مبانی برنامه نویسی
به زبان سی

۲، ۴ و ۶ آذر ۱۳۹۹

جلسه هشتم، نهم و دهم

ملکی مجد

مباحث این هفته:

- `for`
- `switch`
  - (multiple-selection statement)
- `do`…`while`
- `break`
  - (statement for exiting immediately from certain control statements)
- `continue`
  - (statement for skipping the remainder of the body of a repetition statement and proceeding with the next iteration of the loop).
- Logical operators
  - (for combining conditions)

4.6 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: 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.)

- How 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.

```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.

```c
for (sum = 0 , number = 2; number <= 100; sum += number, number += 2 )
   ; /* empty statement */
```

Good Programming Practice 4.3

Although statements preceding a `for` and statements in the body of a `for` can often be merged into the `for` header, avoid doing so because it makes the program more difficult to read.

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

- Consider the following problem statement:
  - 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.

- This problem involves a **loop** that performs the indicated calculation for each of the 10 years the money remains on deposit.

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

/* function main begins program execution */
int main( void )
{
   double amount; /* amount on deposit */
   double principal = 1000.0; /* starting principal */
   double rate = .05; /* annual interest rate */
   int year; /* year counter */

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

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

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

      /* output one table row */
      printf( "%4d%21.2f\n", year, amount );
   } /* end for */

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

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

Although C does not include an exponentiation operator, we can use the Standard Library function `pow` for this purpose.

Type `double` is a floating-point type much like `float`, but typically a variable of type `double` can store a value of much greater magnitude with greater precision than `float`.

```c
/* Fig. 4.6:
   Calculati
#include <st
#include <math.h>

/* functi
int main(
{
   double amount; /* amount on deposit */
   double principal = 1000.0; /* starting principal */
   double rate = .05; /* annual interest rate */
   int year; /* year counter */

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

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

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

      /* output one table row */
      printf( "%4d%21.2f\n", year, amount );
   } /* end for */

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

```
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
```

`<math.h>`

- The header `<math.h>` should be included whenever a math function such as `pow` is used.
- Actually, this program would malfunction without the inclusion of `math.h`, as the linker would be unable to find the `pow` function.
- Function `pow` requires two `double` arguments, but variable `year` is an integer.
- The `math.h` file includes information that tells the compiler to convert the value of `year` to a temporary `double` representation before calling the function.
- This information is contained in something called `pow`'s function prototype.

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

- A field width of `21` specifies that the value printed will appear in `21` print positions.
- The `2` specifies the precision (i.e., the number of decimal positions).
- If the number of characters displayed is less than the field width, then the value will automatically be right justified in the field.
- This is particularly useful for aligning floating-point values with the same precision (so that their decimal points align vertically).
- To left justify a value in a field, place a `-` (minus sign) between the `%` and the field width.
- The minus sign may also be used to left justify integers (such as in `%-6d`) and character strings (such as in `%-8s`).

4.7 `switch` Multiple-Selection Statement

- **Multiple selection**
- 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.
- 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.

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 */
```

```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 ot
         printf( "Incorrect le
         printf( " Enter a new
         break; /* optional; will exit switch anyway */
   } /* end switch */
} /* end while */
```

`(grade = getchar())` executes first

The `getchar` function (from `<stdio.h>`) reads one character from the keyboard

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

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`.

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

`EOF` indicator:
On Linux/UNIX/Mac OS X systems, *`<Ctrl> d`*
On Microsoft Windows, *`<Ctrl> z`*
(You may also need to press *Enter* on Windows.)

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.

```c
   /* 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.)

4.7 switch Multiple-Selection Statement (Cont.)

- Characters are usually represented as one-byte integers in the computer.
- 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' );
```

result is

???

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

- Characters are usually represented as one-byte integers in the computer.
- 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' );
```

  result is

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

  (The integer 97 is the character's numerical representation in the computer.)

- Characters can be read with `scanf` by using the conversion specifier `%c`.

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

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

```
If a = 1, b = 2 , c = 3
What is the result of following ???

a = b = c = 0;
```

4.7 `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).

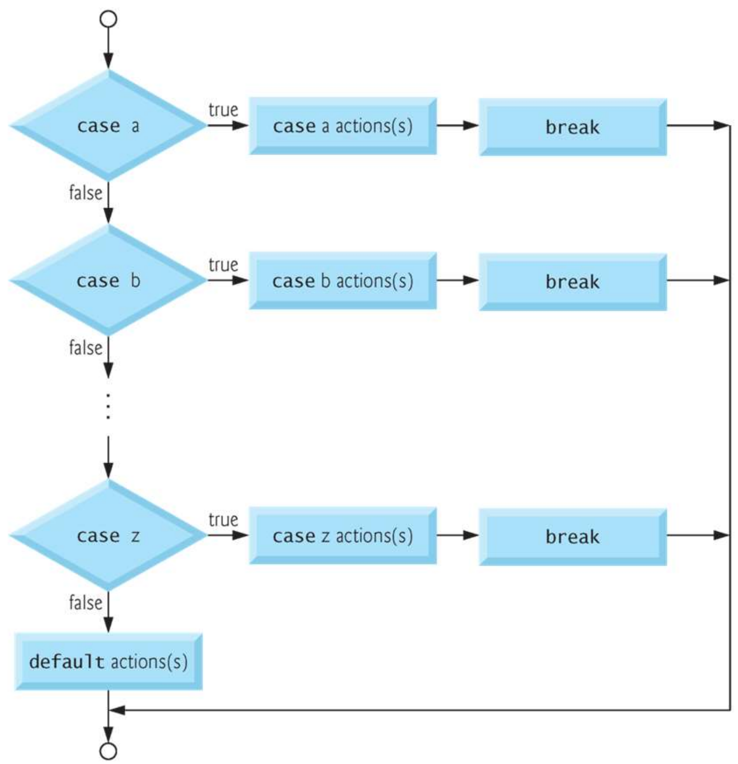

Fig. 4.8 | `switch` multiple-selection statement with `break`s.

4.7 `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.
- To have the program read the characters, they must be sent to the computer by pressing the *Enter* key.

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

- Listing several case labels together (such as `case 'D': case 'd':` in Fig. 4.7) simply means that the same set of actions is to occur for either of these cases.
- When using the `switch` statement, remember that each individual `case` can test only a constant integral expression
  - —i.e., any combination of character constants and integer constants that evaluates to a constant integer value.
- A character constant is represented as the specific character in single quotes, such as `'A'`.
  - characters in double quotes are recognized as strings.

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

- Notes on integral types
  - 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 integers.
  - The range of values for each type depends on the particular computer's hardware.
  - In addition to `int` and `char`, C provides types `short` (an abbreviation of `short int`) and `long` (an abbreviation of `long int`).
  - C specifies that the minimum range of values for `short` integers is –32768 to +32767.
  - For the vast majority of integer calculations, `long` integers are sufficient.
  - The standard specifies that the minimum range of values for `long` integers is –2147483648 to +2147483647.
  - The standard states that the range of values for an `int` is at least the same as the range for `short` integers and no larger than the range for `long` integers.
  - The data type `signed char` can be used to represent integers in the range –128 to +127 or any of the characters in the computer's character set.

4.8 `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.

use a `do…while` statement to print the numbers from 1 to 10.

```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.

```
1  /* Fig. 4.9: fig04_09.c
2     Using the do/while repetition statement */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     int counter = 1; /* initialize counter */
9
10    do {
11       printf( "%d  ", counter ); /* display counter */
12    } while ( ++counter <= 10 ); /* end do...while */
13
14    return 0; /* indicate program ended successfully */
15 } /* end function main */
```

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

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

```
1  /* Fig. 4.1: fig04_01.c
2     Counter-controlled repetition */
3  #include <stdio.h>
4
5  /* function main begins program execution */
6  int main( void )
7  {
8     int counter = 1; /* initialization */
9
10    while ( counter <= 10 ) { /* repetition condition */
11       printf ( "%d\n", counter ); /* display counter */
12       ++counter; /* increment */
13    } /* end while */
14
15    return 0; /* indicate program ended successfully */
16 } /* end function main */
```

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

What happen if in line 8, we write
counter = 11;

4.8 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.
- 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.

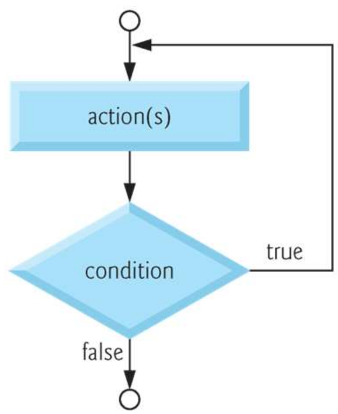

Fig. 4.10 | Flowcharting the `do...while` repetition statement.

- برنامه ای بنویسید که تعدادی عدد از ورودی بخواند و کوچکترین آن ها را چاپ کند.
  - خواندن عدد ها از ورودی تا زمان وارد شدن عدد ۹۹۹۹ ادامه پیدا می کند
  - باید از حلقه do…while استفاده کنید.
- مثال

ورودی:

4 10 3 1 12 9999

خروجی:

1

4.9 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.

When the `if` statement detects that `x` has become `5`, `break` is executed. 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
```

4.9 `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.

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

- the `while` statement could be used in most cases to represent the `for` statement.
- The **one exception** occurs when in the `while` statement

  the increment expression 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`.

Use 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
```

Software Engineering Observation 4.2

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 we'll soon learn, so these programmers do not use* `break` *and* `continue`*.*

Performance Tip 4.1

The `break` *and* `continue` *statements, when used properly, perform faster than the corresponding structured techniques that we'll soon learn.*

4.10 Logical Operators

- logical operators

is 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).

4.10 Logical Operators (Cont.)

- Suppose we wish to ensure that two conditions are **both true**
- In this case, we can use the logical operator **&&** as follows:

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

  - 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 `&&`.
- This condition is true if and only if both of the simple conditions are true.

4.10 Logical Operators (Cont.)

- The `if` statement then considers the combined condition

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

- 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.

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 sets a true value to 1, it accepts any nonzero value as true

4.10 Logical Operators (Cont.)

`||` (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" );:
```

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.

The && operator has a higher precedence than ||. Both operators associate from left to right.

4.10 Logical Operators (Cont.)

Short-circuit evaluation

- 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`).

4.10 Logical Operators (Cont.)

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

expression	!expression
0	1
nonzero	0

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

```
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.

- Also, 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

- There is one type of error that C programmers, That error is accidentally swapping the operators == (equality) and = (assignment).
  - 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 assignments in C produce a value, namely the value that is assigned to the variable on the left side of the assignment operator.

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

- For example, suppose we intend to write

```
if ( payCode == 4 )
   printf( "You get a bonus!" );
```

but we accidentally write

```
if ( payCode = 4 )
   printf( "You get a bonus!" );
```

```
What will be happen?
```

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

- For example, suppose we intend to write

```c
if ( payCode == 4 )
   printf( "You get a bonus!" );
```

  but we accidentally write

```c
if ( payCode = 4 )
   printf( "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 the error—evaluates the assignment expression in the `if` condition.
  - This expression is a simple assignment whose value is the constant 4. Because any nonzero value is interpreted as "true," 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!

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

- Programmers normally write conditions such as `x == 7` with the variable name on the left and the constant on the right.

```
7 == x,
```

- The compiler will treat `7 = x`, as a syntax error, because only a variable name can be placed on the left-hand side of an assignment expression.

4.12 Structured Programming Summary

- You learn control statements
- In flowchart
  - Small circles are used in the figure to indicate the single entry point and the single exit point of each statement.
    - Connecting individual flowchart symbols arbitrarily can lead to unstructured programs.
    - only single-entry/single-exit control statements are used

- Therefore, the programming profession has chosen to combine flowchart symbols to **form a limited set of control statements**, and to build only structured programs by properly combining control statements in two simple ways.

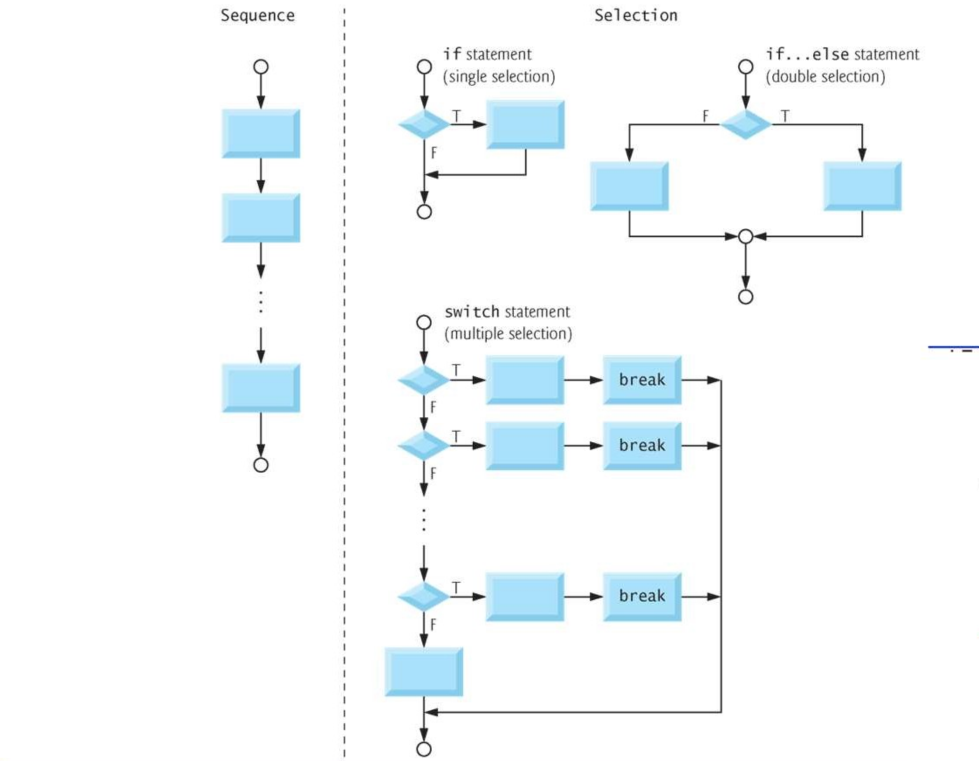

Fig. 4.17 | C's single-entry/single-exit sequence, selection and repetition statements. (Part 1 of 2.)

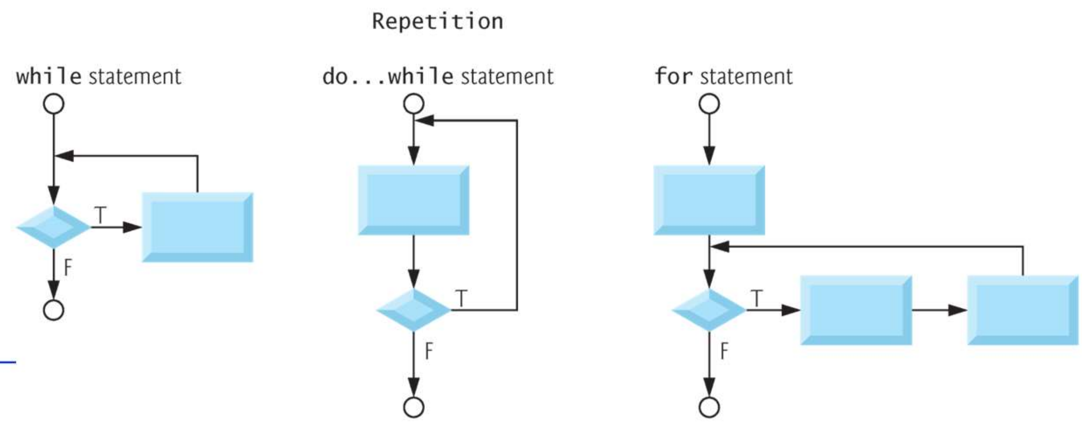

Fig. 4.17 | C's single-entry/single-exit sequence, selection and repetition statements. (Part 2 of 2.)

4.12 Structured Programming Summary (Cont.)

- 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 a program—we have called this “control-statement stacking.”
- The rules for forming structured programs also allow for control statements to be nested.

Rules for Forming Structured Programs
1)Begin with the “simplest flowchart” (Fig. 4.19).
2)Any rectangle (action) can be replaced by two rectangles (actions) in sequence.
3)Any rectangle (action) can be rep’laced 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.

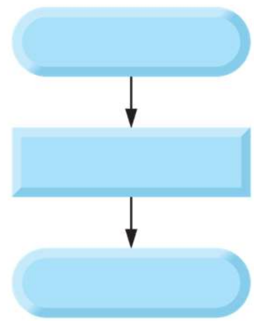

Fig. 4.19 | Simplest flowchart.

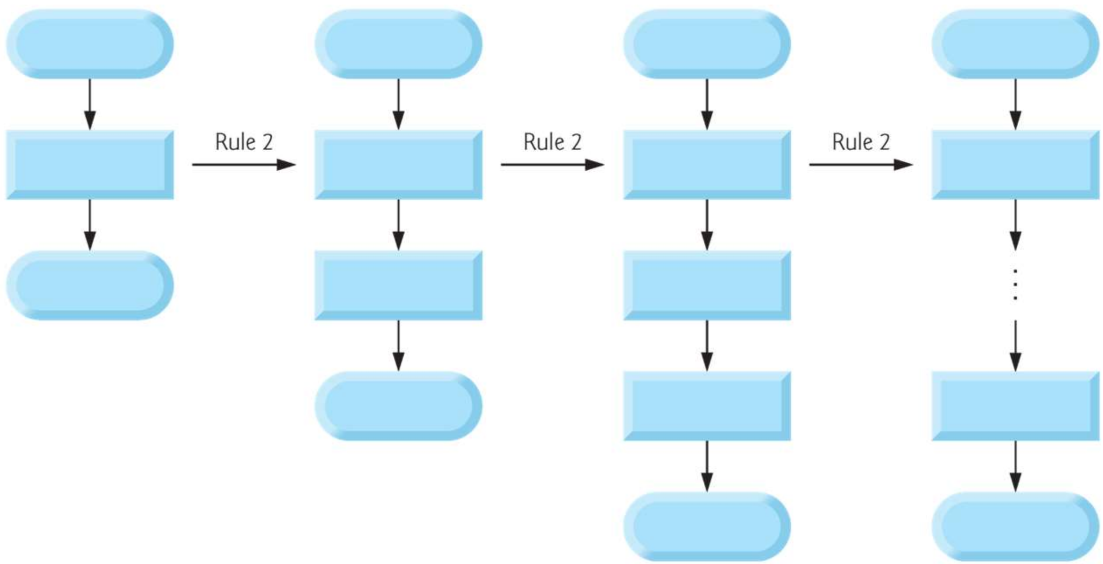

Fig. 4.20 | Repeatedly applying Rule 2 of Fig. 4.18 to the simplest flowchart.

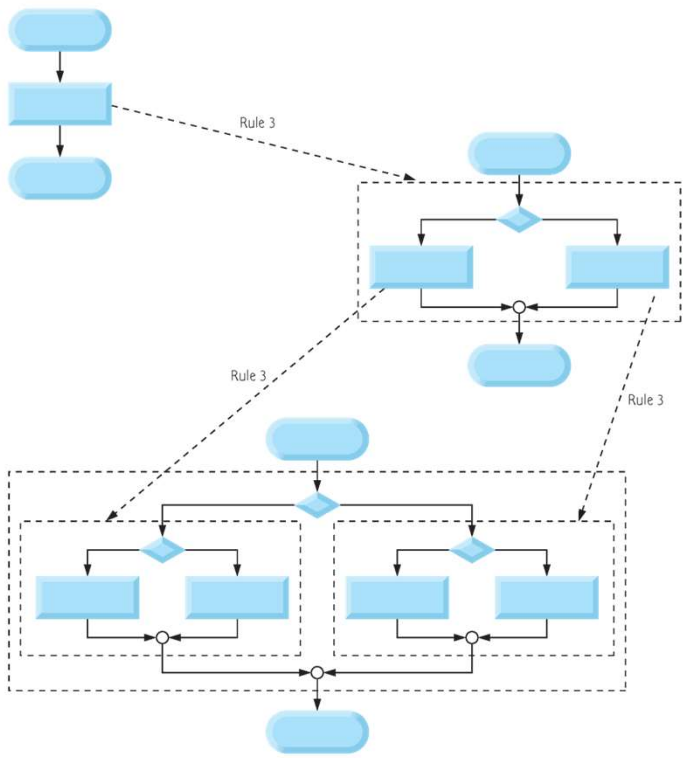

Fig. 4.21 | Applying Rule 3 of Fig. 4.18 to the simplest flowchart.

4.12 Structured Programming Summary (Cont.)

- If the rules in Fig. 4.18 are followed, an unstructured flowchart cannot be created.

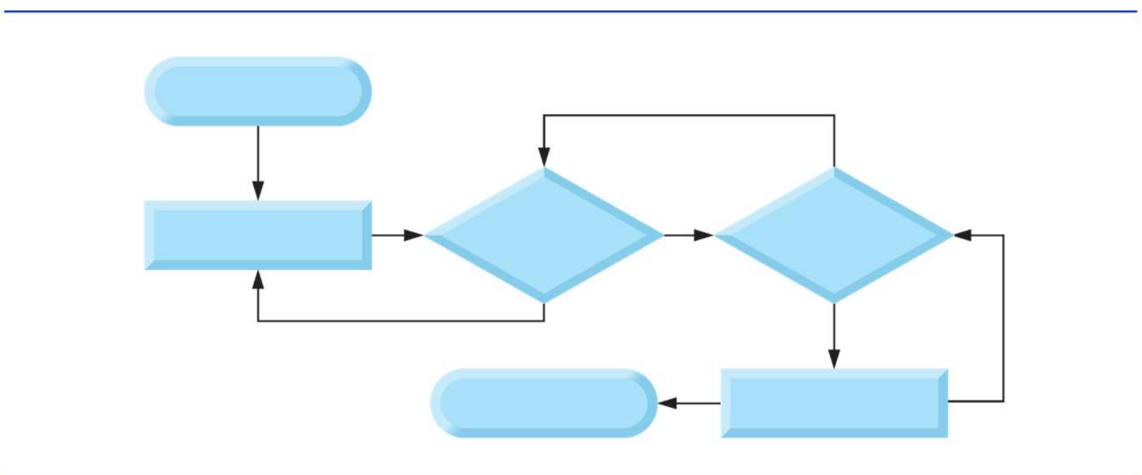

Fig. 4.23 | An unstructured flowchart.