Expressions

Programming in C

Shirley B. Chu
shirley.chu@delasalle.ph

College of Computer Studies
De La Salle University

October 27, 2021

Expressions

- used for data manipulation

Expressions

- used for data manipulation
- may either be

Expressions

- used for data manipulation
- may either be
  - arithmetic,

Expressions

- used for data manipulation
- may either be
  - arithmetic,
  - relational, or

Expressions

- used for data manipulation
- may either be
  - arithmetic,
  - relational, or
  - logical

Arithmetic Operators

Arithmetic Operators

`+` `-` `*` `/` `%`

Things to remember:

- Operator precedence is still **MD**/**AS**, evaluated from **left to right** for operators with the same priority.

Arithmetic Operators

Arithmetic Operators

`+` `-` `*` `/` `%`

Things to remember:

- Operator precedence is still **MD**/**AS**, evaluated from **left to right** for operators with the same priority.

- When both operands are integers, the result is also an integer. Otherwise, the result is a floating-point number.

Arithmetic Operators

Arithmetic Operators

`+` `-` `*` `/` `%`

Things to remember:

- Operator precedence is still **MD**/**AS**, evaluated from **left to right** for operators with the same priority.
- When both operands are integers, the result is also an integer. Otherwise, the result is a floating-point number.
- The modulo `%` operator can have integer operands only.

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`
   `7 - 2 + 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`
   `7 - 2 + 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`
   `7 - 2 + 3`
   `5 + 3`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`
   `7 - 2 + 3`
   `5 + 3`

Examples

1. 5 + 5 / 2 - 2 + 1 * 9 / 3
   5 + 2 - 2 + 1 * 9 / 3
   5 + 2 - 2 + 9 / 3
   5 + 2 - 2 + 3
   7 - 2 + 3
   5 + 3
   8

Examples

1.
```
5 + 5 / 2 - 2 + 1 * 9 / 3
5 + 2 - 2 + 1 * 9 / 3
5 + 2 - 2 + 9 / 3
5 + 2 - 2 + 3
7 - 2 + 3
5 + 3
8
```

2.
```
15 % 6 * 2.5 + 1
```

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`
   `7 - 2 + 3`
   `5 + 3`
   `8`

2. `15 % 6 * 2.5 + 1`

3. `20 / 6 + 4 / 5 * 10`

Examples

1. `5 + 5 / 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 1 * 9 / 3`
   `5 + 2 - 2 + 9 / 3`
   `5 + 2 - 2 + 3`
   `7 - 2 + 3`
   `5 + 3`
   `8`

2. `15 % 6 * 2.5 + 1`

3. `20 / 6 + 4 / 5 * 10`

4. `20 / 6 + 4 / 5.0 * 10`

Evaluating Expressions

Precedence determines the order/priority of operators

Evaluating Expressions

Precedence determines the order/priority of operators

Associativity determines the direction of evaluation

Evaluating Expressions

Precedence determines the order/priority of operators

Associativity determines the direction of evaluation

Example: Given `a = 11`, `b = 2`, and `c = 5`, evaluate the following:

1. `a / b / c`
2. `(a / b) / c`
3. `a / (b / c)`

Arithmetic Operators

highest	`()`	left to right
	unary `+`, unary `-`	right to left
	`%`, `*`, `/`	left to right
	`+`, `-`	left to right

Relational Operators

Relational Operators

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

`==` equality

`!=` inequality

Things to remember:

Relational Operators

Relational Operators

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

equality inequality

Things to remember:

- Evaluating relational expressions results to **true** or **false**

Relational Operators

Relational Operators

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

`==` equality, `!=` inequality

Things to remember:

- Evaluating relational expressions results to **true** or **false**
- In C99, the result type of relational expressions is `int`

Relational Operators

Relational Operators

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

equality inequality

Things to remember:

- Evaluating relational expressions results to **true** or **false**
- In C99, the result type of relational expressions is `int`
  - `0` for false
  - `1` for true

Relational Operators

highest	`()`	left to right
	unary `+`, unary `-`	right to left
	`%`, `*`, `/`	left to right
	`+`, `-`	left to right
	`>`, `>=`, `<`, `<=`	left to right
	`==`, `!=`	left to right

Logical Operators

Logical Operators

!	&&	||
not	and	or

Things to remember:

Logical Operators

Logical Operators

!	&&	||
not	and	or

Things to remember:

- Evaluating logical expressions results to **true** or **false**

Logical Operators

Logical Operators

`!`	`&&`	`||`
not	and	or

Things to remember:

- Evaluating logical expressions results to **true** or **false**
- In C99, the result type of logical expressions is `int`
  - `0` for false
  - `1` for true

Logical Not

- written in C as !

Logical Not

- written in C as !
- is a unary operator

Logical Not

- written in C as !
- is a unary operator
- is a negation operator. It inverts the value of the operand.

Logical Not

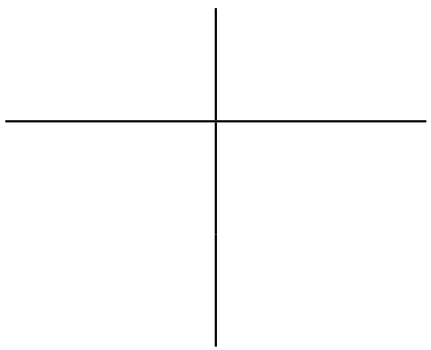
- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

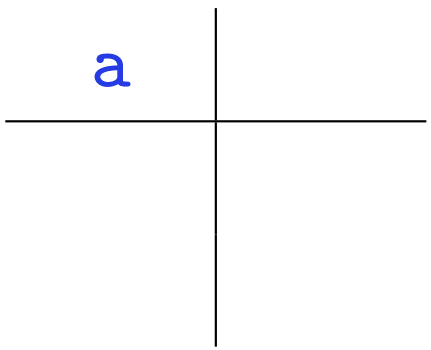
Example: `!a`



Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`



Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

a	!a

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

`a`	`!a`
true	
false	

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

a	!a
true	
false	

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

a	!a
true	false
false	

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

a	!a
true	false
false	

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

`a`	`!a`
true	false
false	true

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

a	!a
true	false
false	true

Evaluate:

1. `!5`

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

a	!a
true	false
false	true

Evaluate:

1. `!5`
2. `!(3 < 3)`

Logical Not

- written in C as `!`
- is a unary operator
- is a negation operator. It inverts the value of the operand.
- result is `0` if the value of its operand compares unequal to 0, `1` if the value of its operand compares equal to 0. The expression `!x` is equivalent to `0 == x`.

Example: `!a`

a	!a
true	false
false	true

Evaluate:

1. `!5`
2. `!(3 < 3)`
3. `!(4 - 2 * 2)`

Logical And

- written in C as &&

Logical And

- written in C as &&
- is a binary operator

Logical And

- written in C as &&
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Logical And

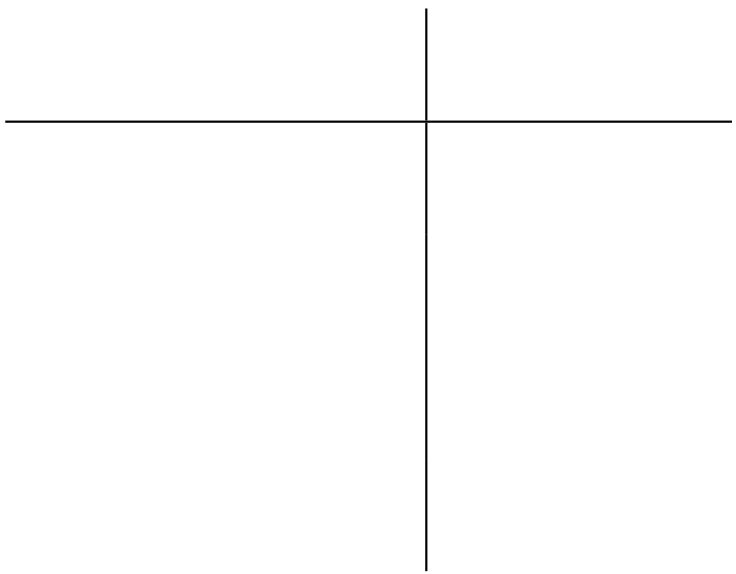
- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`



Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true		
true		

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true		
true		
false		
false		

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	
true		
false		
false		

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	
true	false	
false		
false		

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	
true	false	
false	true	
false		

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	
true	false	
false	true	
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	
true	false	
false	true	
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	
false	true	
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	
false	true	
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Evaluate:

1. `(5 > 4) && (6 > 7)`

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Evaluate:

1. `(5 > 4) && (6 > 7)`
2. `!(3 < 3) && 8`

Logical And

- written in C as `&&`
- is a binary operator
- results to **true** when **both** operands are **true**. Otherwise, the result is **false**.

Example: `a && b`

a	b	a && b
true	true	true
true	false	false
false	true	false
false	false	false

Evaluate:

1. `(5 > 4) && (6 > 7)`
2. `!(3 < 3) && 8`
3. `!(4 - 2 * 2) && (5 % 3 == 2)`

Logical Or

- written in C as ||

Logical Or

- written in C as `||`
- is a binary operator

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Logical Or

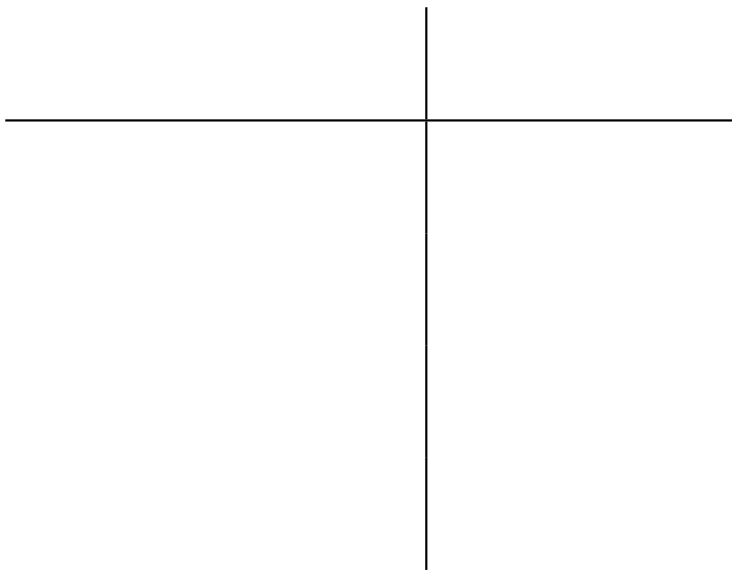
- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`



Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

a	b	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true		
true		

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true		
true		
false		
false		

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	
true		
false		
false		

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	
true	false	
false		
false		

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	
true	false	
false	true	
false		

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	
true	false	
false	true	
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	
true	false	
false	true	
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	true
true	false	
false	true	
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	true
true	false	
false	true	
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	true
true	false	true
false	true	
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	true
true	false	true
false	true	
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	true
true	false	true
false	true	true
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	true
true	false	true
false	true	true
false	false	

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

`a`	`b`	`a || b`
true	true	true
true	false	true
false	true	true
false	false	false

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Evaluate:

1. `(5 < 4) || (6 > 7)`

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Evaluate:

1. `(5 < 4) || (6 > 7)`
2. `!(3 <= 3) || 8`

Logical Or

- written in C as `||`
- is a binary operator
- results to **true** when **at least one** operand is **true**. Otherwise, the result is **false**.

Example: `a || b`

a	b	a || b
true	true	true
true	false	true
false	true	true
false	false	false

Evaluate:

1. `(5 < 4) || (6 > 7)`
2. `!(3 <= 3) || 8`
3. `!(4 / 2 * 2) || (5 % 3 == 2)`

Logical Operators

highest	`()`	left to right
	unary `+`, unary `-` `!`	right to left
	`%`, `*`, `/`	left to right
	`+`, `-`	left to right
	`>`, `>=`, `<`, `<=`	left to right
	`==`, `!=`	left to right
	`&&`	left to right
	`||`	left to right

C Operator Precedence Table: https://en.cppreference.com/w/c/language/operator_precedence

Exercise 1

Evaluate the following.

1. `8 > 10 || 6 < 5 && 9 != 10`

Exercise 1

Evaluate the following.

1. `8 > 10 || 6 < 5 && 9 != 10`

2. Let `a = -5` and `b = 10`,
   `a + b >= 10 && !(a / b < 5) && b % a == 1 / 2`

Exercise 1

Evaluate the following.

1. `8 > 10 || 6 < 5 && 9 != 10`

2. Let `a = -5` and `b = 10`,
   `a + b >= 10 && !(a / b < 5) && b % a == 1 / 2`

3. `!(45 / 6 > 18 % 7) || 95 >= 77 + 23 % 3 * 1.0 &&`
   `45 == 75 - 6 * 5`

Exercise 1

Evaluate the following.

1. `8 > 10 || 6 < 5 && 9 != 10`

2. Let `a = -5` and `b = 10`,
   `a + b >= 10 && !(a / b < 5) && b % a == 1 / 2`

3. `!(45 / 6 > 18 % 7) || 95 >= 77 + 23 % 3 * 1.0 &&`
   `45 == 75 - 6 * 5`

4. `4 * 5.0 / 8 - 10`

Exercise 2

Express the following using expressions in C. Use the underlined words as identifiers.

1. Determine if *value* is an even number.

Exercise 2

Express the following using expressions in C. Use the underlined words as identifiers.

1. Determine if *value* is an even number.
2. Compute for Anna's age: *anna* is 3 times *luisa*'s age.

Exercise 2

Express the following using expressions in C. Use the underlined words as identifiers.

1. Determine if *value* is an even number.
2. Compute for Anna's age: *anna* is 3 times *luisa*'s age.
3. Determine if *mark* is older than *pete*.

Exercise 2

Express the following using expressions in C. Use the underlined words as identifiers.

1. Determine if *value* is an even number.
2. Compute for Anna's age: *anna* is 3 times *luisa*'s age.
3. Determine if *mark* is older than *pete*.
4. $x = \dfrac{a + 3b}{2c}$

☺ Thank you! ☺