Lazy Operators

This lesson explains how lazy operators can be used in D language.

Lazy operators

Lazy evaluation is the delaying of the execution of expressions until the results of those expressions are needed. Naturally, this delay may make programs run faster if the results end up not being needed.

A concept similar to lazy evaluation is the short-circuit behavior of the following operators:

- `||` **(or) operator:** The second expression is evaluated only if the first expression is false.

  ```
  if (anExpression() || mayNotBeEvaluated()) {
      // ...
  }
  ```

  If the result of `anExpression()` is true, then the result of the `||` expression is necessarily true. Since we no longer need to evaluate the second expression to determine the result of the `||` expression, the second expression is not evaluated.

- `&&` **(and) operator:** The second expression is evaluated, only if the first expression is true.

  ```
  if (anExpression() && mayNotBeEvaluated() {
      // ...
  }
  ```

If the result of `anExpression()` is false, then the result of the `&&` expression is necessarily false, so the second expression is not evaluated.

- `?:` **(ternary) operator:** Either the first or the second expression is evaluated, depending on whether the condition is true or false, respectively.

  ```
  int i = condition() ? eitherThis() : orThis();
  ```

The laziness of these operators matters not only to performance. Sometimes, evaluating one of the expressions can be an error.
For example, “is the first letter an A” condition check below would produce an error when the `string` is empty:

```
dstring s;
// ...
if (s[0] == 'A') {
    // ...
}
```

In order to access the first element of `s`, we must first ensure that the `string` does have such an element. For that reason, the following condition check moves that potentially erroneous logical expression to the right-hand side of the `&&` operator to ensure that it will be evaluated only when it is safe to do so:

```
if ((s.length >= 1) && (s[0] == 'A')) {
    // ...
}
```

Lazy evaluations can be achieved by using function pointers, delegates, and ranges as well.

In the next lesson, we will learn the use of while loop in D language.