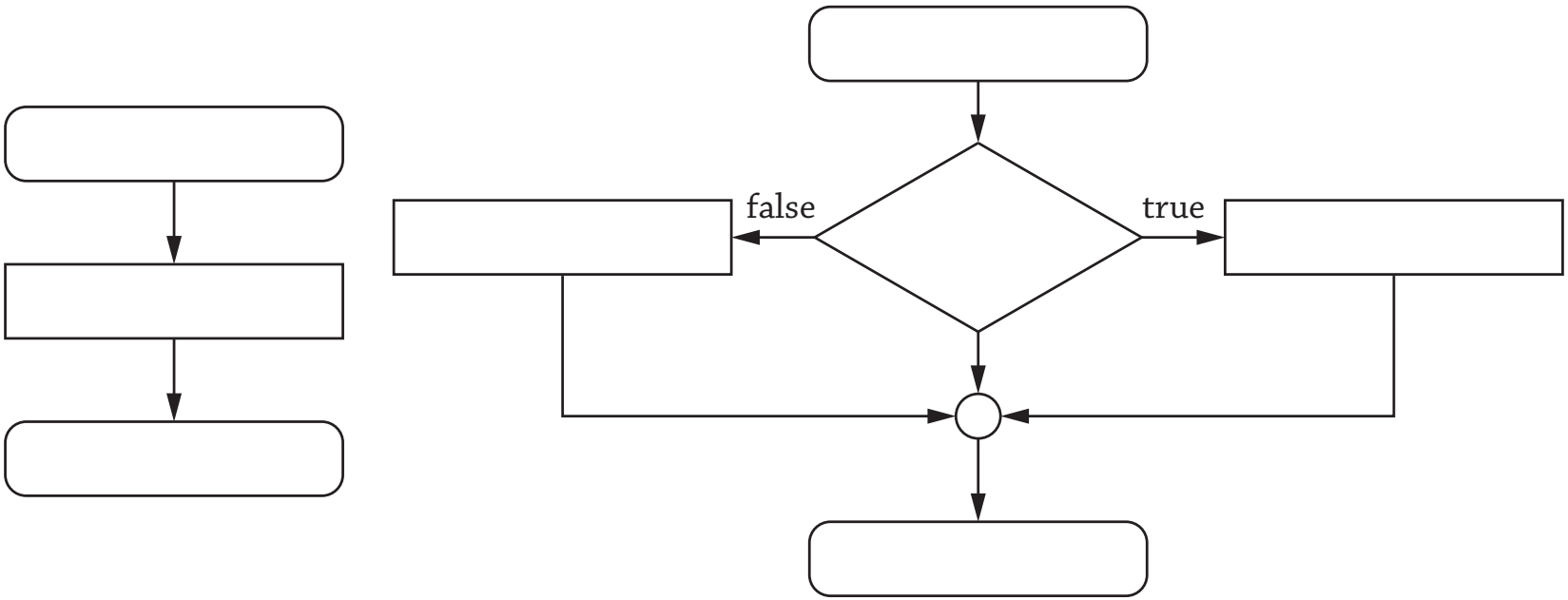

Fig. 4.12 | Simplest flowchart replaced with a double selection statement