2 A Note on Text Encodings and Proper Display

Written by Prof. Bogusław Cyganek © November 2020

Computers were developed in English speaking countries. Therefore the first character encoding considered only English alphabet which perfectly fits on only few bits. The first character encoding, the so called ASCII contained all these together with digits and many more control and special symbols. However, the world is much more colorful! Therefore I frequently encounter questions such as how to print "Podaj imię" (i.e. Enter your name in Polish) in the terminal window. We will try to shed some light and, more importantly, provide some solutions to this problem and few more. This note is intended to accompany theory and examples presented in my recent book on C++ [8]. Stay tuned…

2.1 Problem Analysis and the First Example

What happens if we initialize the `std::string` or `std::wstring` with a text with national letters? Let's find out, as shown in the following code.

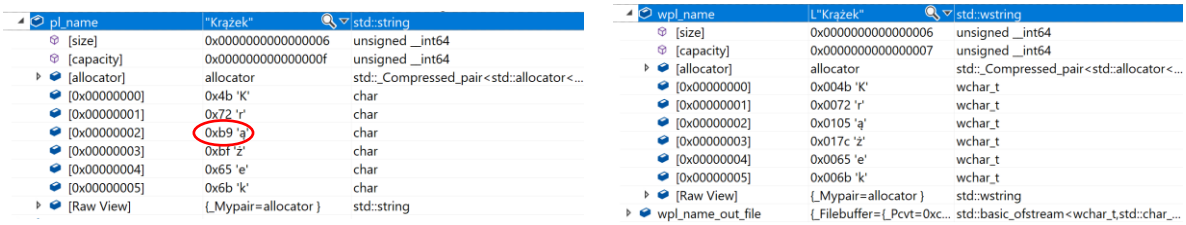
Listing 2-1. Creating two string objects `std::string` and `std::wstring`, both initialized with "Krążek" containing international letters. Both strings are displayed on `std::cout` and `std::wcout`, respectively.

```
1
2
3    // single byte, single letter; plain English letters go ok, other languages
4    // will be encoded, still a letter on a byte ISO-Latin-9 (ISO-8859-15)
5    string s { "Krążek" };              // means a circle or a record
6
7
8    cout << s << endl;
9    cout << s.size() * sizeof( string::value_type ) << endl;
10
11   // This is the wide-char encoded (the same lenght for each char)
12   wstring ws { L"Krążek" };
13
14   wcout << ws << endl;
15   cout << ws.size() * sizeof( wstring::value_type ) << endl;
16
```

The output looks as follows (on my machine with the Windows 10 PL).

```
Kr╣┐ek
6
Kr12
```

Apparently we cannot be content with this. In the second case, even the `endl` did not work as expected. However, before we fix the things up, let's take a look at the character encoding undertaken by the C++ compiler, i.e. "Krążek" in the `std::string` and in the `std::wstring`, respectively. The encodings are shown in Figure 1.



a b

Figure 1. Encoding of "Krążek" in `std::string` (a) and `std::wstring` (b) viewed in the Microsoft Visual's debugger. In the first case each letter occupies a single byte; English letters are the same as in ASCII code, whereas national characters are encoded in Windows-1250. In the second case each character occupies two bytes as indicated by the `wchar_t`. The codes for English alphabet letters are the same as in `std::string`, whereas national letters are encoded differently.

The only one thing we can be sure from the last example is that for a 6 letter text, the `std::string` requires 6 bytes, whereas `std::wstring` 12 bytes, i.e. exactly 2 bytes for *each* letter. It appears that our string s is encoded in the Windows-1250 encoding, shown in Figure 2. For the curious, actually Windows-1250 is *a code page* used on Windows to represent some Central European languages [15][14]. Not surprisingly, it is a default also in our Visual IDE operating on our Windows 10 PL.

Windows-1250																
	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xA	xB	xC	xD	xE	xF
0x	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1x	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2x	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3x	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4x	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5x	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6x	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7x	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL
8x	€	NZ	‚	NZ	„	…	†	‡	NZ	‰	Š	‹	Ś	Ť	Ž	Ź
9x	NZ	‘	’	“	”	•	–	—	NZ	™	š	›	ś	ť	ž	ź
Ax	NBSP	ˇ	˘	Ł	¤	Ą	¦	§	¨	©	Ş	«	¬	SHY	®	Ż
Bx	°	±	˛	ł	´	µ	¶	·	¸	ą	ş	»	Ľ	˝	ľ	ż
Cx	Ŕ	Á	Â	Ă	Ä	Ĺ	Ć	Ç	Č	É	Ę	Ë	Ě	Í	Î	Ď
Dx	Đ	Ń	Ň	Ó	Ô	Ő	Ö	×	Ř	Ů	Ú	Ű	Ü	Ý	Ţ	ß
Ex	ŕ	á	â	ă	ä	ĺ	ć	ç	č	é	ę	ë	ě	í	î	ď
Fx	đ	ń	ň	ó	ô	ő	ö	÷	ř	ů	ú	ű	ü	ý	ţ	˙

Figure 2. Windows-1250 encoding. For example the Polish letter "ą" is encoded as the hexadecimal value B9, as shown in the red circle (from Wikipedia [7][15]).

To obtain a proper display we need to call the **`std::setlocale`** function (from the `#include <clocale>`) with two parameters. First, `LC_CTYPE` denotes a category of the locale to modify, while

the second sets a given country settings. In our case we use `"pl_PL"` to set up the Polish display. Find it out and set it up to your country code (e.g. `de_DE`, `fr_FR`, `en_GB`, `en_US`, `zh_CN`, etc.).

Listing 2-2. To properly display strings with the national characters the locale object is set by calling `std::setlocale`. In this case the Polish locale is chosen be supplying `"pl_PL"`.

```
1   setlocale( LC_CTYPE, "pl_PL" );     // change the encoding of the terminal
2
3   // An encoding default to the IDE will be used - Windows-1250 in our system;
4   // It is a single byte, single letter; plain English letters go ok, other languages
5   // will be encoded, still a letter on a byte
6   string s { "Krążek" };
7
8
9   cout << s << endl;
10  cout << s.size() * sizeof( string::value_type ) << endl;
11
12  // This is the wide-char encoded (the same lenght for each char)
13  wstring ws { L"Krążek" };
14
15  wcout << ws << endl;
16  cout << ws.size() * sizeof( wstring::value_type ) << endl;
17
```

The output looks as follows (on my machine with the Windows 10 PL).

```
Krążek
6
Krążek
12
```

Let's notice that C++ offers a possibility to explicitly encode the constant strings with yet another encoding – UTF-8 – as shown in the following code snippet.

```
18  // We can explicitly ask to encode the text with UTF-8
19  string s8 { u8"Krążek" };    // note the "u8" prefix
20
21  setlocale( LC_CTYPE, "pl_PL.UTF-8" ); // we need to set new locale to reflect UTF-8
22
23  cout << s8 << endl;
24  cout << s8.size() * sizeof( string::value_type ) << endl;
```

Now the output is as follows

```
Krążek
8
```

Let's notice that now we are sure that the inner encoding is UTF-8 which correctly renders on the terminal window if we set the locale to reflect the UTF-8 encoding, as follows `"pl_PL.UTF-8"`. However, the number of bytes used to represent `"Krążek"` in UTF-8 is yet different – it is now 8 bytes.

It is interesting now to take a closer look into this encoding style, as shown in Figure 3. When compared with encodings shown in Figure 1(a) and Figure 1(b) the most striking observation is that the international letters are encoded on 2 bytes, whereas all plain English ones on 1 byte. This means that the number of bytes used for each letter is different! Indeed, in the UTF-8 encoding this length can be from 1 up to 3 bytes.

So, in few lines of code we have already seen as much as 3 different encoding!

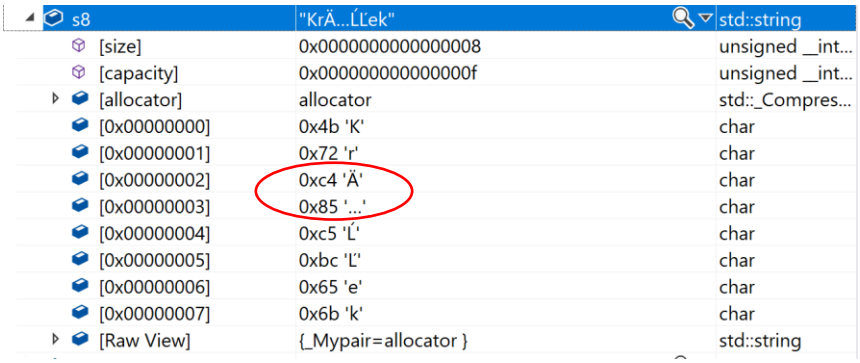


Figure 3. UTF-8 encoding of "Krążek" in `std::string`. The codes for English alphabet letters are 1 byte long and the same as in `std::string`, whereas national letters are encoded on multiple bytes.

Let's conclude this section with few facts:

- There are different character encodings from which the following groups can be distinguished [1][13]:
  - ASCII – the oldest type, used to encode plain English letters and some special characters, each on exactly 7-bits that fit always fit into 1 byte
  - **Wide** character representation – the number of bytes for each character is fixed; usually these are 2 or 4 bytes; to this group the special Windows 1250 encoding can be included
  - **Multibyte** representation in which the number of bytes for each character is variable; English alphabet letters can be encoded all on 1 byte; however these can be followed by 2 or 3 byte encoding of the international letters. An example is the UTF-8 encoding. Multibyte is more compact than the wide encoding, however can be more difficult to process
- `std::wstring`, as well as `std::wcout` process letters in the wide character encoding in which a letter is encoded on 2 bytes, as in our example (sometimes it can be 4 bytes);
- On the other hand, when working with multibyte a better solution is simply to use `std::string` (see Section 2.3.2)
- There is no easy way to decipher a type of the encoding just looking at the bytes representing a string [2]
- To properly display a given encoding we must set the display object; this can be done by calling `std::setlocale( LC_CTYPE, "pl_PL" )`, where `"pl_PL"` or `"pl_PL.UTF8"`, etc. represent the required encoding

Let's notice that the previously seen Windows-1250 is a code page used in Microsoft Windows. It is a version of the more general ISO 8859-1 encoding which assumes always a single-byte per character and that allows to represent the first 256 Unicode characters. All ASCII are encoded exactly the same way. On the other hand, UTF-8 can encode any Unicode character, avoiding the necessity to figure out and set any code page or otherwise indicate what character set we are going to use [15][14].

For better understanding, please read at least [1] and section 16th from [3].

2.2 More on Character Encodings – Dealing with the Files

Let's now make the following experiment: Instead of hard coding "Krążek" to `std::string` or `std::wstring` let's open *Notepad.exe*, directly type in "Krążek", save the file; then try to programmatically open that file and read its text into the `std::string` and `std::wstring` objects,

respectively. The code is shown in the following listing, where our text is stored in a file named *text_pl_test_1.txt*.

Listing 2-3. Reading text to the `std::string` and `std::wstring` objects from a text file *text_pl_test_1.txt* prepared with Notepad. Since Notepad uses the UTF-8 encoding the strings are not properly displayed. To remedy this the locale with `"pl_PL.UTF-8"` encoding needs to be selected.

```
1   string s;
2   wstring ws;
3
4   {
5         // Everything "wide"
6         wifstream     inFile ( L"text_pl_test_1.txt" );
7         inFile >> ws;
8   }
9
10  {
11        // Everything just bytes
12        ifstream      inFile ( "text_pl_test_1.txt" );
13        inFile >> s;
14  }
15
16
17  std::setlocale( LC_CTYPE, "pl_PL" );
18  wcout << ws << endl;
19  cout << s << endl;
```

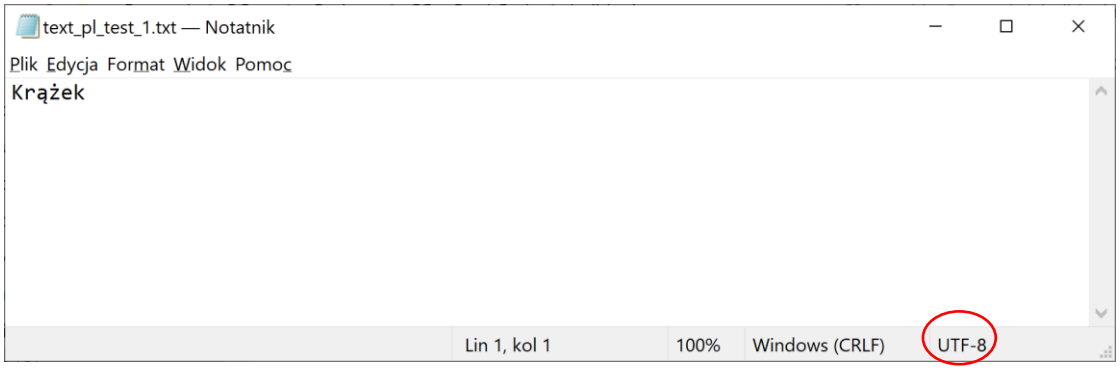
The result of the above code is as follows.

```
KrÄ
KrÄĹĽek
```

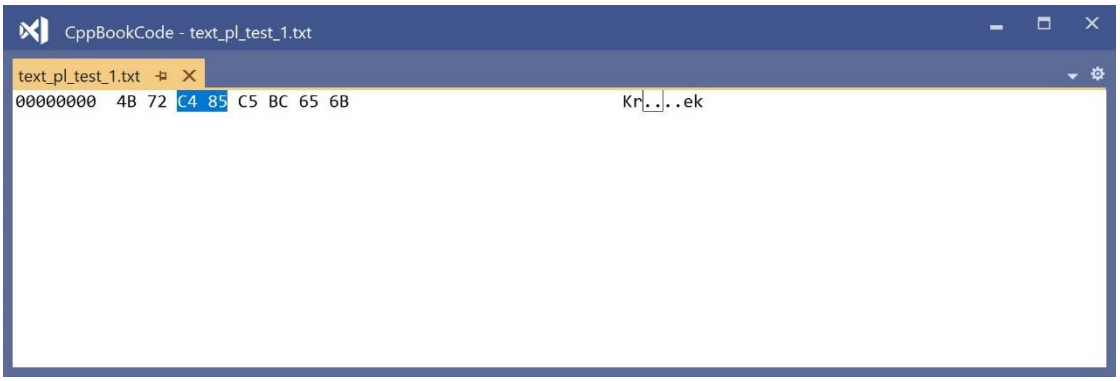
Despite calling `std::setlocale` something is wrong again. What comes to mind is that text in the file is encoded in yet other fashion. Let's analyze the file itself. When we open it we easily notice yet another encoding; this is UTF-8 which name is shown in the Notepad's status bar, as shown in Figure 4(a).

What it means to have text encoded in UTF-8 is revealed in the binary editor shown in Figure 4(b). This is an example of *the multibyte encoding* in which English alphabet letters are encoded each on a single byte and, maybe to our surprise, even with the same encodings as shown in Figure 1. However, the national letters, such as "ą" occupy two bytes, C4 and 85 in this case. This may look as a real nightmare – in few simple examples we encountered as much as three different encodings!

At this point we may ask how to find out type of the encoding? This is another story and the methods are not that obvious. There are many methods, some are based on detection of characteristic preambles or *a byte order mark* (BOM). However, if neither works there are some more esoteric ones, for example based on some heuristics, text statistics, etc., see [2][3][12]. However, in many cases we can ask the tools or look it up in Notepad or Word, as shown in Figure 5.



a



b

Figure 4. A text file with the “Krążek” text directly entered from the keyboard, containing two Polish letters “ą” and “ż” and shown in the Notepad.exe window in the Windows 10 PL OS (a). The assumed encoding is displayed in the lower right corner – here “UTF-8”. Encoding of the same text opened in the binary editor of the Microsoft Visual 2019. Apparently the UTF-8 character encoding is used – some letters occupy 8 bits, whereas the other two or more bytes (such as the special Polish letters).

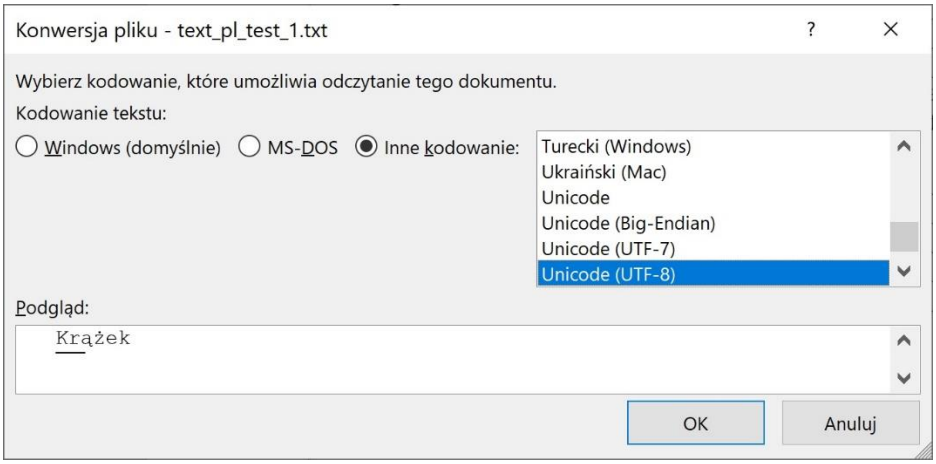


Figure 5. Looking up text encoding from the Microsoft’s Word when opening a text file.

But let’s return to our last example and try to fix the display. We know the type of the encoding of the text read from a file, so to assure a proper display we need to explicitly state the type of the encoding as in the following code snippet.

```
20    std::setlocale( LC_CTYPE, "pl_PL.UTF-8" );
21    wcout << ws << endl;
22    cout << s << endl;
```

Now the output is

```
KrÄ
Krążek
```

Still `std::wstring` is broken since it did not read properly. However, `std::string`, which contains all the required characters but in the UTF-8 encoding, this time renders properly.

Let's go a little bit further and programmatically write back the `std::string` object to a file named *PL_name_1.txt* (`wstring` is broken, anyway). After a successful write and opening it in the system the text renders correctly and, if opened in a binary editor, the character encoding is as shown in Figure 6. This is Windows-1250 encoding scheme, which code table is depicted in Figure 2, and exactly the same as in Figure 1(a).

Figure 6. Encoding of the "Krążek" text obtained after streaming out a `std::string` from the C++ code, opened in the binary editor of the Microsoft Visual 2019. Apparently the Windows-1250 character encoding is used; in this scheme each character is encoded on 8-bits.

So, whether we write `std::string` in UTF-8 or Windows-1250 it renders OK.
Finally, let's try to initialize `std::wstring` and write it directly to a file, as in the following code:

```
wstring ws { L"Krążek" };           // Now we fill
wofstream outFile( L"PL_name_2.txt" );
outFile << ws;
```

Unfortunately, when we open *PL_name_2.txt* only two initial letters "Kr", which follow the plain English encoding, are shown – apparently writing UTF-16 to a file, even with the `wstring` and `wofstream` objects, gives wrong results. Can we fix it? It appears so; let's replace the last two lines with the following ones

Listing 2-4. A way around to write wide strings to a file.

```
std::ofstream outFile( "PL_name_2.txt", std::ios::binary ); // Weird but WORKS ...
outFile.write( (char*) & ws[ 0 ], sizeof( wstring::value_type) * ws.size() );
```

The output displays OK in the Notepad, showing UTF-16 in the status bar. However, let us notice that we have used `std::ofstream` and not `std::wofstream` which won't work.

Nevertheless, having UTF-8 text in `std::string` can be error prone. For example, trying to count the number of elements (are these letters or bytes?)

```
cout << "Num of letters in s: " << s.size() << endl;
```

```
Num of letters in s: 8
```

Apparently this is the number of all bytes (codes) and not the number of letters which in “Krążek” are only 6. On the other hand asking the same from `ws` gives a correct value.

```
Num of letters in ws: 6
```

2.3 Example Project – A Lottery

As a summary of the above analyses let’s show two versions of a simple program whose functionality is as follows:

1. Read the names from a text file; in the following examples it is assumed that names in the text file are encoded with the multibyte UTF-8 encoding, such as the ones written in *Notepad.exe*. However, in other systems other encodings are also possible.
2. Shuffle the names, then draw a single winner;
3. Write a name of the winner to the terminal and to a file;

Two versions will be presented. The first one after reading the names converts them from UTF-8 to the wide version which will be stored in the `std::wstring` objects. Then, after the draw a name of the winner is converted back to UTF-8 and written to the output file. In the second version there is no conversion and all the texts are internally processed in the UTF-8 encoding using the simple `std::string` objects (`std::string` appears preferable if a buffer is needed).

2.3.1 A Version with Encoding Conversion

The first version assumes text encoding conversions done with the help of properly initialized `std::wstring_convert` objects.

Listing 2-5. A first version of the lottery function. Participants’ names are written from the keyboard to a text file. This text file is encoded with the UTF-8 multibyte scheme. Names are then read in, line after line, shuffled and finally a winner is selected. Name of the winner is then printed to the terminal and to an output file. However, before the texts are processed they are converted from the multibyte UTF-8 to the wide 2-byte encoding expected by `std::wstring`.

```
1   void Lottery_1( void )
2   {
3      using VecStr = vector< wstring >;
4      VecStr            names;
5
6      // ----------------------------------------------------
7      // multibyte str to wide converter
8      // std::wstring_convert is already deprecated since C++17
9      // however, there is no good alternative yet ...
10     auto mul_2_wide = [] ( const auto & m_str )
11     {
12         return std::wstring_convert<
13                std::codecvt_utf8< wchar_t > >{}.from_bytes( m_str );
14     };
15
16     // a file with names of all participants
17     std::ifstream inFile( "names.txt" );
18
19     // Read line by line in the multibyte format converting to the wide
20     for( string s; std::getline( inFile, s ); )
21         if( s.size() > 0 )
22                names.emplace_back( mul_2_wide( s ) );
23
24
```

```
25      std::wcout << L"There are " << names.size() << L" participants\n";
26      if( names.size() == 0 )
27          return;
28
29
30      // ------------------------------------------------
31      // Make the draws
32      std::mt19937 rand_gen( ( std::random_device() )() );
33
34      // Now shuffle everything couple of times
35      for( auto i : CppBook::range( 10 ) )
36          std::shuffle( names.begin(), names.end(), rand_gen );
37
38      // values between from_val and to_val inclusive
39      std::uniform_int_distribution<VecStr::size_type> uni_distr( 0, names.size()-1 );
40
41      // ***************************
42      // choose the right encoding , e.g. fr_FR, de_DE, zh_CN, ko_KR, en_US, etc.;
43      const auto kIntEncoding { "pl_PL" };
44      setlocale( LC_CTYPE, kIntEncoding );    // change the encoding of the terminal
45      // ***************************
46
47      const auto & theWinner { names[ uni_distr( rand_gen ) ] };
48      std::wcout << L"And the WINNER is... >>> " << theWinner << L" <<< !\n";
49
50
51      // ------------------------------------------------
52      // Finally, let's write the name of our winner
53      // However, we cannot simply use << on a ofstream,
54      // before we need to convert back to the multibyte representation
55
56      using W2M_Facet = std::codecvt_byname< wchar_t, char, std::mbstate_t >;
57      std::string mb_theWinner =      std::wstring_convert< W2M_Facet >
58                          { new W2M_Facet( kIntEncoding ) }.to_bytes( theWinner );
59
60      std::ofstream outFile( "winner_is.txt" );
61      outFile << mb_theWinner;
62  }
```

On lines [10-14] the `mul_2_wide` lambda function is defined which does the multibyte to wide characters conversion[1]. On line [20] the names are read line by line by `getline` from the byte-oriented stream object initialized on line [17]. Then, on line [22], each name is first converted by `mul_2_wide`, and then emplaced on the consecutive positions in the `names` object; each name with wide characters is represented with `std::wstring`. Next, the names are shuffled on line [35] with help of the `CppBook::range` object set to 10 iteration (explained in the book [8]), and a winner is finally drawn on line [47]. However, for proper display on line [43-44] the encoding of the terminal window is set to `"pl_PL"` and on line [48] a wide character name of the winner is printed out.

To write out the name to a text file, on lines [57-58] we convert it back to the multibyte UTF-8 with help of the facet object aliased on line [56]. This is then output in lines [60-61] to the byte-oriented file object represented with the `std::ofstream` typed object. However, we could avoid this conversion and write the wide object as shown in Listing 2-4. Nevertheless, in such a case the resulting text file would store the letters in the UTF-16 format.

The group of the `codecvt` related classes convert between character encodings, such as UTF-8, UTF-16, UTF-32. `std::codecvt_byname` is a `std::codecvt` facet that encapsulates multibyte-wide character conversion rules based on a locale provided in its constructor [10].

Let's try it out. After opening *Notepad.exe* we write down couple of names, many containing Polish letters, as follows

---

[1] Note that `std::codecvt_utf8` has been deprecated since C++17 [11]. Alternatively, see the libraries such as *utfcpp* [12].

```
Bogusław Cyganek
Ala Żarnowska
Michał Barański
Ola Wójtowicz
Grzegorz Brzechwa
Andrzej Boruta
```

After running `Lottery_1` the following is displayed

```
There are 6 participants
And the WINNER is... >>> Michał Barański <<<!
```

Additionally, *winner_is.txt* contains the following

in the ANSI encoding which is default in my system. However, if on line [58] we change the `kIntEncoding` encoding to `"pl_PL.UTF-8"`

then we have file written in the UTF-8. Certainly, we will have different winners from run to run. Note that std::codecvt_utf8

2.3.2 A Version with the UTF-8 Multibyte Processing

This is a slightly simpler version in which no wide related objects, such as `std::wchar` or `std::wcout`, are used. Also, there are no conversions and all the strings are internally represented with the byte oriented `std::string` objects containing UTF-8 multibyte encodings.

Listing 2-6. A version of the lottery function with code encoding transformations. Participants' names are written from the keyboard to a text file. This text file is encoded with the UTF-8 multibyte scheme. Names are then read in using `std::string`, line after line, shuffled and finally a winner is selected. Name of the winner is then printed to the terminal and to an output file. No text conversions.

```
1    void Lottery_2( void )
2    {
```

```
3       using VecStr = vector< string >;        // simply std::string
4       VecStr             names;
5
6       // a file with names of all participants
7       std::ifstream inFile( "names.txt" );
8
9       // Read line by line in a multibyte format
10      for( string s; std::getline( inFile, s ); )
11          if( s.size() > 0 )
12                 names.emplace_back( s );
13
14
15      std::cout << "There are " << names.size() << " participants\n";
16      if( names.size() == 0 )
17          return;
18
19      // ------------------------------------------------------
20      // Make the draws
21      std::mt19937 rand_gen( ( std::random_device() )() );
22
23      // Now shuffle everything couple of times
24      for( auto i : CppBook::range( 10 ) )
25          std::shuffle( names.begin(), names.end(), rand_gen );
26
27      // values between from_val and to_val inclusive
28      std::uniform_int_distribution<VecStr::size_type> uni_distr( 0, names.size()-1 );
29      const auto & theWinner { names[ uni_distr( rand_gen ) ] };
30
31      // ------------------------------------------------------
32      // Finally, let's write the name of our winner
33
34      setlocale( LC_CTYPE, "pl_PL.UTF8" ); // change the encoding of the terminal
35      std::cout << "And the WINNER is... >>> " << theWinner << " <<< !\n";
36
37      std::ofstream outFile( "winner_is.txt" );
38      outFile << theWinner;
39  }
```

Let's observe that on line [34] we need to set slightly different locale to accommodate not only `pl_PL` but also the UTF-8 encoding. The results are the same as when running `Lottery_1` from Listing 2-5.

```
There are 6 participants
And the WINNER is... >>> Ala Żarnowska <<< !
```

Indeed `Lottery_2` is shorter and simpler than `Lottery_1` from Listing 2-5. However, let's notice that this happens because we are not doing any operations on single letters, which in the case of the UTF-8 multibytes would be much more cumbersome due to the various number of bytes for different characters. For example, the following works fine in the wide characters version

```
40      std::cout << "len = " << theWinner.length() << std::endl;
41      std::transform(       theWinner.begin(), theWinner.end(), theWinner.begin(),
```

```
42                          []( auto & c ){ return std::towupper( c ); } );
```

whereas the seemingly equivalent code in the multibyte version does not

```
43    cout << "len = " << theWinner.length() << endl;
44    std::transform(       theWinner.begin(), theWinner.end(), theWinner.begin(),
45                          []( auto & c ){ return std::toupper( c ); } );
```

For more on this and related problems refer to [9], as well as to [16] and references therein.

2.4 More Locales

When writing about locales let's observe that we can adjust our framework to do more than properly display the national characters. For instance we can set a local format of displaying time and date or numerical values, as in the following example.

Listing 2-7. Changing locales to print numeric values with local facets.

```
1     double price { 13.45 };
2
3     cout << "price = " << price << endl;
4
5     // turn to the Polish system (fractions separated with comma ,)
6     cout.imbue( locale( "pl_PL" ) );
7
8     cout << "price = " << price << endl;
9
10    // turn back to the default (use "C" or be more precise and set e.g. "en_EN")
11    cout.imbue( locale( "C" ) );
12
13    cout << "price = " << price << endl;
```

To set a local style for numeric values on line [6] the `imbue` function is called on behalf of the `cout` object. Its parameter is the new `std::locale` object, this time set for Polish style. Thanks to this, a fractional part is separated with comma rather than a dot used in the English speaking countries. To return back to a default setting on line [11] `imbue` is called again, this time with the locale set to **"C"**, meaning a default locale.

2.5 Conclusions

There is a real mess in encoding! Once we go slightly out of the plain English texts we can be bogged down for real. However, based on the above analysis and examples let's try to draw some conclusions. These are as follows:

- To represent international characters there are many encodings possible. These can be split into multibyte (such as UTF-8), in which each character can consume a different number of bytes 1, 2, 3 or even 4, and the wide character encoding, which assumes the same number of bits for each character and this is either 16 or 32 bits
- In the code with international characters either use UTF-8 with only `std::string` everywhere and avoid `std::wstring`
- or go with the wide encoding, consequently using `std::wstring` in the code, converting it to and from other encodings when necessary for IO operations

The above rules have different 'color' when discussing particular OS. Probably the first solution is preferable if you stick to Linux, whereas the second one may be more appropriate if on Windows [5][12].

The third conclusion relates display

- For proper display in a terminal use the `std::setlocale`

Have fun and avoid the mess with character encodings!

References

[1] https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/
[2] http://architectshack.com/TextFileEncodingDetector.ashx
[3] Josuttis N.: *The C++ Standard Library*. 2nd edition, Addison-Wesley, 2012
[4] https://stackoverflow.com/questions/402283/stdwstring-vs-stdstring
[5] http://utf8everywhere.org/
[6] https://en.wikipedia.org/wiki/UTF-8#Compared_to_UTF-16
[7] https://en.wikipedia.org/wiki/Wide_character
[8] Cyganek B.: *Introduction to Programming with C++ for Engineers*. Wiley, 2020.
[9] https://stackoverflow.com/questions/36897781/how-to-uppercase-lowercase-utf-8-characters-in-c
[10] https://en.cppreference.com/w/cpp/locale/codecvt_byname
[11] https://codingtidbit.com/2020/02/09/c17-codecvt_utf8-is-deprecated/
[12] https://github.com/nemtrif/utfcpp
[13] http://www.unicode.org/
[14] https://stackoverflow.com/questions/7048745/what-is-the-difference-between-utf-8-and-iso-8859-1
[15] https://en.wikipedia.org/wiki/Windows-1250
[16] Corentin J.. P2020R0: Locales, Encodings and Unicode. https://wg21.link/p2020r0, 2020.