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

  `<  <=  ==  >=  >`
  `!=`

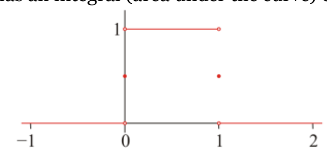
- Problem:
  - What if more than one condition is required?
  - What if two conditions result in the same consequent?
  - What if we require that a condition must be false?

The unit pulse

- Suppose we want to implement the function:

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

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



The unit pulse

- We could implement this program as follows:

```
#include <iostream>

// Function declarations
int main();

// Function definitions
int main() {
    double x{};
    std::cout << "Enter a number: ";
    std::cin >> x;

    if ( x < 0.0 ) {
        std::cout << 0.0 << std::endl;
    } else if ( x == 0.0 ) {
        std::cout << 0.5 << std::endl;
    } else if ( x < 1.0 ) {
        std::cout << 1.0 << std::endl;
    } else if ( x == 1.0 ) {
        std::cout << 0.5 << std::endl;
    } else {
        std::cout << 0.0 << std::endl;
    }

    return 0;
}
```

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

The unit pulse

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

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

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

- 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

The unit pulse

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

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

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

- 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

Logical operators

- In C++, there are two binary logical operators
  - They take two Boolean-valued operands and return a Boolean value
- The OR operator `||` returns `true` if either operands is `true`
- The AND operator `&&` returns `true` if both operands are `true`

Consequent	Conditions	C++
`0.0`	$x<0$ OR $x>1$	`(x < 0.0) || (x > 1.0)`
`0.5`	$x=0$ OR $x=1$	`(x == 0.0) || (x == 1.0)`
`1.0`	$x>0$ AND $x<1$	`(x > 0.0) && (x < 1.0)`

Logical operators

- Thus, we may implement this as follows:

```
if ( (x < 0.0) || (x > 1.0) ) {
    std::cout << 0.0 << std::endl;
} else if ( (x == 0.0) || (x == 1.0) ) {
    std::cout << 0.5 << std::endl;
} else {
    std::cout << 1.0 << std::endl;
}

if ( (x > 0.0) && (x < 1.0)  ) {
    std::cout << 1.0 << std::endl;
} else if ( (x == 0.0) || (x == 1.0)  ) {
    std::cout << 0.5 << std::endl;
} else {
    std::cout << 0.0 << std::endl;
}
```


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

Maximum of three

- We could implement this program as follows:

```
#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 could implement this program as follows:

```
    if ( (x >= y) && (x >= z) ) {
        std::cout << "max(x, y, z) = " << x << std::endl;
    } else if ( y >= z ) {
        std::cout << "max(x, y, z) = " << y << std::endl;
    } else {
        std::cout << "max(x, y, z) = " << z << std::endl;
    }

    return 0;
}
```

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 operand are `true`
  - It is `false` if either operands 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

+	0	1	2	3	4	5	6	7	8	9
0	0	1	2	3	4	5	6	7	8	9
1	1	2	3	4	5	6	7	8	9	10
2	2	3	4	5	6	7	8	9	10	11
3	3	4	5	6	7	8	9	10	11	12
4	4	5	6	7	8	9	10	11	12	13
5	5	6	7	8	9	10	11	12	13	14
6	6	7	8	9	10	11	12	13	14	15
7	7	8	9	10	11	12	13	14	15	16
8	8	9	10	11	12	13	14	15	16	17
9	9	10	11	12	13	14	15	16	17	18

×	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
1	0	1	2	3	4	5	6	7	8	9
2	0	2	4	6	8	10	12	14	16	18
3	0	3	6	9	12	15	18	21	24	27
4	0	4	8	12	16	20	24	28	32	36
5	0	5	10	15	20	25	30	35	40	45
6	0	6	12	18	24	30	36	42	48	54
7	0	7	14	21	28	35	42	49	56	63
8	0	8	16	24	32	40	48	56	64	72
9	0	9	18	27	36	45	54	63	72	81

Truth tables

- With only two possible values of the operands, these truth tables are much simpler:

x && y		y: true	y: false
x	true	true	false
	false	false	false

x || y		y: true	y: false
x	true	true	true
	false	true	false

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

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 ||

Logical expressions

- For example, our program may have:

```
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 expressions

- If there are many conditions that must be `true`,
  you can string these together with &&:

```
if ( (w > 0.0) && (x > 0.0) && (y > 0.0) && (z > 0.0) ) {
    // Do something specific to this condition
}
```

- If there are many conditions of which only one need be `true`,
  you can string these together with ||:

```
if ( (w > 0.0) || (x > 0.0) || (y > 0.0) || (z > 0.0) ) {
    // Do something specific to this condition
}
```


Logical expressions

- If you are mixing such conditions, use parentheses to be clear:

```
if ( ((x > 0.0) && (x < 1.0)) || ((y > 0.0) && (y < 1.0)) ) {
    // 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

- For example, our program may have:

```
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 expressions

```
    std::cout << "Do you have any of:" << std::endl;
    std::cout << "\tdifficulty breating," << std::endl;
    std::cout << "\tchest pains, or" << std::endl;
    std::cout << "\tloss 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::cout;
    } 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) || (x > 10)`
  `(x < -10) || ((x > -1) && (x < 1)) || (x > 10)`

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

Short-circuit evaluation

- Consider these logical expressions:
  `(x < -10) || (x > 10)`
  `(x < -10) || ((x > -1) && (x < 1)) || (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) && (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`

Short-circuit evaluation

- Similarly, consider
  `(x > -10) && (x < 10)`
  `(x > -10) && ((x < -1) || (x > 1)) && (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 && any-other-conditions` must be `false`

Short-circuit evaluation

- Similarly, consider
  `(x > -10) && (x < 10)`
  `(x > -10) && ((x < -1) || (x > 1)) && (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) || (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:

```
if ( ((x >= -1.0) && (x <= 1.0)) || (x > 10.0) || (x < -10.0) ) {
    std::cout << "true" << std::endl;
} else {
    std::cout << "false" << std::endl;
}

if ( (x < -10.0) || (x > 10.0) || ((x <= 1.0) && (x >= -1.0)) ) {
    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 negation

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

x	!x
true	false
false	true

- Consider, for example

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

if ( !((x > 0) && (x < 10)) ) {
    std::cout << "'x' is not in the open interval (0, 10)" << std::endl;
}
```


Logical negation

- Note that all three are the same:

```
if ( !((x > 0) && (x < 10)) ) {
    std::cout << "'x' is not in the open interval (0, 10)" << std::endl;
}

if ( (!(x > 0) || !(x < 10)) ) {
    std::cout << "'x' is not in the open interval (0, 10)" << std::endl;
}

if ( (x <= 0) || (x >= 10) ) {
    std::cout << "'x' is not in the open interval (0, 10)" << std::endl;
}
```


Logical negation

- The following Boolean-valued statements are equivalent[1]:

```
         (x != 1)                  !(x == 1)
         (x  >  0)                 !(x <= 0)
  (x >= -1) && (x <= 1)      !((x < -1) || (x > 1))
```

[1]If the operands are the same, the result is the same.

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.