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
  - Three complementary pairs

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

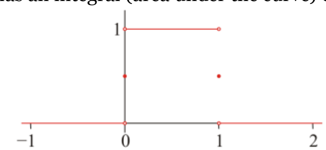
- Problem:
  - What if more than one condition is required?
  - What if two conditions result in the same consequent?
  - What if we require that a condition must be false?

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 function as follows:

```
double unit( double x );

double unit( double x ) {
    if ( x < 0.0 ) {
        return 0.0;
    } else if ( x == 0.0 ) {
        return 0.5;
    } else if ( x < 1.0 ) {
        return 1.0;
    } else if ( x == 1.0 ) {
        return 0.5;
    } else {
        return 0.0;
    }
}
```

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

The unit pulse

- Can we implement this using two consequents and one alternative?

```
double unit( double x );

double unit( double x ) {
    if ( Condition A ) {
        return 0.0;
    } else if ( Condition B ) {
        return 0.5;
    } else {
        return 1.0;
    }
```

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

Logical operators

- In English, we would simply say that the result is
  - 0 if either $x < 0$ **or** $x > 1$,
  - 0.5 if either $x = 0$ **or** $x = 1$, and
  - 1 if both $x > 0$ **and** $x < 1$

Logical operators

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

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

Logical operators

- Thus, we may implement the function as:

```
#include <cassert>
double unit( double x );

double unit( double x ) {
    if ( (x < 0) || (x > 1) ) {
        return 0.0;
    } else if ( (0 == x) || (1 == x) ) {
        return 0.5;
    } else {
        assert( (x > 0) && (x < 1) );
        return 1.0;
    }
}
```


Maximum of three

- We can now implement the maximum-of-three function as follows:

```
double max( double x, double y, double z );

double max( double x, double y, double z ) {
    if ( (x >= y) && (x >= z) ) {
        // 'x' is the maximum if it is greater than or equal
        // to 'y' and greater than or equal to 'z'
        return x;
    } else if ( y >= z ) {
        // Now, 'y' is the maximum if 'y' is greater than or
        // equal to 'z'
        //  - if 'y' was not greater than 'x', the first
        //    condition would have been true
        return y;
    } else {
        return z;
    }
}
```


Truth tables

- We know that the logical OR operator `||` is `true` if either operand is `true`
  - It is `false` if both operands are `false`
- We know that 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:

&&	true	false
true	true	false
false	false	false

||	true	false
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

A reminder…

- Just to remind you, however, the result of a logical operator is simply `0` or `1`:

```
// All these print '1':
std::cout << ((3 < 4)  && (4 <  5) ) << std::endl;
std::cout << ((6 > 12) || (4 <= 5) ) << std::endl;
std::cout << ((3 == 3) || (5 >  0) ) << std::endl;
std::cout << ((3 <= 4) || (6 >= 15)) << std::endl;
```


Multiple conditions that may be true

- If you have four conditions, **any** of which need be true, parentheses are not necessary:

```
if ( (1st-cond) || (2nd-cond) || (3rd-cond) || (4th-cond) ) {
    // Do something...
}
```

- Like addition, logical OR is associative:

```
(a  +  b)  +  c == a  +  (b  +  c)
(a || b) || c == a || (b || c)
```

- If any one condition is `true`, then
  `(1st-cond) || (2nd-cond) || (3rd-cond) || (4th-cond)`
  evaluates to `true`
  - If all conditions are `false`, the logical expression evaluates to `false`

Multiple conditions that must be true

- If you have four conditions, all of which **must** be true, you need only parenthesize the operands

```
if ( (1st-cond) && (2nd-cond) && (3rd-cond) && (4th-cond) ) {
    // Do something...
}
```

- Like addition and multiplication,
  both logical OR and logical AND are associative

- If all conditions are `true`, then
  `(1st-cond) && (2nd-cond) && (3rd-cond) && (4th-cond)`
  evaluates to `true`
  - If even one condition is `false`, the logical expression evaluates to `false`

Multiple conditions

- Note that you may combine both logical operators, but you must be clear what you mean:
  `(x == 0) || ((x <= 2) && (x >= 1))`
  is very different from
  `((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]

- If you leave it as
  `(x == 0) || (x <= 2) && (x >= 1)`
  the compiler will decide, and programmers will be left guessing

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

- Suppose now that `'x'` has the value `0`:

```
(x < -10) || (x > 10)
(x < -10) || ((x > -1) && (x < 1)) || (x > 10)
```

- 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

- Suppose now that `'x'` has the value `0`:
```
(x > -10) && (x < 10)
(x > -10) && ((x < -1) || (x > 1)) && (x < 10)
```
- 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

- These functions have equivalent logical expressions:
```
int f( int x ) {
    if ( ((x >= -1) && (x <= 1)) || (x > 10) || (x < -10) ) {
        return -1;
    } else {
        return 1;
    }
}

int g( int x ) {
    if ( (x < -10) || (x > 10) || ((x <= 1) && (x >= -1)) ) {
        return -1;
    } else {
        return 1;
    }
}
```
- When do they stop evaluating when the argument passed is:
```
-12  -5  -1  7  15
```


Logical negation

- Suppose we want to print a message if some number is not divisible by 13:
```
int print_good_luck( int n );

int print_good_luck( int n ) {
    if ( is_divisible( n, 13 ) ) {
        // Do nothing
    } else {
        std::cout << "Good choice!" << std::endl;
    }
}
```

Logical negation

- Verbally, you would simply say: "If *n* is not divisible by 13, ..."
- We can do this in C++ by using the unary logical NOT operator !:

```
int print_good_luck( int n );

int print_good_luck( int n ) {
    if ( !is_divisible( n, 13 ) ) {
        std::cout << "Good choice!" << std::endl;
    }
}
```

An alternative is:

```
if ( is_divisible( n, 13 ) == false ) {
```


Logical negation

- If a Boolean value is true, its negation is false, and vice versa

x	!x
true	false
false	true

Logical negation

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

x is not equal to 1 `(x != 1)`	It is not true that *x* is equal to 1 `!(x == 1)`
x is greater than 0 `(x > 0)`	It is not true that *x* is less than or equal to 0 `!(x <= 0)`
x is between –1 and 1 `(x >= -1) && (x <= 1)`	It is not true that *x* is less than –1 or greater than 1 `!((x < -1) || (x > 1))`
x – *y* when divided by 2 has a remainder of 0 `((x - y) % 2) == 0`	It is not true that *x* – *y* when divided by 2 has a remainder of 1 `!(((x - y) % 2) == 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
}
```

Logical negation

- Once again, all the unary logical NOT operator does is change the value of `true` (that is, `1`) to `false` (`0`) and vice versa

```
void check( int n ) {
    std::cout << !(n == 0) << std::endl;
    std::cout <<  (n != 0) << std::endl;

    std::cout << !(n >= 1) << std::endl;
    std::cout <<  (n <  1) << std::endl;

    std::cout << !((n == 0) || (n >= 1)) << std::endl;
    std::cout <<  ((n != 0) && (n <  1)) << std::endl;
}
```


Decision making

- Why do we include these operators?
  - The literal logical values `true` and `false` together with operations such as AND, OR and NOT are sufficient to define Boolean logic
  - In 1938, Claude Shannon wrote his master's thesis where he demonstrated that the behavior of relays can be modelled by Boolean logic
- A relay is a switch that can be turned on or off
  - Usually with an electromagnet
- Transistors are excellent solid-state approximations of switches
  - Their behavior can still be modelled by Boolean logic

One final aside…

- In Claude Shannon's master's thesis, written in 1937 when he was 21-years old, he demonstrated that Boolean algebra was sufficient to construct any logical, numerical relationship
  - He founded information theory
  - Shannon's maxim: "The enemy knows the system"
  - He also invented the *ultimate machine*:

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?

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.