String Length, Literals and Comparison

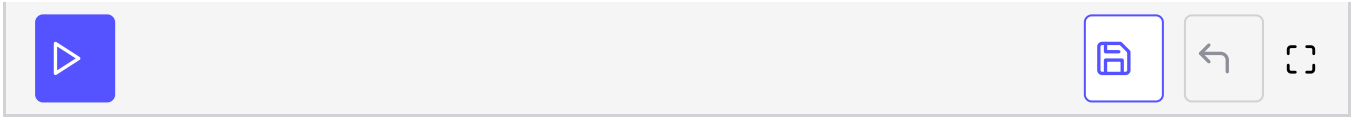
In this lesson, we will discuss potentially confusing length of strings, string literals and comparisons of strings.

WE'LL COVER THE FOLLOWING

Potentially confusing length of strings

We have seen that some Unicode characters are represented by more than one byte. For example, the character 'é' (the latin letter 'e' combined with an acute accent) is represented using at least two bytes in UTF-8 encoding. This fact is reflected in the `.length` property of strings:

```
import std.stdio;

void main() {

    writeln("résumé".length);

}
```



Although `résumé` contains six letters, the length of the string is the number of UTF-8 code units that it contains i.e, 8.
Here `résumé` is a string literal where each element type is a char and each char value represents a UTF-8 code unit. The type of the elements of string literals like "hello" is `char` and each `char` value represents a UTF-8 code unit. This may cause a problem when we try to replace a character using two code

units with the one using a single code unit:

```
char[] s = "résumé".dup;
writeln("Before: ", s);
s[1] = 'e';
s[5] = 'e';
writeln("After : ", s);
```

The two ‘e’ characters do not replace the two ‘é’ characters; they replace single code units of the é character. This results in an invalid UTF-8 encoding:

```
Before: résumé
After : re?sueé    ← INCORRECT
```

```
import std.stdio;

void main() {
    char[] s = "résumé".dup;
    writeln("Before: ", s);
    s[1] = 'e';
    s[5] = 'e';
    writeln("After : ", s);
}
```

When dealing with letters, symbols and other Unicode characters directly, as in the code above, the correct type to use is dchar:

```
import std.stdio;

void main() {

    dchar[] s = "résumé"d.dup;
    writeln("Before: ", s);
    s[1] = 'e';
    s[5] = 'e';
    writeln("After : ", s);

}
```

Please note the two differences in the new code:

1. The type of the string is `dchar[]`.

2. There is a **d** at the end of the literal `"résumé"d`, specifying its type as an array of dchars.

In any case, keep in mind that the use of `dchar[]` and `dstring` does not solve all of the problems of manipulating Unicode characters. For instance, if the user inputs the text “résumé,” we cannot assume that the string length will be 6 even for `dchar` strings. It might be greater if at least one of the ‘é’ characters is not encoded as a single code point but as the combination of an ‘e’ and a combining acute accent. To avoid dealing with this and many other Unicode issues, consider using a Unicode-aware text manipulation library in your programs.

Types of string literals

The optional character specified after string literals determines the type of the elements of the string:

```
import std.stdio;

void main() {
    string s = "résumé"c;   // same as "résumé"
    wstring w = "résumé"w;
    dstring d = "résumé"d;

    writeln(s.length);
    writeln(w.length);
    writeln(d.length);
}
```

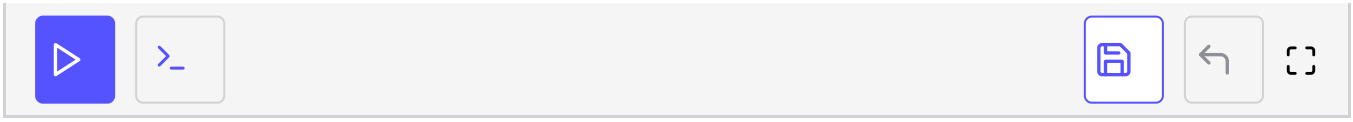
String literals

Because all of the Unicode characters of `résumé` can be represented by a single `wchar` or `dchar`, the last two lengths are equal to the number of characters.

String concatenation

Since they are actually arrays, all of the array operations can be applied to strings as well. `~` concatenates two strings, and `~=` appends to an existing string:

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

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

    // Concatenate:
    string greeting = "Hello " ~ name;

    // Append:
    greeting ~= "! Welcome...";

    writeln(greeting);
}
```



String concatenate and append

Comparing strings

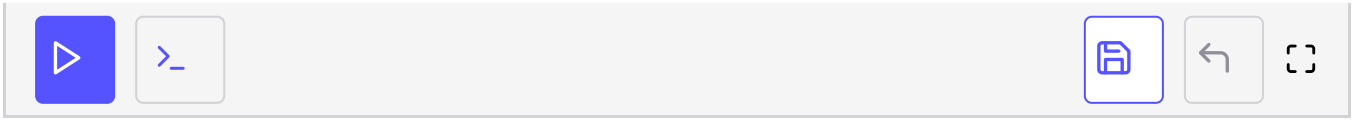
> **Note:** Unicode does not define how the characters are ordered other than their Unicode codes. For that reason, you may get results that don't match your expectations as shown below.

We have used comparison operators `<`, `>=`, etc. with integer and floating point values before. The same operators can be used with strings as well, but with a different meaning: *strings are ordered lexicographically*. This ordering takes each character's code according to its place in the Unicode alphabet. Therefore, the concepts of less than and greater than are replaced with *before* and *after* a character in this hypothetical alphabet:

> **Note:** While entering the strings, press enter when you have written the first string.

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

void main() {
    write("      Enter a string: ");
```

```
    string s1 = strip(readln());

    write("Enter another string: ");
    string s2 = strip(readln());

    if (s1 == s2) {
        writeln("They are the same!");

    } else {
        string former;
        string latter;

        if (s1 < s2) {
            former = s1;
            latter = s2;

        } else {
            former = s2;
            latter = s1;
        }

        writeln("'", former, "' comes before '", latter, "'.");
    }
}
```



String comparison using ==,< and >

Because Unicode adopts the letters of the basic Latin alphabet from the ASCII table, the strings that contain only the letters of the ASCII table will always be ordered correctly.

In the next lesson, you will find a coding challenge to test your understanding of strings in D.