Characters: D's Unicode Problems and Support

In this lesson, we explore problems with reading D's Unicode characters and D's Unicode support.

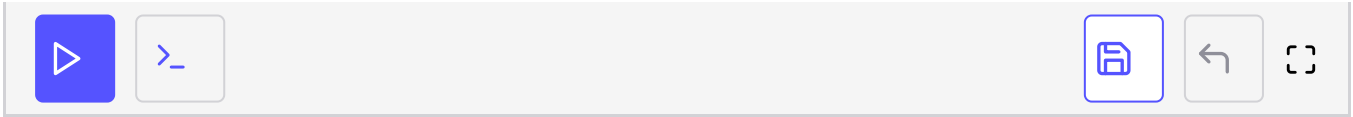
WE'LL COVER THE FOLLOWING

Problems with reading characters

The flexibility and power of D's Unicode characters may cause unexpected results when reading characters from an input stream. This happens due to the multiple meanings of the term character. Before expanding on this further, let's look at a program that exhibits this problem:

```
import std.stdio;

void main() {
    char letter;
    write("Please enter a letter: ");
    readf(" %s", &letter);
    writeln("The letter that has been read: ", letter);
}
```



Reading a character

If you test this program in an environment that does not use Unicode, you may see that all the characters are read and displayed as ASCII.

On the other hand, if you start the same program in an environment with Unicode support (e.g., a Linux terminal), you may see that the character displayed on the output is not the same character that has been entered. To see this, let's enter a non-ASCII character in a terminal that uses the UTF-8 encoding (like most Linux terminals):

```
Please enter a letter: ğ
The letter that has been read:    ← no letter on the output
```

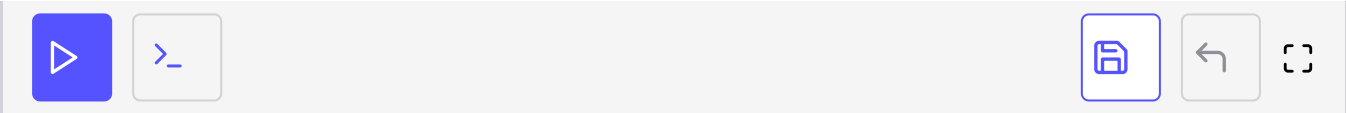
The reason for this problem is that the non-ASCII characters like 'ğ' are represented by two codes, and reading a char from the input reads only the first one of those codes. Since that single char is not sufficient to represent the whole unicode character, the program does not have a complete character to display.
To show that the UTF-8 codes that make up a character are indeed read one character at a time, let's read two char variables and print them one after the other:

```
import std.stdio;

void main() {
    char firstCode;
    char secondCode;

    write("Please enter a letter: "); // Enter a non-ascii character e.g, ğ
    readf(" %s", &firstCode);
    readf(" %s", &secondCode);

    writeln("The letter that has been read: ",
            firstCode, secondCode);
}
```



Reading two characters

The program reads two `char` type variables from the input and displays them in the same order that they are read. When those codes are sent to the terminal in that same order, they complete the UTF-8 encoding of the Unicode character on the terminal, and this time the Unicode character is printed correctly:

correctly.

```
Please enter a letter: ğ
The letter that has been read: ğ
```

These results are also related to the fact that the standard inputs and outputs of programs are char streams.
We will see later in the strings lesson that it is easier to read characters as strings, instead of dealing with UTF codes individually.

D's Unicode support

Unicode is a large and complicated standard. D supports a very useful subset of it. A Unicode-based document consists of the following levels of concepts, from the most fine-grained to the most abstract.

Code unit

The values that make up the UTF encodings are called **code units**. Depending on the encoding and the characters themselves, Unicode characters are made up of one or more code units. For example, in the UTF-8 encoding, the letter 'a' is made up of a single code unit and the letter 'ğ' is made up of two code units.
D's character types `char`, `wchar` and `dchar` correspond to **UTF-8**, **UTF-16** and **UTF-32** code units, respectively.

Code point

Every letter, numeral, symbol, etc. that the Unicode standard defines is called a **code point**. For example, the Unicode code values of 'a' and 'ğ' are two distinct code points.

Depending on the encoding, code points are represented by one or more code units. As mentioned above, in the UTF-8 encoding 'a' is represented by a single code unit, and 'ğ' is represented by two code units. On the other hand, both 'a' and 'ğ' are represented by a single code unit in both UTF-16 and UTF-32 encodings.
The D type that supports code points is `dchar`, whereas `char` and `wchar` can only be used to store code units.

Character

A **character** is any symbol that the Unicode standard defines and what we call “character” or “letter” in daily talk. Characters can be made up of more than one codepoint and are basic elements of textual information.
This definition of character is flexible in Unicode, which brings a complication. Some characters can be formed by more than one code point. For example, the letter ‘ğ’ can be specified in two ways:

- as the single code point for ‘ğ’
- as the two code points for ‘g’ and ’ ˘’ (combining breve)

Although they would mean the same character to a human reader, the single code point ‘ğ’ is different from the two consecutive code points ‘g’ and ’ ˘’.

Summary

- Unicode supports all characters of all writing systems.
- `char` is for UTF-8 encoding; although it is not suitable to represent characters in general, it supports the ASCII table.
- `wchar` is for UTF-16 encoding; although it cannot represent all characters in general, most letters of multiple alphabets can be supported.
- `dchar` is for UTF-32 encoding; since it has 32 bits, it can represent all characters of multiple alphabets.

In the next lesson, we will start exploring strings in D.