Working with diacritics in C++

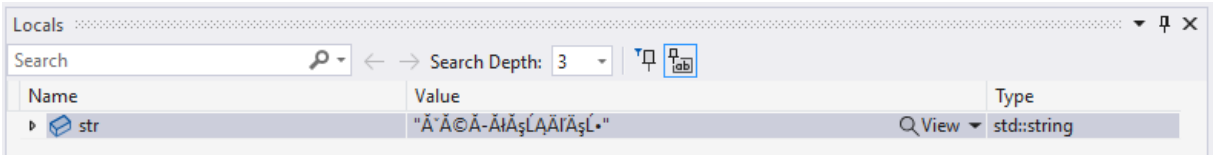
String representation

The C++ standard library defines a class template `std::basic_string<CharT>` for string representation. The `CharT` parameter specifies the type that the class uses internally to represent characters. The most well-known instance of the template is the `std::string` class for which `CharT = char`.

The `std::string` class does not assume or work with any encoding. It is only used to store a string representation as an array of characters. One element of such an array is called *code unit*. However, a single *code point* (e.g., a single character of the Unicode table, see Lecture 1) can be represented by more than one *code unit*. However, methods of class `std::string` such as `size` or `substr` and (almost) all other algorithms from the standard library work only with the given array, regardless of the actual stored string. We can see the effect of this behavior in the following example, where the `str` variable stores a string in the **UTF-8** encoding consisting of 9 characters (*code points*). However, the length of the string from the `std::string` point of view (the number of *code units*) is 18.

```
1 std::string str("áéíóúťľĺŕ");
2 std::cout << str.size() << "\n"; // 18
```

Another practical consequence is viewing the values of variables using the debugger. The variable `str` can be displayed by the debugger in the *Visual Studio* environment as follows. Instead of the 9 Unicode characters listed in the code, we see the values of the individual *code units* interpreted in the Latin 2 encoding that was currently set in Windows.



When processing strings in encodings where one *code point* may correspond to multiple *code unit*, it may, therefore, be necessary to use encoding-specific functions, e.g., to obtain size or substring.

Input and output

In addition to the encoding of strings within your program, you also need to take into account the encoding used in the "outside world" with which your program interacts. Typically, this is the encoding used by the terminal and the encoding of the files from which we read data. Especially in the case of the terminal, the default setting will depend on the operating system.

Linux, macOS

The terminal works in UTF-8 encoding on these systems, so we recommend that your programs work in this encoding as well. To store strings, it is convenient to use the `std::string` class. For input and output, just use the tools you are used to, such as `std::cout`, `std::iftream`, and
mintinlinecppstd::getline. Input data files should be converted (if they are not already) to UTF-8 encoding. Since for UTF-8 encoding, a single *code point* can be represented by multiple *code unit*, the above implications need to be taken into account.

Windows

The Windows terminal uses *Code Page 852 (Latin 2)* encoding (or, possibly, some other code page, depending on the system language). You can verify this by typing `chcp` in the command prompt. This encoding is an extended version of the ASCII encoding (see Lecture 1). This means that one *code point* is represented by one *code unit*, so the aforementioned problems do not arise.

To work correctly with diacritics, the input data files need to be converted (if they are not already) to *Code page 852 (Latin 2)* encoding. To store strings, it is convenient to use the `std::string` class. For input and output, just use the tools you are used to, e.g. `std::cout`, `std::iftream`, and `std::getline`.

Another (similar) option is to use another extended version of the ASCII table, namely *Code Page 1250 (Windows-1250)*. In this case, at the beginning of the `main` function, the console needs to be set to this encoding as follows:

```
#include <Windows.h>

auto main () -> int
{
    SetConsoleOutputCP(1250);
    SetConsoleCP(1250);

    // ...
}
```

Next, you need to convert (if not already) the input data files to *Code Page 1250 (Windows-1250)* encoding.

Third Party Libraries

If you are using third-party libraries (for example, to create a graphical user interface), you need to check the documentation of that library or its components that work with strings to see what encoding they work with. If necessary, you need to convert your strings to the desired encoding at the appropriate places.

String Comparison

The `std::string` class implements the `operator`<, which implements a lexicographic comparison with another string—strings are compared on a character-by-character basis by comparing values of type `char` (i.e., numbers in the interval [-128, 127] or [0, 255]). However, this

behavior is not correct when working with strings containing diacritics. Depending on the encoding used, we get various incorrect results. For example, in the *Windows-1250* encoding, characters with diacritics will be represented by values from the extended part of the ASCII table, i.e., they will always be after the letter `'Z'`. An example of this problem is illustrated in the following code:

```
1  #include <algorithm>
2  #include <iostream>
3  #include <vector>
4  #include <Windows.h>
5
6  int main()
7  {
8      SetConsoleCP(1250);
9      SetConsoleOutputCP(1250);
10
11     std::vector<std::string> names(
12     {
13         "Necpaly",
14         "Údol",
15         "Čabradský Vrbovok",
16         "Ábelová",
17         "Zubrohlava",
18         "Ňagov",
19         "Demänovská Dolina",
20         "Ardovo",
21     });
22
23     std::sort(names.begin(), names.end());
24
25     for (const auto& name : names)
26     {
27         std::cout << name << "\n";
28     }
29 }
```

When sorting strings containing diacritics, it is therefore not possible to use the `operator<` implementation of the `std::string` class. The standard library provides the class `std::locale`, which is (among other things) a function object (implementing `operator()`) that compares strings according to the specified encoding. The function `std::sort` takes the comparator in addition to the iterators of the sorted structure. Therefore, to sort with respecting string encoding, line 23 needs to be replaced on Windows OS as follows:

```
1      std::sort(names.begin(), names.end(), std::locale("Slovak_Slovakia.1250"));
```

and the following call to OS Linux and macOS:

```
1      std::sort(names.begin(), names.end(), std::locale("sk_SK.UTF-8"));
```

When working with another language, it is necessary to use the locale of the respective language, e.g., for Czech `"Czech_Czechia.1250"` or `"cz_CZ.UTF-8"`.