Logical Expressions

This lesson covers a particular type of expression that is used in conditional statements.

WE'LL COVER THE FOLLOWING

Expressions

The actual work that a program performs is accomplished by *expressions*. Any part of a program that produces a value or a side effect is called an **expression**. It has a very wide definition because even a constant value like 42 and a string like "hello" are expressions since they produce the respective constant values 42 and "hello."

Note: Values such as constants above can be used independently of

> **Note:** Values such as constants above can be used independently of variables

Functions like `writeln` are also *expressions* because they also produce values for the output stream. In the case of `writeln`, the output stream is modified by placing characters on it. Another example from the programs we've written so far is the assignment operation, which also modifies the variable on its left-hand side.

Expressions can be part of other expressions. This allows us to form more complex expressions from simpler ones. For example, assuming that there is a function named `currentTemperature` that produces the value of the current air temperature, the value that it produces may directly be used in a `writeln` expression:

```
writeln("It's ", currentTemperature(), " degrees at the moment.");
```

This line consists of four expressions:

1. "It's "
2. `currentTemperature()`
3. " degrees at the moment."
4. The `writeln()` expression that makes use of the other three.

In this lesson, we will cover the particular type of expression that is used in conditional statements.

Before going further, though, let us revisit the assignment operator discussed in the lesson assigning values to variables, this time emphasizing the two expressions that appear on its left and right sides: the assignment operator `=` assigns the value of the expression on its right-hand side to the expression on its left-hand side (e.g. to a variable).

```
temperature = 23 // temperature's value becomes 23
```

Logical expressions

Logical expressions are used in *boolean arithmetic*. These are what make computer programs make decisions like "if the answer is yes, I will save the file."

Logical expressions can have only one of the two values:

- **false** that indicates *falsity*
- **true** that indicates *truth*

Let's see the use of `writeln` expressions in the following examples. If a line has *true* printed at the end, it means that what is printed on the line is *true*. Similarly, *false* will mean that what is on the line is false. For example, if the output of a program is the following,

```
There is coffee: true
```

*then it will mean that "*there is coffee"**. Similarly,

```
There is coffee: false
```

will mean that "there isn't coffee".

> **Note:** I use the "... is ...: false" construct to mean "is not" or "is false".

Logical expressions are used extensively in conditional statements, loops and function parameters. It is essential to understand how they work. Luckily, logical expressions are easy to explain and use.

Logical operators

The logical operators that are used in logical expressions are:

`==` operator

The `==` operator answers the question "**is equal to?**". It compares the two expressions on its left and right sides and produces *true* if they are equal and *false* if they are not. By definition, the value that `==` produces is a logical expression.
As an example, let's assume that we have the following two variables:

```
int daysInWeek = 7;
int monthsInYear = 12;
```

The following code has two logical expressions that use these values:

```
import std.stdio;

int main() {
    int daysInWeek = 7;
    int monthsInYear = 12;

    writeln(daysInWeek == 7);    // true
    writeln(monthsInYear == 11); // false
  return 0;
}
```

`!=` operator

The `!=` operator answers the question “**is not equal to?**”. It compares the two expressions on its sides and produces the opposite of `==`.

```
import std.stdio;

int main() {
    int daysInWeek = 7;
    int monthsInYear = 12;

    writeln(daysInWeek != 7);    // false
    writeln(daysInWeek == 7);    // true (opposite of !=
    writeln(monthsInYear != 11); // true
  return 0;
}
```

`||` operator

The `||` operator means “**or**” and produces *true* if any one of the logical expressions is true. If the value of the left-hand expression is true, it produces *true* without even looking at the expression on the right-hand side. If the left-hand side is false, then it produces the value of the right-hand side. This operator is similar to the “or” in English: if the left one, the right one or both are true, then it produces true.
The following table lists all possible values for both sides of this operator and the result:

Left expression	Operator	Right expression	Result
false	||	false	false
false	||	true	true
true	||	false (not evaluated)	true
true	||	true (not evaluated)	true

Results of | | operator

```
import std.stdio;

void main() {
    // false means "no", true means "yes"

    bool existsCoffee = false;
    bool existsTea = true;

    writeln("There is warm drink: ",
            existsCoffee || existsTea);
}
```

Logical expression code

Because at least one of the two expressions is true, the logical expression above produces true.

`&&` operator

The `&&` operator means "**and**" and produces *true* if both of the expressions are true.
If the value of the left-hand expression is false, it produces *false* without even looking at the expression that is on the right-hand side. If the left-hand side is true, then it produces the value of the right-hand side as a result. This operator is similar to the "and" in English: if the left value and the right value are true, only then it produces true.

Left expression	Operator	Right expression	Result
false	&&	false (not evaluated)	false
false	&&	true (not evaluated)	false
true	&&	false	false
true	&&	true	true

Results of && operator

> **Note:** The fact that the `||` and `&&` operators may not evaluate the right-hand expression is called their **short-circuit behavior**. The ternary operator `?:`, which we will see in a later chapter, is similar in that it never evaluates one of its three expressions. All of the other operators always evaluate and use all of their expressions.

`^` operator

The `^` operator answers the question "**is one or the other but not both?**". This operator produces *true* if only one expression is true, but not both.

> **Note:** In reality, this operator is not a logical operator but an arithmetic one. It behaves like a logical operator only if both of the expressions are boolean.

Left expression	Operator	Right expression	Result
false	^	false	false
false	^	true	true
true	^	false	true
true	^	true	false

Results of ^ operator

For example, the logic that represents a user playing chess if only one of their two friends showed up can be coded like this:

```
import std.stdio;

int main() {
    bool jimShowedUp = false;
    bool bobShowedUp = true;

    writeln("I will play chess: ", jimShowedUp ^ bobShowedUp);
    return 0;
}
```

`<` operator

The `<` operator answers the question “**is less than?**” (or “does come before in sort order?”).

```
import std.stdio;

int main() {
    int theirScore = 3;
    int ourScore = 4;

    writeln("We beat them: ", theirScore < ourScore);
    return 0;
}
```

`>` operator

The `>` operator answers the question “**is greater than?**” (or “does come after in sort order?”).

```
import std.stdio;

int main() {
    int theirScore = 3;
    int ourScore = 4;

    writeln("They beat us: ", theirScore > ourScore);
    return 0;
}
```

`<=` operator

The `<=` operator answers the question “**is less than or equal to?**” (or “does come before or the same in sort order?”). This operator is the opposite of the `>` operator.

```
import std.stdio;

int main() {
    int theirScore = 3;
    int ourScore = 4;

    writeln("We were not beaten: ", theirScore <= ourScore);
    return 0;
}
```

`>=` operator

The `>=` operator answers the question “**is greater than or equal to?**” (or “does come after or the same in sort order?”). This operator is the opposite of the `<` operator.

```
import std.stdio;

int main() {
    int theirScore = 4;
    int ourScore = 4;

    writeln("We did not beat them: ", theirScore >= ourScore);
    return 0;
}
```

`!` operator

The `!` operator means “**the opposite of**”. Different from the other logical operators, it takes just one expression and produces true if that expression is false, and false if that expression is true.

```
import std.stdio;

int main() {
    bool existsBicycle = true;

    writeln("I will walk: ", !existsBicycle);
    return 0;
}
```

Grouping expressions

The order in which the expressions are evaluated can be specified by using parentheses to group them. When parenthesized expressions appear in more complex expressions, the parenthesized expressions are evaluated before they complex expressions, the parenthesized expressions are evaluated before they can be used in the expressions that they appear in. For example, the

expression “if there is coffee or tea, and also cookie or scone; then I am happy” can be coded like the following:

```
import std.stdio;

int main() {
    bool existsCoffee = false;
    bool existsTea = true;
    bool existsCookie = false;
    bool existsScone = true;

    writeln("I am happy: ",
           (existsCoffee || existsTea) && (existsCookie || existsScone));
    return 0;
}
```

If the sub-expressions were not parenthesized, the expressions would be evaluated according to the operator precedence rules of D (which have been inherited from the C language). Since in these rules `&&` has higher precedence over `||`, writing the expression without parentheses would not be evaluated as intended:

```
import std.stdio;

int main() {
    bool existsCoffee = false;
    bool existsTea = true;
    bool existsCookie = false;
    bool existsScone = true;

    writeln("I am happy: ",
           existsCoffee || existsTea && existsCookie || existsScone);
    return 0;
}
```

The `&&` operator would be evaluated first, and the whole expression would be the semantic equivalent of the following expression:

```
writeln("I am happy: ",
existsCoffee || (existsTea && existsCookie) || existsScone);
existsCoffee || (existsTea && existsCookie) || existsScone);
```

That has a totally different meaning: “if there is coffee, or tea and cookie, or scone; then I am happy”.

```
import std.stdio;

int main() {
    bool existsCoffee = false;
    bool existsTea = true;
    bool existsCookie = false;
    bool existsScone = true;

    writeln("I am happy: ",
           existsCoffee || (existsTea && existsCookie) || existsScone);
    return 0;
}
```

Reading bool input

All of the bool values above are automatically displayed as `false` or `true`. Also, `readf()` automatically converts strings `false` and `true` to bool values false and true, respectively. It accepts any combination of lower and uppercase letters as well. For example, ‘False’ and ‘FALSE’ are converted to false and ‘True’ and ‘TRUE’ are converted to true.

In the next lesson, you will find a coding challenge related to logical expressions.