Format Specifiers: Positional Parameters and Format

This lesson explains how to position parameters in format specifiers, followed by a discussion of the `format()` function of the `std.string` module.

WE'LL COVER THE FOLLOWING

- Positional parameters
- `Format()` function
    - Checked format string

Positional parameters

We have seen above that the arguments are associated one by one with the specifiers in the format string. It is also possible to use position numbers within format specifiers. This enables the specifiers to associate with specific arguments. Arguments are numbered in increasing fashion, starting with 1. The argument numbers are specified immediately after the `%` character, followed by a `$`:

%	**position$**	**flags**	**width**	**precision**	**format_character**

Argument number

An advantage of positional parameters is being able to use the same argument in more than one place in the same format string:

```
writefln("%1$d %1$x %1$o %1$b", 42);
```

The format string above uses the argument numbered `1` within four specifiers to display it in decimal, hexadecimal, octal and binary formats:

```
42 2a 52 101010
```

```
import std.stdio;


void main() {

    writefln("%1$d %1$x %1$o %1$b", 42);

}
```

Another application of positional parameters is supporting multiple natural languages. When referred by position numbers, arguments can be moved anywhere within the specific format string for a given human language. For example, the number of students in a given classroom can be printed like the following:

```
import std.stdio;

void main() {

    int count = 20;
    string room = "1A";

    writefln("There are %s students in room %s.", count, room);

}
```

Let’s assume that the program must also support Turkish. In this case, the format `string` needs to be selected according to the active language. The following method takes advantage of the ternary operator:

```
auto format = (language == "en"
? "There are %s students in room %s."
: "%s sınıfında %s öğrenci var.");
writefln(format, count, room);
```

Unfortunately, when the arguments are associated one by one, the classroom and student `count` information appear in reverse order in the Turkish message; the `room` and `count` variable values are swapped.

```
20 sınıfında 1A öğrenci var. ← Wrong: means "room 20", and "1A students"!
```

```
import std.stdio;

void main() {

    int count = 20;
    string room = "1A";
    string language;

    auto format = (language == "en"
                    ? "There are %s students in room %s."
                    : "%s sınıfında %s öğrenci var.");
    writefln(format, count, room);
}
```

To avoid this, the arguments can be specified by numbers, such as `1$` and `2$`, to associate each specifier with the exact argument:

```
auto format = (language == "en"
? "There are %1$s students in room %2$s."
: "%2$s sınıfında %1$s öğrenci var.");
writefln(format, count, room);
```

Now, the arguments appear in the proper order regardless of the language selected:

```
There are 20 students in room 1A.
```

```
1A sınıfında 20 öğrenci var.
```

```
import std.stdio;

void main() {

    int count = 20;
    string room = "1A";
    string language;

    auto format = (language == "en"
                    ? "There are %1$s students in room %2$s."
                    : "%2$s sınıfında %1$s öğrenci var.");
    writefln(format, count, room);
}
```

`Format()` function

Formatted output is available through the `format()` function of the `std.string` module as well. `format()` works in the same way as `writef()`, but it returns the result as a string instead of displaying it as an output:

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

void main() {
    write("What is your name? ");
    auto name = strip(readln());

    auto result = format("Hello %s!", name);
    writeln(result);
}
```

format() function

The program can make use of that result in later expressions.

Checked format string

There is an alternative syntax for functions that take a format string (such as `writef`, `writefln`, `formattedWrite`, `readf`, `formattedRead`, etc.). It is possible to provide the format `string` as a template argument to these functions so that the validity of the format string and the arguments are checked at the compile time:

```
/* NOTE: This program is expected to fail compilation. */

import std.stdio;

void main() {
    writefln!"%s %s"(1);       // ← compilation ERROR (extra %s)
    writefln!"%s"(1, 2);       // ← compilation ERROR (extra 2)
    writefln!"%s %d"(1, 2.5);  // ← compilation ERROR (mismatched %d and 2.5)
}
```

format string compilation error

The `!` character above is the template instantiation operator.

> **Note:** Although this syntax is safer because it catches potential programmer errors at compile time, it may also make compilation times longer.

In the next lesson, you will find a coding challenge based on the concepts of formatted output.