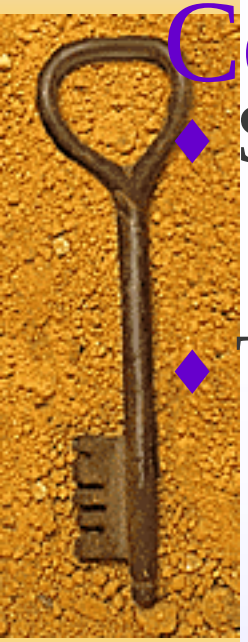


Control Structures

- Sequential execution
  - Statements executed in order
- Transfer of control
  - Next statement executed *not* next one in sequence
- 3 control structures (Bohm and Jacopini)
  - Sequence structure
    - Programs executed sequentially by default
  - Selection structures
    - **`if`**, **`if/else`**, **`switch`**
  - Repetition structures
    - **`while`**, **`do/while`**, **`for`**

Control Structures

- C++ keywords
  - Cannot be used as identifiers or variable names

C++ Keywords				
Keywords common to the C and C++ programming languages				
`auto`	`break`	`case`	`char`	`const`
`continue`	`default`	`do`	`double`	`else`
`enum`	`extern`	`float`	`for`	`goto`
`if`	`int`	`long`	`register`	`return`
`short`	`signed`	`sizeof`	`static`	`struct`
`switch`	`typedef`	`union`	`unsigned`	`void`
`volatile`	`while`			
C++ only keywords				
`asm`	`bool`	`catch`	`class`	`const_cast`
`delete`	`dynamic_cast`	`explicit`	`false`	`friend`
`inline`	`mutable`	`namespace`	`new`	`operator`
`private`	`protected`	`public`	`reinterpret_cast`	
`static_cast`	`template`	`this`	`throw`	`true`
`try`	`typeid`	`typename`	`using`	`virtual`
`wchar_t`				

Control Structures

- Flowchart
  - Graphical representation of an algorithm
  - Special-purpose symbols connected by arrows (flowlines)
  - Rectangle symbol (action symbol)
    - Any type of action
  - Oval symbol
    - Beginning or end of a program, or a section of code (circles)

Control Structures

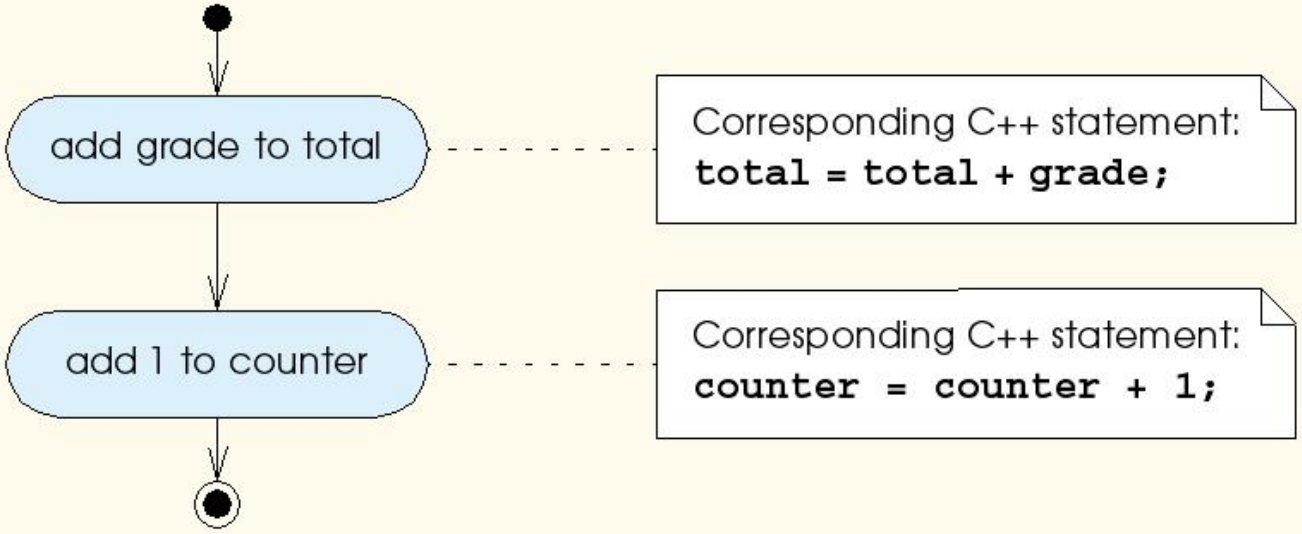


Fig. 2.1 Sequence structure activity diagram.

if Selection Structure

- Selection structure
  - Choose among alternative courses of action
  - Pseudocode example:

    If student's grade is greater than or equal to 60

    Print "Passed"
  - If the condition is **`true`**
    - Print statement executed, program continues to next statement
  - If the condition is **`false`**
    - Print statement ignored, program continues
  - Indenting makes programs easier to read
    - C++ ignores whitespace characters (tabs, spaces, etc.)

if Selection Structure

- Translation into C++

  If student's grade is greater than or equal to 60

  Print "Passed"

```
if ( grade >= 60 )
   cout << "Passed";
```

- Diamond symbol (decision symbol)
  - Indicates decision is to be made
  - Contains an expression that can be true or false
    - Test condition, follow path
- **`if`** structure
  - Single-entry/single-exit

if Selection Structure

- Flowchart of pseudocode statement

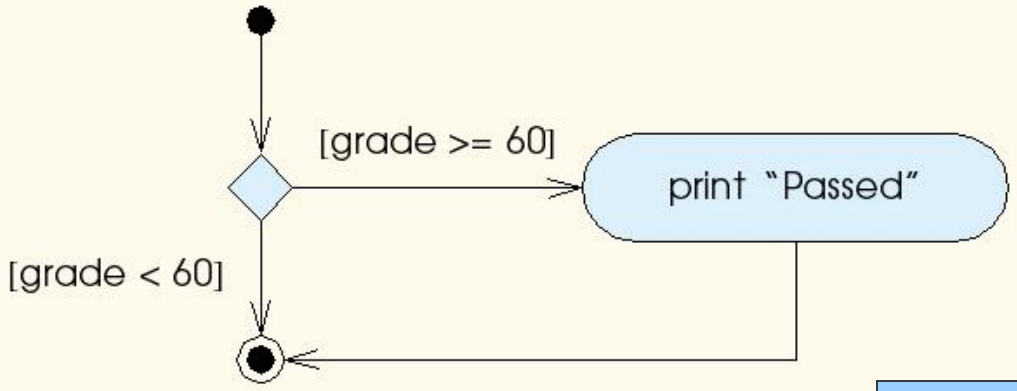


Fig. 2.3 **if** single-selection structure activity diagram.

A decision can be made on any expression.

zero - **false**

nonzero - **true**

Example:

3 - 4 is **true**

`if/else` Selection Structure

- **`if`**
  - Performs action if condition true
- **`if/else`**
  - Different actions if conditions true or false
- Pseudocode

  if student's grade is greater than or equal to 60
  print "Passed"
  else
  print "Failed"

- C++ code

```
if ( grade >= 60 )
   cout << "Passed";
else
   cout << "Failed";
```

`if/else` Selection Structure

- Ternary conditional operator (**`?:`**)
  - Three arguments (condition, value if **`true`**, value if **`false`**)
- Code could be written:

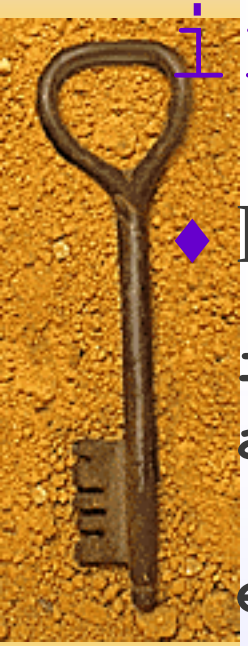
```
cout << ( grade >= 60 ? “Passed” : “Failed” );
```

Condition — Value if true — Value if false

`if/else` Selection Structure

- Nested **`if/else`** structures
  - One inside another, test for multiple cases
  - Once condition met, other statements skipped

```
if student's grade is greater than or equal to 90
    Print "A"
else
    if student's grade is greater than or equal to 80
      Print "B"
    else
      if student's grade is greater than or equal to 70
        Print "C"
      else
        if student's grade is greater than or equal to 60
          Print "D"
        else
            Print "F"
```



if/else Selection Structure

- Example

```
if ( grade >= 90 )          // 90 and
above
   cout << "A";
else if ( grade >= 80 )     // 80-89
   cout << "B";
else if ( grade >= 70 )     // 70-79
   cout << "C";
else if ( grade >= 60 )     // 60-69
   cout << "D";
else                        // less than
60
   cout << "F";
```

`if/else` Selection Structure

- Compound statement
  - Set of statements within a pair of braces

```
if ( grade >= 60 )
   cout << "Passed.\n";
else {
   cout << "Failed.\n";
   cout << "You must take this course
again.\n";
}
```

  - Without braces,

```
cout << "You must take this course
  again.\n";
```

always executed

- Block
  - Set of statements within braces

while Repetition Structure

- Repetition structure
  - Action repeated while some condition remains true
  - Psuedocode

    while there are more items on my shopping list

    Purchase next item and cross it off my list
  - **`while`** loop repeated until condition becomes false
- Example

```
int product = 2;
while ( product <= 1000 )
   product = 2 * product;
```

Formulating Algorithms (Counter-Controlled Repetition)

- Counter-controlled repetition
  - Loop repeated until counter reaches certain value
- Definite repetition
  - Number of repetitions known
- Example

  A class of ten students took a quiz. The grades (integers in the range 0 to 100) for this quiz are available to you. Determine the class average on the quiz.

```cpp
// Fig. 2.7: fig02_07.cpp
// Class average program with counter-controlled repetition.
#include <iostream>
using namespace std;
// function main begins program execution
int main()
{
   int total;       // sum of grades input by user
   int gradeCounter; // number of grade to be entered next
   int grade;       // grade value
   int average;     // average of grades

   // initialization phase
   total = 0;         // initialize total
   gradeCounter = 1;  // initialize loop counter
```

fig02_07.cpp
(1 of 2)

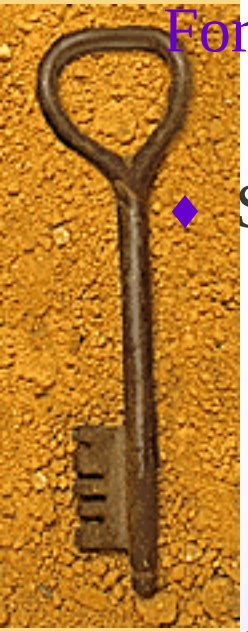
```
   while ( gradeCounter <= 10 ) {      // loop 10 times
      cout << "Enter grade: ";         // prompt for input
      cin >> grade;                    // read grade from user
      total = total + grade;           // add grade to total
      gradeCounter = gradeCounter + 1;  // increment counter
   }
   // termination phase
   average = total / 10;               // integer division
   cout.setf (ios::fixed)
   cout.setf(ios::showpoint);
   cout.precision(2);
   // display result
   cout << "Class average is " << average << endl;
   return 0;   // indicate program ended successfully
   } // end function main
```

The counter gets incremented each time the loop executes. Eventually, the counter causes the loop to end.

fig02_07.cpp
(2 of 2)

fig02_07.cpp
output (1 of 1)

- Enter grade: 98
- Enter grade: 76
- Enter grade: 71
- Enter grade: 87
- Enter grade: 83
- Enter grade: 90
- Enter grade: 57



Formulating Algorithms (Sentinel-Controlled Repetition)

- Suppose problem becomes:

  Develop a class-averaging program that will process an arbitrary number of grades each time the program is run

  - Unknown number of students
  - How will program know when to end?
- Sentinel value
  - Indicates “end of data entry”
  - Loop ends when sentinel input
  - Sentinel chosen so it cannot be confused with regular input
    - -1 in this case

Formulating Algorithms (Sentinel-Controlled Repetition)

- Many programs have three phases
  - Initialization
    - Initializes the program variables
  - Processing
    - Input data, adjusts program variables
  - Termination
    - Calculate and print the final results
  - Helps break up programs for top-down refinement

fig02_09.cpp
(1 of 3)

```
#include <iostream>
#include <iomanip>        // parameterized stream manipulators
using namespace std;
// sets numeric output precision
// function main begins program execution
int main()
{
 int total;        // sum of grades
    int gradeCounter;  // number of grades entered
    int grade;         // grade value

    double average;    // number with decimal point for average

    // initialization phase
    total = 0;         // initialize total
   gradeCounter = 0;  // initialize loop counter
```

Data type **`double`** used to represent decimal numbers.

```
28    // get first grade from user
29    cout << "Enter grade, -1 to end: ";  // prompt for input
30    cin >> grade;                        // read grade from user
31
32    // loop until sentin
33    while ( grade != -1
34       total = total + gr
35       gradeCounter =
36
37       cout << "Enter g
38       cin >> grade;                     // read next grade
39
40    } // end while
41
42    // termination phase
43    // if user entered at least one grade ...
44    if ( gradeCounter != 0 ) {
45     // calculate average of all grades entered
47       average = static_cast< double >( total ) / gradeCounter;
48
```

`static_cast<double>()` treats **`total`** as a **`double`** temporarily (casting).

Required because dividing two integers truncates the remainder.

`gradeCounter` is an **`int`**, but it gets *promoted* to **`double`**.

```
      cout.setf(ios::showpoint);
      cout.precision(2);
50    cout << "Class average is " <<average << endl;
52
53    } // end if part of if/else
54
55    else // if no grades were entered, output appropriate message
56       cout << "No grades w
57
58    return 0;   // indicate pro
59
60 } // end function main
```

fig02_09.cpp
(3 of 3)

fig02_09.cpp
output (1 of 1)

`fixed` forces output to print in fixed point format (not scientific notation). Also, forces trailing zeros and decimal point to print.

Include **`<iostream>`**

`sion(2)` prints two digits past t (rounded to fit precision).

t use this must include **`<iomanip>`**

- Enter grade, -1 to end: 75
- Enter grade, -1 to end: 94
- Enter grade, -1 to end: 97
- Enter grade, -1 to end: 88
- Enter grade, -1 to end: 70
- Enter grade, -1 to end: 64
- Enter grade, -1 to end: 83
- Enter grade, -1 to end: 89
- Enter grade, -1 to end: -1

Nested Control Structures

- Problem statement

  A college has a list of test results (1 = pass, 2 = fail) for 10 students. Write a program that analyzes the results. If more than 8 students pass, print "Raise Tuition".

- Notice that
  - Program processes 10 results
    - Fixed number, use counter-controlled loop
  - Two counters can be used
    - One counts number that passed
    - Another counts number that fail
  - Each test result is 1 or 2
    - If not 1, assume 2

fig02_11.cpp
(1 of 2)

```
3  #include <iostream>
4  using namespace std;
5  // function main begins program execution
10 int main()
11 {
12    // initialize variables in declarations
13    int passes = 0;          // number of passes
14    int failures = 0;        // number of failures
15    int studentCounter = 1;  // student counter
16    int result;              // one exam result
17
18    // process 10 students using counter-controlled loop
19    while ( studentCounter <= 10 ) {
20
21       // prompt user for input and obtain value from user
22       cout << "Enter result (1 = pass, 2 = fail): ";
23       cin >> result;
24
```

fig02_11.cpp
(2 of 2)

```
27          passes = passes + 1;
28
29       else  // if result not 1, increment failures
30          failures = failures + 1;
31
32       // increment studentCounter so loop eventually terminates
33       studentCounter = studentCounter + 1;
34
35    } // end while
36
37    // termination phase; display number of passes and failures
38    cout << "Passed " << passes << endl;
39    cout << "Failed " << failures << endl;
40
41    // if more than eight students passed, print "raise tuition"
42    if ( passes > 8 )
43       cout << "Raise tuition " << endl;
44
45    return 0;   // successful termination
46
47 } // end function main
```

- Enter result (1 = pass, 2 = fail): 2
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 2
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 2
- Passed 6
- Failed 4

- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 2
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Enter result (1 = pass, 2 = fail): 1
- Passed 9
- Failed 1
- Raise tuition

Assignment Operators

- Assignment expression abbreviations
  - Addition assignment operator

    `c = c + 3;` abbreviated to

    `c += 3;`

- Statements of the form

  ```
  variable = variable operator
   expression;
  ```

  can be rewritten as

  ```
  variable operator= expression;
  ```

- Other assignment operators

  ```
  d -= 4      (d = d - 4)
  e *= 5      (e = e * 5)
  f /= 3      (f = f / 3)
  g %= 9      (g = g % 9)
  ```

Increment and Decrement Operators

- Increment operator (**++**) - can be used instead of **`c += 1`**
- Decrement operator (**--**) - can be used instead of **`c -= 1`**
  - Preincrement
    - When the operator is used before the variable (**`++c`** or **`–c`**)
    - Variable is changed, then the expression it is in is evaluated.
  - Posincrement
    - When the operator is used after the variable (**`c++`** or **`c--`**)
    - Expression the variable is in executes, then the variable is changed.

Increment and Decrement Operators

- Increment operator (`++`)
  - Increment variable by one
  - `c++`
    - Same as `c += 1`
- Decrement operator (`--`) similar
  - Decrement variable by one
  - `c--`

Increment and Decrement Operators

- Preincrement
  - Variable changed before used in expression
    - Operator before variable (`++c` or `--c`)
- Postincrement
  - Incremented changed after expression
    - Operator after variable (`c++, c--`)

Essentials of Counter-Controlled Repetition

- Counter-controlled repetition requires
  - Name of control variable/loop counter
  - Initial value of control variable
  - Condition to test for final value
  - Increment/decrement to modify control variable when looping

```
3  #include <iostream>
4  using namespace std;
5  // function main begins program execution
9  int main()
10 {
11    int counter = 1;           // initialization
12
13    while ( counter <= 10 ) {    // repetition condition
14       cout << counter << endl;  // display counter
15       ++counter;                // increment
16
17    } // end while
18
19    return 0;   // indicate successful termination
20
21 } // end function main
```

fig02_16.cpp
(1 of 1)

for Repetition Structure

- General format when using **`for`** loops

```
for ( initialization; LoopContinuationTest;
                increment )
    statement
```

- Example

```
for( int counter = 1; counter <= 10;
  counter++ )
   cout << counter << endl;
```

– Prints integers from one to ten

No semicolon after last statement

```
1  // Fig. 2.17: fig02_17.cpp
2  // Counter-controlled repetition with the for structure.
3  #include <iostream>
4  using namespace std;
5  // function main begins program execution
9  int main()
10 {
11    // Initialization, repetition condition and incrementing
12    // are all included in the for structure header.
13
14    for ( int counter = 1; counter <= 10; counter++ )
15       cout << counter << endl;
16
17    return 0;   // indicate successful termination
18
19 } // end function main
```

fig02_17.cpp
(1 of 1)

for Repetition Structure

- **`for`** loops can usually be rewritten as **`while`** loops

```
initialization;
while ( loopContinuationTest){
   statement
   increment;
}
```

- Initialization and increment
  - For multiple variables, use comma-separated lists

```
for (int i = 0, j = 0;  j + i <= 10;
  j++, i++)
    cout << j + i << endl;
```

```
1   // Fig. 2.20: fig02_20.cpp
2   // Summation with for.
3   #include <iostream>
4   using namespace std;
    // function main begins program execution
9   int main()
10  {
11     int sum = 0;                  // initialize sum
12
13     // sum even integers from 2 through 100
14     for ( int number = 2; number <= 100; number += 2 )
15        sum += number;             // add number to sum
16
17     cout << "Sum is " << sum << endl;  // output sum
18     return 0;                     // successful termination
19
20  } // end function main
```

```
Sum is 2550
```

fig02_20.cpp
(1 of 1)

fig02_20.cpp
output (1 of 1)

Examples Using the for Structure

- Program to calculate compound interest
- *A person invests $1000.00 in a savings account yielding 5 percent 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:*

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

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

```
1   // Fig. 2.21: fig02_21.cpp
2   // Calculating compound interest.
3   #include <iostream>
    #include <iomanip>
11  using namespace std;
12  using std::setw;
13  using std::setprecision;
14
15  #include <cmath>  // enables p
16
17  // function main begins program execution
18  int main()
19  {
20     double amount;           // amount on deposit
21     double principal = 1000.0;  // starting principal
22     double rate = .05;       // interest rate
23
```

<cmath> header needed for the **pow** function (program will not compile without it).

fig02_21.cpp
(1 of 2)

```
24    // output table column heads
25    cout << "Year" << setw( 21 ) << "Amount on deposit" << endl;
26
27    // set floating-point number format
28    cout << fixed << setprecision( 2 );
29
30    // calculate amount on deposit for each of ten ye
31    for ( int year = 1; year <= 10; year++ ) {
32
33       // calculate new amount for specified year
34       amount = principal * pow( 1.0 + rate, year );
35
36       // output one table row
37       cout << setw( 4 ) << year
38            << setw( 21 ) << amount << endl;
39
40    } // end for
41
42    return 0;   // indicate successful termination
43
44 } // end function main
```

Sets the field width to at least 21 characters. If output less than 21, it is right-justified.

`pow(x,y)` = x raised to the yth power.

switch Multiple-Selection Structure

Test variable for multiple values

– Series of **`case`** labels and optional **`default`** case

```
switch ( variable ) {
    case value1:          // taken if variable == value1
    statements
    break;                      // necessary to exit switch

    case value2:
    case value3:     // taken if variable == value2 or == value3
    statements
    break;

    default:          // taken if variable matches no other cases
    statements
     break;
}
```

do/while Repetition Structure

- Similar to **`while`** structure
  - Makes loop continuation test at end, not beginning
  - Loop body executes at least once
- Format

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

```
1  // Fig. 2.24: fig02_24.cpp
2  // Using the do/while repetition structure.
3  #include <iostream>
4  using namespace std;
8  // function main begins program execution
9  int main()
10 {
11    int counter = 1;           // initialize counter
12
13    do {
14       cout << counter << "  ";   // display counter
15    } while ( ++counter <= 10 );  // end do/while
16
17    cout << endl;
18
19    return 0;   // indicate successful termination
```

Notice the preincrement in loop-continuation test.

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

fig02_24.cpp
(1 of 1)

fig02_24.cpp
output (1 of 1)

break and continue Statements

- **`break`** statement
  - Immediate exit from **`while`**, **`for`**, **`do/while`**, **`switch`**
  - Program continues with first statement after structure
- Common uses
  - Escape early from a loop
  - Skip the remainder of **`switch`**

```
2  // Using the break statement in a for structure.
3  #include <iostream>

// function main begins program execution
9  int main()
10 {
11
12    int x;  // x declared here so it can be used after the loop
13
14    // loop 10 times
15    for ( x = 1; x <= 10; x++ ) {
16
17       // if x is 5, terminate loop
18       if ( x == 5 )
19          break;          // break loop only if x is 5
20
21       cout << x << " ";   // display value of x
22
23    } // end for
24
25    cout << "\nBroke out of loop when x became " << x << endl;
```

Exits **`for`** structure when **`break`** executed.

fig02_26.cpp
(1 of 2)

```
26
27       return 0;   // indicate successful termination
28
29    } // end function main
```

```
1 2 3 4
Broke out of loop when x became 5
```

fig02_26.cpp
(2 of 2)

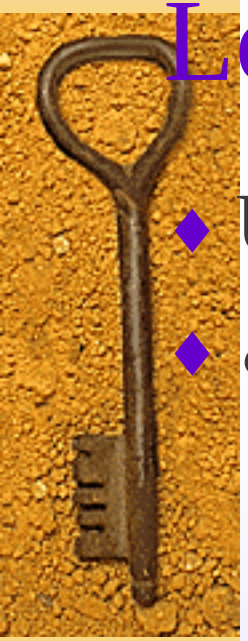
fig02_26.cpp
output (1 of 1)

break and continue Statements

- **`continue`** statement
  - Used in **`while`**, **`for`**, **`do/while`**
  - Skips remainder of loop body
  - Proceeds with next iteration of loop
- **`while`** and **`do/while`** structure
  - Loop-continuation test evaluated immediately after the **`continue`** statement
- **`for`** structure
  - Increment expression executed
  - Next, loop-continuation test evaluated

```
   3  #include <iostream>
      using namespace std;
   8  // function main begins program execution
   9  int main()
  10  {
  11     // loop 10 times
  12     for ( int x = 1; x <= 10; x++ ) {
  13
  14        // if x is 5, continue with next iterati
  15        if ( x == 5 )
  16           continue;       // skip remaining code in loop body
  17
  18        cout << x << " ";   // display value of x
  19
  20     } // end for structure
  21
  22     cout << "\nUsed continue to skip printing the value 5"
  23          << endl;
  24
  25     return 0;              // indicate successful termination
  26
  27  } // end function main
```

Skips to next iteration of the loop.



Logical Operators

- Used as conditions in loops, if statements
- **`&&`** (logical **AND**)
  - **`true`** if both conditions are **`true`**

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

- **`||`** (logical **OR**)
  - **`true`** if either of condition is **`true`**

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

Logical Operators

- **!** (logical **NOT**, logical negation)
  - Returns **true** when its condition is **false**, & vice versa

```
if ( !( grade == sentinelValue ) )
   cout << "The next grade is " << grade
<< endl;
```

Alternative:

```
if ( grade != sentinelValue )
   cout << "The next grade is " << grade
<< endl;
```