Lecture 10:
Control Statements (cont)

Ioan Raicu
Department of Electrical Engineering & Computer Science
Northwestern University

EECS 211
Fundamentals of Computer Programming II
April 13th, 2010

5.9 break and continue Statements

- The break statement, when executed in a `while`, `for`, `do`…`while` or `switch` statement, causes 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.

5.9 break and continue Statements

```cpp
// Fig. 5.13: fig05_13.cpp
// break statement exiting a for statement.
#include <iostream>
using namespace std;

int main()
{
   int count; // control variable also used after loop terminates

   for ( count = 1; count <= 10; count++ ) // loop 10 times
   {
      if ( count == 5 )
         break; // break loop only if x is 5

      cout << count << " ";
   } // end for

   cout << "\nBroke out of loop at count = " << count << endl;
} // end main
```

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

Fig. 5.13 | break statement exiting a for statement.

5.9 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 statement and proceeds with the next iteration of the loop.
- In while and do…while statements, the loop-continuation test evaluates immediately after the continue statement executes.
- In the for statement, the increment expression executes, then the loop-continuation test evaluates.

5.9 break and continue Statements

```cpp
// Fig. 5.14: fig05_14.cpp
// continue statement terminating an iteration of a for statement.
#include <iostream>
using namespace std;

int main()
{
   for ( int count = 1; count <= 10; count++ ) // loop 10 times
   {
      if ( count == 5 ) // if count is 5,
         continue;      // skip remaining code in loop

      cout << count << " ";
   } // end for

   cout << "\nUsed continue to skip printing 5" << endl;
} // end main
```

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

Fig. 5.14 | continue statement terminating a single iteration of a for statement.

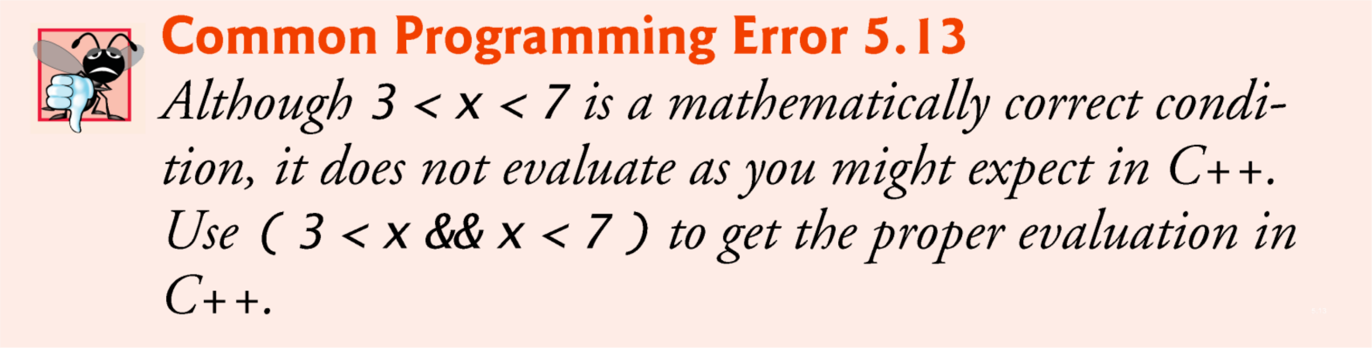
5.10 Logical Operators

- C++ provides logical operators that are 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).

5.10 Logical Operators (cont.)

- The && (logical AND) operator is used to ensure that two conditions are *both* ***true*** *before we choose a certain path of execution*.
- The simple condition to the left of the && operator evaluates first.
- If necessary, the simple condition to the right of the && operator evaluates next.
- The right side of a logical AND expression is evaluated only if the left side is `true`.

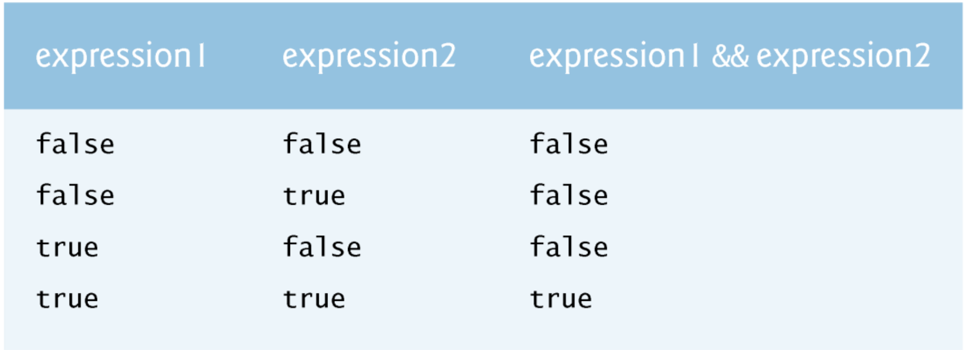
5.10 Logical Operators (cont.)



Common Programming Error 5.13

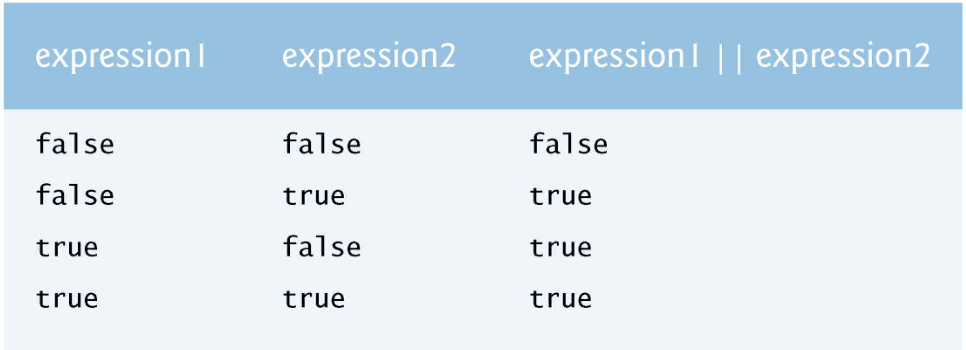
Although 3 < x < 7 is a mathematically correct condition, it does not evaluate as you might expect in C++. Use (3 < x && x < 7) to get the proper evaluation in C++.

5.10 Logical Operators (cont.)



expression1	expression2	expression1 && expression2
false	false	false
false	true	false
true	false	false
true	true	true

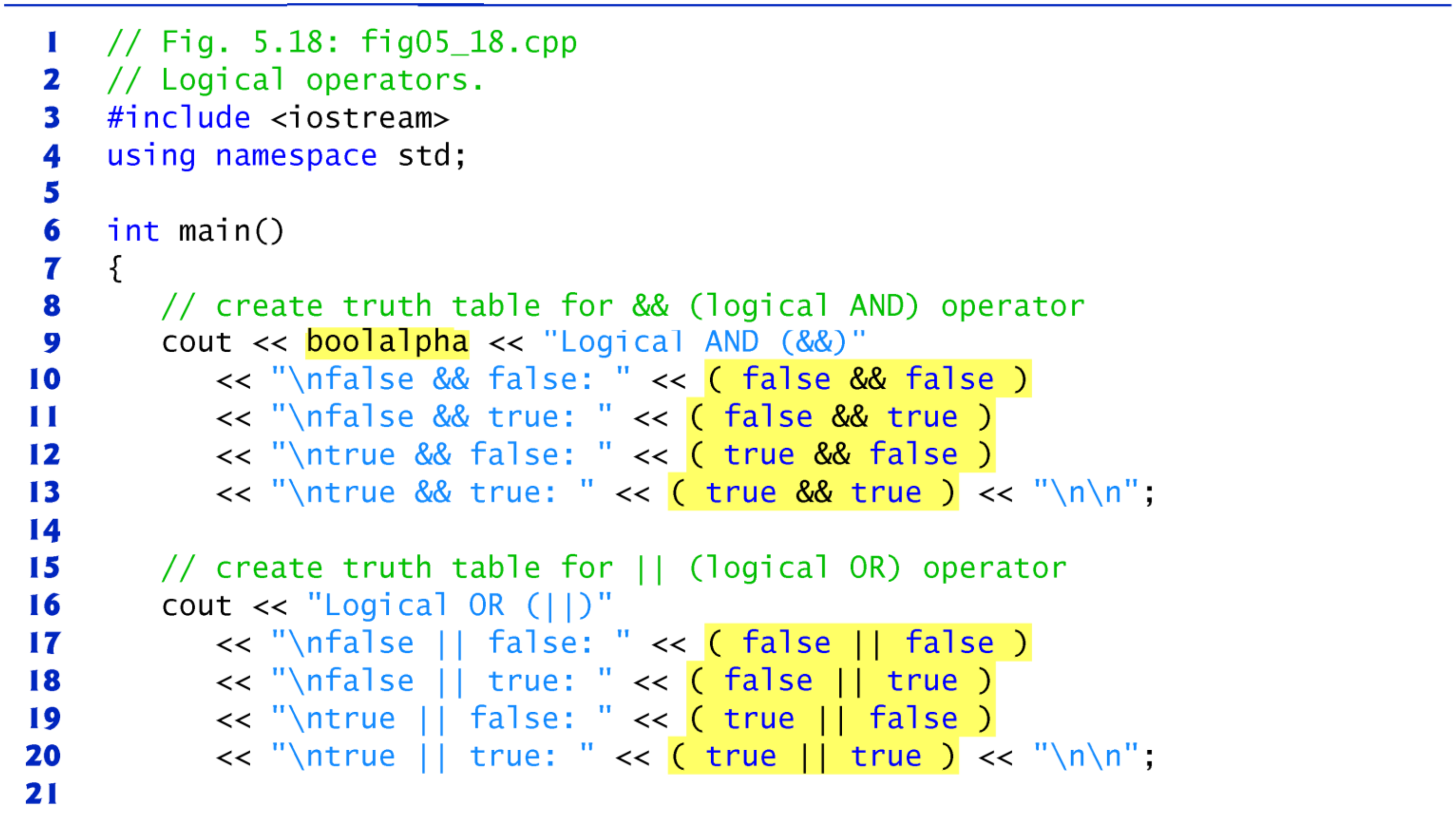
Fig. 5.15 | && (logical AND) operator truth table.



expression1	expression2	expression1 || expression2
false	false	false
false	true	true
true	false	true
true	true	true

Fig. 5.16 | || (logical OR) operator truth table.

5.10 Logical Operators (cont.)



```cpp
// Fig. 5.18: fig05_18.cpp
// Logical operators.
#include <iostream>
using namespace std;

int main()
{
   // create truth table for && (logical AND) operator
   cout << boolalpha << "Logical AND (&&)"
      << "\nfalse && false: " << ( false && false )
      << "\nfalse && true: " << ( false && true )
      << "\ntrue && false: " << ( true && false )
      << "\ntrue && true: " << ( true && true ) << "\n\n";

   // create truth table for || (logical OR) operator
   cout << "Logical OR (||)"
      << "\nfalse || false: " << ( false || false )
      << "\nfalse || true: " << ( false || true )
      << "\ntrue || false: " << ( true || false )
      << "\ntrue || true: " << ( true || true ) << "\n\n";

```

Fig. 5.18 | Logical operators.

5.10 Logical Operators (cont.)

- C++ provides the ! (logical NOT, also called logical negation) operator to “reverse” a condition’s meaning.
- The unary logical negation operator has only a single condition as an operand.
- You can often avoid the ! operator by using an appropriate relational or equality operator.
- Figure 5.17 is a truth table for the logical negation operator (!).

5.10 Logical Operators (cont.)

```
   // create truth table for ! (logical negation) operator
   cout << "Logical NOT (!)"
      << "\n!false: " << ( !false )
      << "\n!true: " << ( !true ) << endl;
} // end main
```

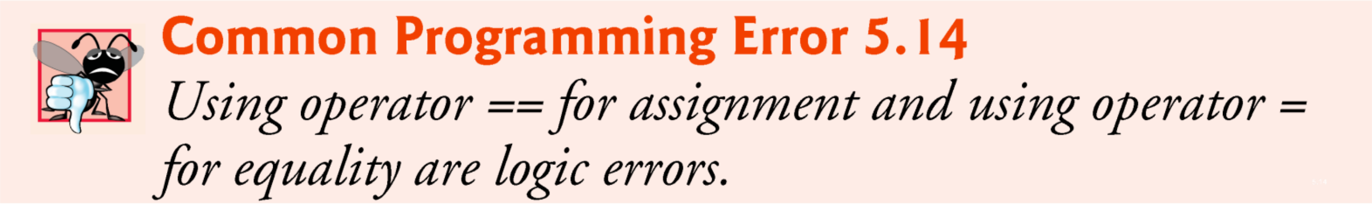
```
Logical AND (&&)
false && false: false
false && true: false
true && false: false
true && true: true

Logical OR (||)
false || false: false
false || true: true
true || false: true
true || true: true

Logical NOT (!)
!false: true
!true: false
```

Fig. 5.18 | Logical operators. (Part 3 of 3.)

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

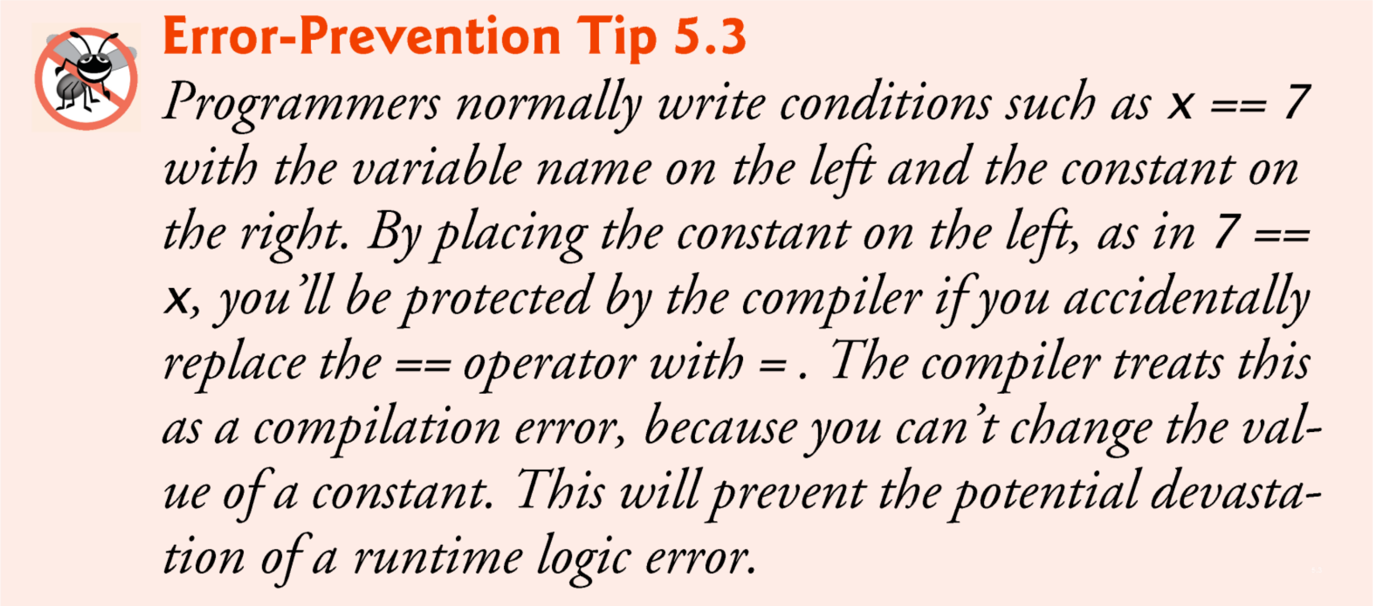
- Accidentally swapping the operators == (equality) and = (assignment).
- Damaging because they ordinarily do not cause syntax errors.
- Rather, statements with these errors tend to compile correctly and the programs run to completion, often generating incorrect results through runtime logic errors.
- [*Note: Some compilers issue a warning when = is used in a context where == typically is expected.]*
- Two aspects of C++ contribute to these problems.
  - One is that *any expression that produces a value can be used in the decision portion of any control statement.*
  - The second is that assignments produce a value—namely, the value assigned to the variable on the left side of the assignment operator.
- *Any nonzero value is interpreted as* ***`true`***

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



Common Programming Error 5.14

Using operator == for assignment and using operator = for equality are logic errors.



Error-Prevention Tip 5.3

Programmers normally write conditions such as `x == 7` *with the variable name on the left and the constant on the right. By placing the constant on the left, as in* `7 == x`*, you'll be protected by the compiler if you accidentally replace the == operator with = . The compiler treats this as a compilation error, because you can't change the value of a constant. This will prevent the potential devastation of a runtime logic error.*

5.11 Confusing the Equality (==) and Assignment (=) Operators (cont.)

- 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.*

Questions

?