BAKI ALİ NEFT MƏKTƏBİ
BAKU HIGHER OIL SCHOOL

Programming and Computer Applications-1

Program Control

Instructor : PhD, Associate Professor Leyla Muradkhanli

Outline

- Counter-controlled repetition
- for repetition statement
- switch multiple selection statement
- do…while repetition statement
- break and continue statements
- Logical operators

Repetition Essentials

- A loop is a group of instructions the computer executes repeatedly while some loop-continuation condition remains true.
- We have discussed two means of repetition:
  - Counter-controlled repetition
  - Sentinel-controlled repetition
- Counter-controlled repetition is sometimes called definite repetition because we know in advance exactly how many times the loop will be executed.
- Sentinel-controlled repetition is sometimes called indefinite repetition because it’s not known in advance how many times the loop will be executed.

Repetition Essentials (Cont.)

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

Repetition Essentials (Cont.)

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

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).

Counter-Controlled Repetition (Cont.)

- Consider the simple program shown in Fig. 4.1, which prints the numbers from 1 to 10.
- The definition

```
    int counter = 1; /* initialization */
```

  names the control variable (`counter`), defines it to be an integer, reserves memory space for it, and sets it to an initial value of `1`.
- This definition is not an executable statement.

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

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

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

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

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

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

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

Counter-Controlled Repetition (Cont.)

- The definition and initialization of `counter` could also have been written as

```
int counter;
counter = 1;
```

- The definition is not executable, but the assignment is.
- We use both methods of initializing variables.
- The statement

```
++counter; /* increment */
```

increments the loop counter by 1 each time the loop is performed.

Counter-Controlled Repetition (Cont.)

- The loop-continuation condition in the `while` statement tests if the value of the control variable is less than or equal to `10` (the last value for which the condition is true).
- The body of this `while` is performed even when the control variable is `10`.
- The loop terminates when the control variable exceeds `10` (i.e., `counter` becomes `11`).

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( "%d\n", counter );
```

- This code saves a statement because the incrementing is done directly in the `while` condition before the condition is tested.
- Also, this code eliminates the need for the braces around the body of the `while` because the `while` now contains only one statement.

`for` Repetition Statement

- The `for` repetition statement handles all the details of counter-controlled repetition.
- To illustrate its power, let’s rewrite the program of Fig. 4.1.
- The result is shown in Fig. 4.2.

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

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

   /* initialization, repetition condition, and increment
      are all included in the for statement header. */
   for ( counter = 1; counter <= 10; counter++ ) {
      printf( "%d\n", counter );
   } /* end for */

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

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

`for` Repetition Statement (Cont.)

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

`for` Repetition Statement (Cont.)

- Since the control variable is now equal to `2`, the final value is not exceeded, so the program performs the `printf` statement again.
- This process continues until the control variable `counter` is incremented to its final value of 11—this causes the loop-continuation test to fail, and repetition terminates.
- The program continues by performing the first statement after the `for` statement (in this case, the `return` statement at the end of the program).
- Figure 4.3 takes a closer look at the `for` statement of Fig. 4.2.

`for` Repetition Statement (Cont.)

- Notice that the `for` statement “does it all”—it specifies each of the items needed for counter-controlled repetition with a control variable.
- If there is more than one statement in the body of the `for`, braces are required to define the body of the loop.

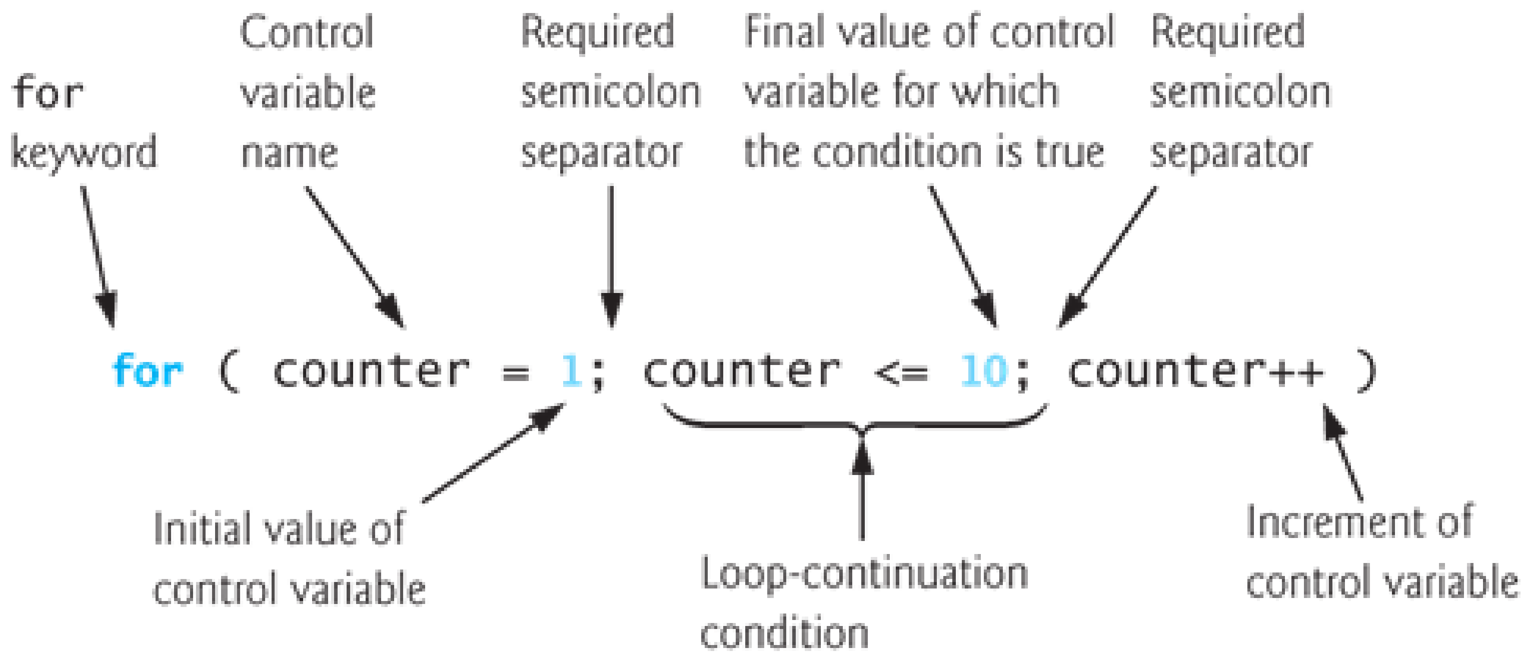
for keyword
Control variable name
Required semicolon separator
Final value of control variable for which the condition is true
Required semicolon separator
for (counter = 1; counter <= 10; counter++)
Initial value of control variable
Loop-continuation condition
Increment of control variable

Fig. 4.3 | `for` statement header components.

`for` Repetition Statement (Cont.)

- 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.

`for` Repetition Statement (Cont.)

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

  where *expression1* initializes the loop-control variable, *expression2* is the loop-continuation condition, and *expression3* increments the control variable.
- In most cases, the `for` statement can be represented with an equivalent `while` statement as follows:

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

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 )
    ```

Examples Using the `for` Statement (Cont.)

- The next example provides simple applications of the `for` statement.
- Figure 4.5 uses the `for` statement to sum all the even integers from `2` to `100`.

```c
/* Fig. 4.5: fig04_05.c
   Summation with for */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int sum = 0; /* initialize sum */
   int number; /* number to be added to sum */

   for ( number = 2; number <= 100; number += 2 ) {
      sum += number; /* add number to sum */
   } /* end for */

   printf( "Sum is %d\n", sum ); /* output sum */
   return 0; /* indicate program ended successfully */
} /* end function main */
```

```
Sum is 2550
```

Fig. 4.5 | Using for to sum numbers.

Examples Using the `for` Statement (Cont.)

- The body of the `for` statement in Fig. 4.5 could actually be merged into the rightmost portion of the `for` header by using the comma operator as follows:

```
for ( number = 2; number <= 100; sum += number,
number += 2 )
   ; /* empty statement */
```

- The initialization `sum = 0` could also be merged into the initialization section of the `for`.

`switch` Multiple-Selection Statement

- Occasionally, an algorithm will contain a series of decisions in which a variable or expression 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 4.7 uses `switch` to count the number of each different letter grade students earned on an exam.

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

/* function main begins program execution */
int main( void )
{
   int grade; /* one grade */
   int aCount = 0; /* number of As */
   int bCount = 0; /* number of Bs */
   int cCount = 0; /* number of Cs */
   int dCount = 0; /* number of Ds */
   int fCount = 0; /* number of Fs */

   printf(  "Enter the letter grades.\n"  );
   printf(  "Enter the EOF character to end input.\n"  );

```

Fig. 4.7 | switch example. (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; /* increment aCount */
             break; /* necessary to exit switch */

          case 'B': /* grade was uppercase B */
          case 'b': /* or lowercase b */
             ++bCount; /* increment bCount */
             break; /* exit switch */

          case 'C': /* grade was uppercase C */
          case 'c': /* or lowercase c */
             ++cCount; /* increment cCount */
             break; /* exit switch */

```

```

         case 'd': /* or lowercase d */
            ++dCount; /* increment dCount */
            break; /* exit switch */

         case 'F': /* grade was uppercase F */
         case 'f': /* or lowercase f */
            ++fCount; /* increment fCount */
            break; /* exit switch */

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

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

```

```
    /* output summary of results */
    printf( "\nTotals for each letter grade are:\n" );
    printf( "A: %d\n", aCount ); /* display number of A grades */
    printf( "B: %d\n", bCount ); /* display number of B grades */
    printf( "C: %d\n", cCount ); /* display number of C grades */
    printf( "D: %d\n", dCount ); /* display number of D grades */
    printf( "F: %d\n", fCount ); /* display number of F grades */c
    return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.7 | `switch` example. (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
```

Fig. 4.7 | switch example. (Part 5 of 5.)

switch Multiple-Selection Statement (Cont.)

- In the program, the user enters letter grades for a class.
- In the while header (line 19),
    - `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.

`switch` Multiple-Selection Statement (Cont.)

- 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 specifiers `%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.

`switch` Multiple-Selection Statement (Cont.)

- Many computers today use the ASCII (American Standard Code for Information Interchange) character set in which 97 represents the lowercase letter `'a'`.
- Characters can be read with `scanf` by using the conversion specifier `%c`.
- Assignments as a whole actually have a value.
- This value is assigned to the variable on the left side of `=`.
- The value of the assignment expression `grade = getchar()` is the character that is returned by `getchar` and assigned to the variable `grade`.

`switch` Multiple-Selection Statement (Cont.)

- The fact that assignments have values can be useful for setting several variables to the same value.
- For example,

```
        a = b = c = 0;
```

- first evaluates the assignment `c = 0` (because the = operator associates from right to left).
- The variable `b` is then assigned the value of the assignment `c = 0` (which is 0).
- Then, the variable `a` is assigned the value of the assignment `b = (c = 0)` (which is also 0).
- 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”).

`switch` Multiple-Selection Statement (Cont.)

- 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.
- Chose to represent characters in this program as `int`s because `EOF` has an integer value (normally `-1`).

`switch` Multiple-Selection Statement (Cont.)

- 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 then simultaneously press both *Ctrl* and *d*.
- On other systems, such as Microsoft Windows, the `EOF` indicator can be entered by typing

```
        <Ctrl>  z
```

- You may also need to press *Enter* on Windows.
- The user enters grades at the keyboard.

`switch` Multiple-Selection Statement (Cont.)

- 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 (line 22) 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`.

`switch` Multiple-Selection Statement (Cont.)

- If a match occurs (`case 'C':`), the statements for that `case` are executed.
- In the case of the letter `C`, `cCount` is incremented by `1` (line 36), 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 `case`s 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 `case`s will be executed.

`switch` Multiple-Selection Statement (Cont.)

- If no match occurs, the `default` case is executed, and an error message is printed.
- 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-selection statement (using a `break` in each `case`) is flowcharted in Fig. 4.8.
- The flowchart makes it clear that each `break` statement at the end of a `case` causes control to immediately exit the `switch` statement.

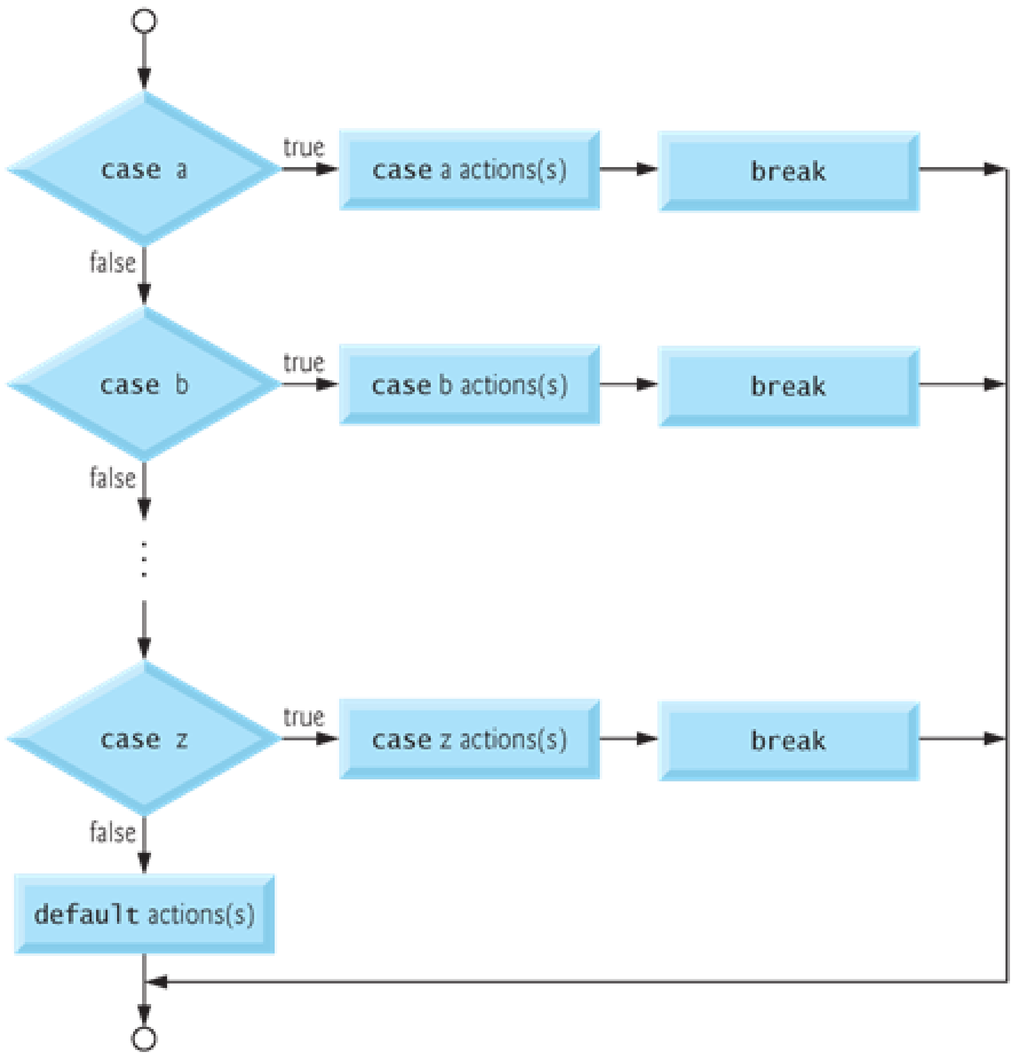

Fig. 4.8 | `switch` multiple-selection statement with `break`s.

`switch` Multiple-Selection Statement (Cont.)

- 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.
- Reading characters one at a time can cause some problems.

`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 once.
- When a `do…while` terminates, execution continues with the statement after the `while` clause.

`do…while` Repetition Statement (Cont.)

- It’s not necessary to use braces in the `do…while` statement if there is only one statement in the body.
- However, the braces are usually included to avoid confusion between the `while` and `do…while` statements.
- For example,

```
    while ( condition )
```

- is normally regarded as the header to a `while` statement.

do…while Repetition Statement (Cont.)

- A do…while with no braces around the single-statement body appears as

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

which can be confusing.

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

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.
- Note also the use of the braces to enclose the single-statement body of the do…while.

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

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

   do {
      printf( "%d  ", counter ); /* display counter */
   } while ( ++counter <= 10 ); /* end do...while */

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

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

Fig. 4.9 | do...while statement example.

`do…while` Repetition Statement (Cont.)

- Figure 4.10 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.

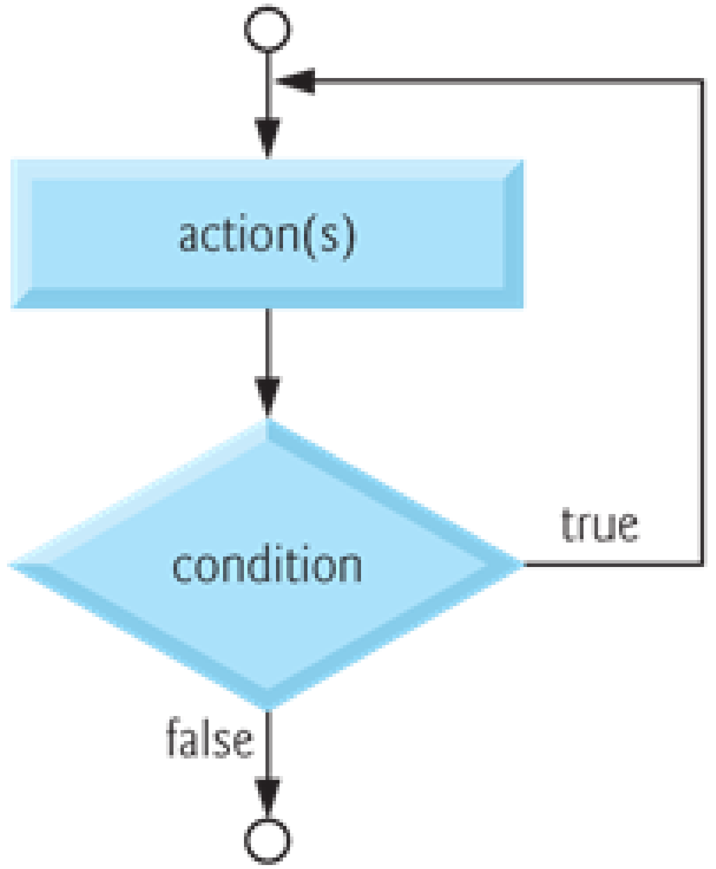

Fig. 4.10 | Flowcharting the `do...while` repetition statement.

`break` and `continue` Statements

- The `break` and `continue` statements are used to alter the flow of control.
- 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 of a `switch` statement (as in Fig. 4.7).

`break` and `continue` Statements (Cont.)

- Figure 4.11 demonstrates the break statement in a `for` repetition statement.
- When the `if` statement detects that `x` has become `5`, `break` is executed.
- This terminates the `for` statement, and the program continues with the `printf` after the `for`.
- The loop fully executes only four times.

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

/* function main begins program execution */
int main( void )
{
   int x; /* counter */

   /* 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 */
      } /* end if */

      printf( "%d ", x ); /* display value of x */
   } /* end for */

   printf( "\nBroke out of loop at x == %d\n", x );
   return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.11 | Using the break statement in a for statement. (Part 1 of 2.)

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

Fig. 4.11 | Using the `break` statement in a `for` statement. (Part 2 of 2.)

`break` and `continue` Statements (Cont.)

- 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.

`break` and `continue` Statements (Cont.)

- Earlier, we said that the `while` statement could be used in most cases to represent the `for` statement.
- The one exception occurs when the increment expression in the `while` statement follows the `continue` statement.
- In this case, the increment is not executed before the repetition-continuation condition is tested, and the `while` does not execute in the same manner as the `for`.
- Figure 4.12 uses the `continue` statement in a `for` statement to skip the `printf` statement and begin the next iteration of the loop.

```c
/* Fig. 4.12: fig04_12.c
   Using the continue statement in a for statement */
#include <stdio.h>

/* function main begins program execution */
int main( void )
{
   int x; /* counter */

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

      /* if x is 5, continue with next iteration of loop */
      if ( x == 5 ) {
         continue; /* skip remaining code in loop body */
      } /* end if */

      printf( "%d ", x ); /* display value of x */
   } /* end for */

   printf( "\nUsed continue to skip printing the value 5\n" );
   return 0; /* indicate program ended successfully */
} /* end function main */
```

Fig. 4.12 | Using the `continue` statement in a `for` statement. (Part 1 of 2.)

```
1 2 3 4 6 7 8 9 10
Used continue to skip printing the value 5
```

Fig. 4.12 | Using the `continue` statement in a `for` statement. (Part 2 of 2.)

Logical Operators

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

Logical Operators (Cont.)

- In this case, we can use the logical operator && as follows:

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

- This if statement contains two simple conditions.
- The condition gender == 1 might be evaluated, for example, to determine if a person is a female.
- The condition age >= 65 is evaluated to determine if a person is a senior citizen.
- The two simple conditions are evaluated first because the precedences of == and >= are both higher than the precedence of &&.

Logical Operators (Cont.)

- The `if` statement then considers the combined condition

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

- This condition is true if and only if both of the simple conditions are true.
- Finally, if this combined condition is indeed 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`.
- Figure 4.13 summarizes the && operator.

Logical Operators (Cont.)

- The table shows all four possible combinations of zero (false) and nonzero (true) values for expression1 and expression2.
- Such tables are often called truth tables.
- 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.

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.

Logical Operators (Cont.)

- Now let’s consider the `||` (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 )
  printf( "Student grade is A\n" );
```

- This statement also contains two simple conditions.
- The condition `semesterAverage >= 90` is evaluated to determine if the student deserves an “A” in the course because of a solid performance throughout the semester.

Logical Operators (Cont.)

- The condition `finalExam >= 90` is evaluated to determine if the student deserves an “A” in the course because of an outstanding performance on the final exam.
- 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 message “`Student grade is A`” is not printed only when both of the simple conditions are false (zero).
- Figure 4.14 is a truth table for the logical OR operator (`||`).

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.

Logical Operators (Cont.)

- 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` (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`).
- This performance feature for the evaluation of logical AND and logical OR expressions is called short-circuit evaluation.

Logical Operators (Cont.)

- C provides `!` (logical negation) to enable a programmer 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.
- Figure 4.15 is a truth table for the logical negation operator.

expression	!expression
0	1
nonzero	0

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

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 );
```

- Figure 4.16 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	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.