Outline

- In this lesson, we will:
  - See the need for asking if more than one condition is satisfied
    - The unit pulse function
  - Describe the binary logical AND and OR operators
  - Introduce truth tables
  - Describe chaining numerous logical expressions
  - Describe short-circuit evaluation
  - Describe the unary logical negation (NOT)

Background

- We have seen six comparison operators
  
<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>not equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

- Problem:
  - What if more than one condition is required?
  - What if two conditions result in the same consequent?
  - What if we require that a condition must be false?

The unit pulse

- Suppose we want to implement the function:

\[
\text{unit}(x) = \begin{cases} 
0 & x < 0 \\
\frac{1}{2} & x = 0 \\
1 & x > 0 \text{ and } x < 1 \\
\frac{1}{2} & x = 1 \\
0 & x > 1 
\end{cases}
\]

- This function has an integral (area under the curve) equal to 1
The unit pulse

- We could implement this program as follows:

```cpp
#include <iostream>

// Function declaration
int main();

// Function definition
int main() { double x;
    std::cin << "Enter a number: ";
    std::cin >> x;
    if (x < 0.0) {
        std::cout << 0.0 << std::endl;
    } else if (x == 0.0) {
        std::cout << 0.0 << std::endl;
    } else if (x > 0.0) {
        if (x < 0.5) {
            std::cout << 0.0 << std::endl;
        } else if (x == 0.5) {
            std::cout << 0.5 << std::endl;
        } else if (x > 0.5) {
            std::cout << 1.0 << std::endl;
        } else if (x == 1.0) {
            std::cout << 1.0 << std::endl;
        } else {
            std::cout << 0.0 << std::endl;
        }
    }
    return 0;
}
```

- Alternatively, could we swap the first consequent block and the alternative?

```cpp
if (condition-1) {
    std::cout << 1.0 << std::endl;
    if (x > 0.0) {
        std::cout << 1.0 << std::endl;
        if (x < 0.0) {
            std::cout << 0.0 << std::endl;
        } else if (x == 0.0) {
            std::cout << 0.0 << std::endl;
        } else if (x > 0.0) {
            if (x < 0.5) {
                std::cout << 0.0 << std::endl;
            } else if (x == 0.5) {
                std::cout << 0.5 << std::endl;
            } else if (x > 0.5) {
                std::cout << 1.0 << std::endl;
            } else if (x == 1.0) {
                std::cout << 1.0 << std::endl;
            } else {
                std::cout << 0.0 << std::endl;
            }
        }
    } else if (condition-2) {
        std::cout << 0.5 << std::endl;
    } else {
        std::cout << 0.0 << std::endl;
    }
}
```

- In English, we would simply say that we should print
  1 if both \( x > 0 \) AND \( x < 1 \)
  ½ if either \( x = 0 \) OR \( x = 1 \)
  0 otherwise

The unit pulse

- Can we implement this cascading conditional statement using only two conditions?

```cpp
if (condition-1) {
    std::cout << 0.0 << std::endl;
} else if (condition-2) {
    unit(x) = (x > 0.0) ? (x < 1.0) ? 1 : 0 : 1;
} else {
    std::cout << 1.0 << std::endl;
}
```

- In English, we would simply say that we should print
  0 if either \( x < 0 \) OR \( x > 1 \)
  ½ if either \( x = 0 \) OR \( x = 1 \)
  1 otherwise

Logical operators

- In C++, there are two binary logical operators:
  - The OR operator \( \lor \) returns true if either operand is true
  - The AND operator \( \land \) returns true if both operands are true

<table>
<thead>
<tr>
<th>Consequent</th>
<th>Conditions</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.0</td>
<td>( x &lt; 0 ) OR ( x &gt; 1 )</td>
<td>( x &lt; 0.0 ) OR ( x &gt; 1.0 )</td>
</tr>
<tr>
<td>0.5</td>
<td>( x = 0 ) OR ( x = 1 )</td>
<td>( x = 0.0 ) OR ( x = 1.0 )</td>
</tr>
<tr>
<td>1.0</td>
<td>( x &gt; 0 ) AND ( x &lt; 1 )</td>
<td>( x &gt; 0.0 ) AND ( x &lt; 1.0 )</td>
</tr>
</tbody>
</table>
Logical operators

- Thus, we may implement this as follows:

```cpp
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    double x[];
    double y[];
    double z[];

    std::cout << "Enter a value 'x': ";
    std::cin >> x;

    std::cout << "Enter a value 'y': ";
    std::cin >> y;

    std::cout << "Enter a value 'z': ";
    std::cin >> z;
}
```

Maximum of three

- We can now implement a maximum of three values
  - Given $x, y$ and $z$,
    - If $x \geq y$ and $x \geq z$, max($x, y, z$) = $x$
    - Otherwise, if $y \geq z$, max($x, y, z$) = $y$
    - Otherwise, max($x, y, z$) = $z$

- We could also describe this as:
  - Given $x, y$ and $z$,
    - If $x > y$ and $x > z$, max($x, y, z$) = $x$
    - Otherwise, if $y > z$, max($x, y, z$) = $y$
    - Otherwise, max($x, y, z$) = $z$

- Both are correct, but the first gets us, in some cases, to our answer quicker.
Logical operators

Truth tables

• The logical OR operator | is true if either operand is true
  – It is false if both operands are false

• The logical AND operator && is true if both operands are true
  – It is false if either operand is false

• To display this visually, we use a truth table

Truth tables

• In elementary school, you saw addition and multiplication tables:
  – Given two operands, the table gave the result of the operation

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>12</td>
<td>14</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
<td>18</td>
<td>21</td>
<td>24</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
<td>20</td>
<td>24</td>
<td>28</td>
<td>32</td>
<td>36</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
<td>30</td>
<td>35</td>
<td>40</td>
<td>45</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>6</td>
<td>12</td>
<td>18</td>
<td>24</td>
<td>30</td>
<td>36</td>
<td>42</td>
<td>48</td>
<td>54</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>7</td>
<td>14</td>
<td>21</td>
<td>28</td>
<td>35</td>
<td>42</td>
<td>49</td>
<td>56</td>
<td>63</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>8</td>
<td>16</td>
<td>24</td>
<td>32</td>
<td>40</td>
<td>48</td>
<td>56</td>
<td>64</td>
<td>72</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>9</td>
<td>18</td>
<td>27</td>
<td>36</td>
<td>45</td>
<td>54</td>
<td>63</td>
<td>72</td>
<td>81</td>
</tr>
</tbody>
</table>

Truth tables

• An alternate form is to consider all values of the operands:

| x  | y  | x && y | x || y |
|----|----|--------|--------|
| true | true | true   | true   |
| true | false| false  | true   |
| false| false| false  | false  |
| false| true | false  | true   |

| x  | y  | x && y | x || y |
|----|----|--------|--------|
| true | true | true   | true   |
| true | false| false  | true   |
| false| false| false  | false  |
| false| true | false  | true   |
Logical operators

Logical expressions

- We have seen that the condition of a logical statement may be:
  1. A comparison operation, or
  2. Two comparison operations joined by && or ||

- More generally, a logical expression may be:
  1. The Boolean literals true or false,
  2. A local variable of type bool,
  3. A comparison operation, or
  4. Two logical expressions joined by && or ||

For example, our program may have:

```c
bool is_valid = true;
bool is_found = false;
double x;
double y;
double z;
// Do something...
if (is_valid || (((x > 3) && (x < 12.5)) || (y < z) && is_found)) {
    // Do something specific to this condition
}
```

- In general, it will never be this complicated, but it is just like an arithmetic expression:
  
  \[
  u = a((b + c)d + e);
  \]

Logical operators

Logical expressions

- If you are mixing such conditions, use parentheses to be clear:
  ```c
  if ( ((x > 0.0) && (y > 0.0)) && (y < 0.0) ) {
    // Do something specific to this condition
  }
  ```

- There is an order-of-operations for logical operations, but
  - Most people don't intuitively remember them
  - You may get it wrong...

- Please, just use parentheses always when mixing || and &&
Multiple conditions

- For example, consider:
  \[(x == 0) || (x <= 2) && (x >= 1)\]

- Does this mean:
  \[(x == 0) || ((x <= 2) && (x >= 1))\]
  or
  \[((x == 0) || (x <= 2)) && (x >= 1)\]

  - The first is true if \(x\) is 0 or \(x\) is in the closed interval \([1, 2]\)
  - The second is true only if \(x\) is in the closed interval \([1, 2]\)

Logical expressions

```cpp
int main() {
    bool has_fever();
    bool has_dry_cough();
    bool is_tired();
    bool has_serious_symptom();

    std::cout << "Do you have a fever?" << std::endl;
    std::cout << "Enter 1 (yes) or 0 (no): ";
    std::cin >> has_fever;

    std::cout << "Do you have a dry cough?" << std::endl;
    std::cout << "Enter 1 (yes) or 0 (no): ";
    std::cin >> has_dry_cough;

    std::cout << "Are you more tired than usual?" << std::endl;
    std::cout << "Enter 1 (yes) or 0 (no): ";
    std::cin >> is_tired;
}
```

Logical operators

```cpp
std::cout << "Do you have any of:" << std::endl;
std::cout << "difficulty breathing," << std::endl;
std::cout << "chest pains, or" << std::endl;
std::cout << "loss of speech or movement?" << std::endl;
std::cout << "Enter 1 (yes) or 0 (no): ";
std::cin >> has_serious_symptom;

if (has_serious_symptom || (has_fever && has_dry_cough && is_tired)) {
    std::cout << "Get medical help now." << std::endl;
} else if (has_fever || has_dry_cough || is_tired) {
    std::cout << "Please self-isolate for two weeks and"
    << "seek medical help if the symptoms get worse." << std::endl;
} else {
    std::cout << "You can go out, but wear a mask." << std::endl;
}
return 0;
```

Short-circuit evaluation

- C++ produces code that does the minimum work necessary:
  - Suppose you wonder: Is the speaker taller than 6' and stupid?
    - I tell you I'm 180 cm
  - Instead, you may wonder: Does the speaker drink coffee or drink tea?
    - I tell you I drink coffee
Short-circuit evaluation

- Consider these logical expressions:
  \[(x < -10) \lor (x > 10)\]
  \[(x < -10) \lor ((x > -1) \land (x < 1)) \lor (x > 10)\]

- Suppose that \(x\) has the value -100
  - The first comparison operation returns true
  - Is there any reason to even bother testing the others?
    - No: the result of true \(\lor\) any-other-conditions must be true
  - This is referred to as short-circuit evaluation

- Similarly, consider
  \[(x > -10) \land (x < 10)\]
  \[(x > -10) \land ((x < -1) \lor (x > 1)) \land (x < 10)\]

- Suppose that \(x\) has the value -100
  - The first comparison operation returns false
  - Is there any reason to even bother testing the others?
    - No: the result of false \(\land\) any-other-conditions must be false

Short-circuit evaluation

- Consider these logical expressions:
  \[(x < -10) \lor (x > 10)\]
  \[(x < -10) \lor ((x > -1) \land (x < 1)) \lor (x > 10)\]

- Suppose that \(x\) has the value 0
  - The first condition is false, and
    - In the first example, \((x > 10)\) is false and it is the last condition, so the expression is false
    - In the second example, \(((x > -1) \land (x < 1))\) is true, so the entire logical expression is true
      - There is no need at this point to evaluate \((x > 10)\)
      - Even though it is false, the entire expression is still true

- Similarly, consider
  \[(x > -10) \land (x < 10)\]
  \[(x > -10) \land ((x < -1) \lor (x > 1)) \land (x < 10)\]

- Suppose that \(x\) has the value 0
  - The first condition is true, and
    - In the first example, \((x < 10)\) is true and it is the last condition, so the expression is true
    - In the second example, \(((x < -1) \lor (x > 1))\) is false, so the entire logical expression is false
      - There is no need at this point to evaluate \((x < 10)\)
      - Even though it is true, the entire expression is still false
Short-circuit evaluation

- Suppose that x is a local variable:
  ```cpp
  if ( ((x >= -1.0) && (x <= 1.0)) || (x > 10.0) || (x < -10.0) ) {
    std::cout << "true" << std::endl;
  } else {
    std::cout << "false" << std::endl;
  }
  ```

- When do they stop evaluating the local variable x equals:
  -12  -5  -1  7  15

Logical operators

Logical negation

- If a logical expression is true, its negation is false, and vice versa
  - The unary NOT operator

<table>
<thead>
<tr>
<th>x</th>
<th>!x</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>false</td>
<td>true</td>
</tr>
</tbody>
</table>

- Consider, for example
  ```cpp
  if ( (x > 0) && (x < 10) ) {
    std::cout << "'x' is in the open interval (0, 10)" << std::endl;
  }
  ```

- If the operands are the same, the result is the same.

Logical negation

- The following Boolean-valued statements are equivalent:

<table>
<thead>
<tr>
<th>(x != 1)</th>
<th>!(x == 1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x &gt; 0)</td>
<td>!(x &lt;= 0)</td>
</tr>
<tr>
<td>(x &gt;= -1) &amp;&amp; (x &lt;= 1)</td>
<td>!(x &lt; -1)</td>
</tr>
</tbody>
</table>

Note that all three are the same:
```cpp
if ( ((x >= -1.0) && (x <= 1.0)) || (x > 10.0) || (x < -10.0) ) {
  std::cout << "'x' is not in the open interval (0, 10)" << std::endl;
}
```
Logical operators

• Logical negation

- The behavior of these two conditional statements are equivalent:
  
  ```
  if ( some-condition ) {
    // Do something
  } else {
    // Do something completely different
  }

  if ( !some-condition ) {
    // Do something completely different
  } else {
    // Do something
  }
  ```

Summary

• Following this lesson, you now:
  - Understand that two or more conditions can be chained together
    • With a logical AND (&&), all must be true for the result to be true
    • With a logical OR (||), one must be true for the result to be true
  - Are familiarized with truth tables
  - Understand the idea of short-circuit evaluation
    • As soon as one condition is false in a chain of logical ANDs, we're done: the result must be false
    • As soon as one condition is true in a chain of logical ORs, we're done: the result must be true
  - Understand that logical negation switches between true and false

References

[1] No references?

Acknowledgements

• None so far.
Colophon

These slides were prepared using the Georgia typeface. Mathematical equations use Times New Roman, and source code is presented using Consolas.

The photographs of lilacs in bloom appearing on the title slide and accenting the top of each other slide were taken at the Royal Botanical Gardens on May 27, 2018 by Douglas Wilhelm Harder. Please see https://www.rbg.ca/ for more information.

Disclaimer

These slides are provided for the ECE 150 Fundamentals of Programming course taught at the University of Waterloo. The material in it reflects the authors’ best judgment in light of the information available to them at the time of preparation. Any reliance on these course slides by any party for any other purpose are the responsibility of such parties. The authors accept no responsibility for damages, if any, suffered by any party as a result of decisions made or actions based on these course slides for any other purpose than that for which it was intended.