Referenced Variables As Parameters

This lesson explains how referenced variables are not copied when used with functions.

WE'LL COVER THE FOLLOWING

Referenced variables are not copied

Even parameters of reference types like slices, associative arrays and class variables are copied to functions. However, the original variables that are referenced (i.e. elements of slices and associative arrays and class objects) are not copied. Effectively, such variables are passed to functions as references; the parameter becomes another reference to the original object. It means that a modification made through the reference modifies the original object as well.
Since strings are slices made up of characters, this concept applies to strings as well:

```
import std.stdio;

void makeFirstLetterDot(dchar[] str) {
    str[0] = '.';
}

void main() {
    dchar[] str = "abc"d.dup;
    makeFirstLetterDot(str);
    writeln(str);
}
```

String as a referenced parameter

The change made to the first element of the parameter affects the actual element in `main()`:

```
.bc
```

However, the original slice and associative array variables are still passed by copy. This may have surprising and seemingly unpredictable results unless the parameters are qualified as `ref` themselves.

Surprising reference semantics of slices

As we saw in the slices and other array features lesson, adding elements to a slice may terminate element sharing. Obviously, once sharing ends, a slice parameter like `str` above would not be a reference to the elements of the passed-in original variable anymore.
For example, the element that is appended by the following function will not be seen by the calling function:

```
import std.stdio;

void appendZero(int[] arr) {
    arr ~= 0;
    writefln("Inside appendZero()       : %s", arr);
}

void main() {
    auto arr = [ 1, 2 ];
    appendZero(arr);
    writefln("After appendZero() returns: %s", arr);
}
```

Effect of appendZero( ) on arr not reflected in the main( )

The element is appended only to the function parameter, not to the original slice.
If the new elements need to be appended to the original slice, then the slice must be passed as `ref`:

```
void appendZero(ref int[] arr) {
    // ...
}
```

Surprising reference semantics of associative arrays

Associative arrays that are passed as function parameters may cause surprises as well because associative arrays start their lives as null, not empty.

In this context, **null** means an uninitialized associative array. Associative arrays are initialized automatically when their first key-value pair is added. As a consequence, if a function adds an element to a null associative array, then this element cannot be seen in the original array. Although the parameter is initialized, the original variable remains null:

```
import std.stdio;

void appendElement(int[string] aa) {
    aa["red"] = 100;
    writefln("Inside appendElement()       : %s", aa);
}

void main() {
    int[string] aa;    // ← null to begin with
    appendElement(aa);
    writefln("After appendElement() returns: %s", aa);
}
```

The original variable does not have the added element

On the other hand, if the associative array was not null to begin with, then the added element would be seen by the calling function as well:

```
int[string] aa;
aa["blue"] = 10; // ← Not null before the call
appendElement(aa);
```

This time the added element is seen by the caller:

```
Inside appendElement()       : ["red":100, "blue":10]
After appendElement() returns: ["red":100, "blue":10]
```

Added element can be seen by the caller

For that reason, it may be better to pass the associative array as a `ref`

For that reason, it may be better to pass the associative array as a `ref` parameter.

In the next lesson, we will start exploring parameter qualifiers.