foreach Loop

This lesson explains the foreach loop and how it is used with different data structures in D.

WE'LL COVER THE FOLLOWING

What is the `foreach` loop?

One of the most commonly used statements in D is the `foreach` loop. It is used to perform the same operation to every element of a container (or a range).

Operations that are applied to elements of containers are very common in programming. We have seen in the for loop lesson that the elements of an array are accessed with an index value that is incremented at each iteration:

```
import std.stdio;

void main() {

    int [] array = [1, 2, 3, 4, 5];
    for (int i = 0; i < array.length; ++i) {
        writeln(array[i]);
    }

}
```

The following steps are involved in iterating over all the elements:

- Defining a variable as a counter, which is conventionally named as `i`
- Iterating the loop up to the value of the `.length` property of the array
- Incrementing `i`
- Accessing the element

`foreach` has essentially the same behavior, but it simplifies the code by handling those steps automatically:

```
foreach (element; array) {
    writeln(element);
}
```

Part of the power of `foreach` comes from the fact that it can be used the same way regardless of the type of the container. As we have already seen, one way of iterating over the values of an associative array in a `for` loop is by first calling the array's `.values` property:

```
auto values = aa.values;
for (int i = 0; i != values.length; ++i) {
    writeln(values[i]);
}
```

`foreach` does not require anything special for associative arrays; it is used exactly the same as with arrays:

```
foreach (value; aa) {
    writeln(value);
}
```

The `foreach` syntax

`foreach` consists of three sections:

```
foreach (names; container or range) {
    operations
}
```

- **Container or range** specifies where the elements are.
- **Operations** specifies the operations to apply to each element.
- **Names** specifies the name of the element and potentially other variables depending on the type of the container or the range. Although the choice of names is up to the programmer, the number and the types of these names depend on the type of the container.

`continue` and `break`

These keywords have the same meaning for `foreach` as the `for` loop: `continue` moves to the next iteration before completing the rest of the operations for the current element, and `break` terminates the loop altogether.

`foreach` with arrays

When using `foreach` with plain arrays and a single name in the names section, that name represents the value of the element at each iteration:

```
foreach (element; array) {
    writeln(element);
}
```

When two names are specified in the names section, they represent an automatic counter and the value of the element, respectively:

```
foreach (i, element; array) {
    writeln(i, ": ", element);
}
```

```
import std.stdio;

void main(){

    int [] array = [1, 2, 3, 4, 5];

    foreach (i, element; array) {
        writeln(i, ": ", element);
    }
}
```

The counter is incremented automatically by `foreach`. Although it can be given any name, `i` is a very common name for the automatic counter.

`foreach` with strings and `std.range.stride`

Since strings are arrays of characters, `foreach` works with strings the same way as it does with arrays: A single name refers to the character, two names refer to the counter and the character, respectively:

```
import std.stdio;

void main() {
    foreach (c; "hello") {
        writeln(c);
    }

    foreach (i, c; "hello") {
        writeln(i, ": ", c);
    }
}
```

However, since `char` and `wchar` support UTF-8 code units, we need to iterate over UTF code units, not Unicode code points for these string types:

```
import std.stdio;

void main() {

    foreach (i, code; "abcçd") {
        writeln(i, ": ", code);
    }

}
```

The two UTF-8 code units that make up `ç` would be accessed as separate elements:

```
0: a
1: b
```

```
2: c
3: 

4: �
5: d
```

One way of iterating over Unicode characters of strings in a `foreach` loop is `stride` from the `std.range` module. `stride` considers the string as a container that consists of Unicode characters. It takes two arguments: the UTF-8 string and the number of steps that it should take as it strides over the characters:

```
import std.range;
import std.stdio;

void main() {

    foreach (c; stride("abcçd", 1)) {
        writeln(c);
    }

}
```

std.range module and foreach loop

`foreach` with associative arrays

When using `foreach` with associative arrays, a single name refers to the value, while two names refer to the *key* and the *value*, respectively:

```
import std.stdio;

void main() {
    int[string] aa = [ "blue" : 10, "green" : 20 ];

    foreach (value; aa) {
        writeln(value);
    }
    foreach (key, value; aa) {
        writeln(key, ": ", value);
    }
}
```

Associative arrays can provide their keys and values as ranges also. `.byKey`, `.byValue` and `.byKeyValue` return efficient range objects that are useful in contexts other than `foreach` loops.

`.byValue` does not have any benefit in `foreach` loops over the regular value iteration above. On the other hand, `.byKey` is the only efficient way of iterating over just the keys of an associative array:

```
import std.stdio;

void main() {

    int[string] aa = [ "blue" : 10, "green" : 20 ];

    foreach (key; aa.byKey) {
        writeln(key);
    }
}
```

`.byKeyValue` provides access to each key-value element through a variable that is similar to a tuple. The key and the value are accessed separately through the `.key` and `.value` properties of that variable:

```
import std.stdio;

void main() {
    int[string] aa = [ "blue" : 10, "green" : 20 ];

    foreach (element; aa.byKeyValue) {
        writefln("The value for key %s is %s", element.key, element.value);
    }

}
```

`foreach` with number ranges

We have seen number ranges before in the slices and other array features lesson. It is possible to specify a number range in the *container or range* section:

```
import std.stdio;
import std.stdio;

void main() {

    foreach (number; 10..15) {
        writeln(number);
    }

}
```

Remember that 10 would be included in the range but 15 would not be.

`foreach` with structs, classes and ranges

`foreach` can also be used with objects of user-defined types that define their own iteration in `foreach` loops. Structs and classes provide support for `foreach` iteration either by their `opApply()` member functions or by a set of range member functions.

In the next lesson, we will explore a few more properties of the `foreach` loop.