How Perl Added Unicode Support 10 Years Ago Without You Noticing It

TOBI OETIKER

Tobi Oetiker is the author of several wel-known open source applications: MRTG, RRDtool, and SmokePing. He co-owns and works at Oetiker+Partner AG, a consultancy and development company in Olten, Switzerland. Tobi's current pet open source projects are extopus.org, a tool for integrating results from multiple monitoring systems into a cool JavaScript-based Web frontend, and remOcular.org, a slick, interactive command line-to-Web converter. Read more from tobi on @oetiker, G+ or tobi.oetiker.ch
tobi@oetiker.ch

Imagine that your new German customer sends you a ton of text files that you have to add to your document database. You write a Perl script which neatly imports all the data into your shiny new PostgreSQL database. As you tell this to your DBA, she wonders what had happened to all the German ä, ö, ü umlauts and ß characters in the process. You had not suspected that there might be a problem, but, as you look, all is well—this, despite the database being UTF-8 encoded while the German text files were seemingly normal text files. Another shining example of Perl doing exactly what you want even when you don't know what you are doing.

All seems well, that is, until someone from accounting notices that all the Euro symbols (€) have been turned into ¤ symbols. That's when you start digging into how this really works with Perl and character encodings and Unicode.

A Short Introduction to Unicode

Back in the '60s, the American Standard Code for Information Interchange (aka ASCII) had become the lingua franca for encoding English text for electronic processing outside the IBM mainframe world.

As the use of computers spread to other languages, the whole encoding business became a jumbled mess. The vendors, as well as some international standardization bodies, fell over each other to come up with sensible ways of encoding all the extra characters found in non-English languages. Each language or group of similar languages got one or several encodings. In Western Europe, the Latin1, or ISO-8859-1, encoding became popular in the '80s and '90s. It sported all the characters required to write in the Western European languages.

Working with a single language, this was fine, but as soon as multiple languages were in play, it all became quite confusing; data had to be converted from one encoding to another, often losing information as some symbols from encoding A could not be represented in encoding B.

In the late '80s, work had begun to create a single universal encoding, capable of encoding text from all the world's languages in a unified manner. In 1991 the Unicode consortium was incorporated, and it published its first standard later that year. The current version of the standard is Unicode 6.0, published in October 2010. It covers 109,000 symbols from 93 different scripts. Each symbol is listed with a visual reference, as well as a name made up from ASCII letters, and is tagged with properties giving additional information as to the character's purpose.

With its huge number of characters, Unicode requires multiple bytes to store each symbol. A pretty wasteful undertaking, when you recall that most languages written in Latin script will require only about 70 different symbols to get by. Therefore, a number of different Unicode "encoding" schemes were proposed over time. These days the UTF-8 and UTF-16 schemes are the most popular. Both are variable length encodings, where symbols will use a varying number of bytes to be stored.

UTF-16 is the "native" encoding used by Microsoft operating systems since Windows 2000. It requires at least 2 bytes per character. UTF-8 is the primary encoding used in most Internet-based applications and also in the UNIX/Linux world. Its main feature is that it encodes all of the original 7-bit ASCII characters as themselves. This means that every US-ASCII encoded document is equivalent to its UTF-8 counterpart. All other Unicode characters are encoded by several bytes. The encoding is arranged such that most of the extra symbols required by Western European languages end up as 2-byte sequences.

These days UTF-8 is widely used. XML documents, for example, are encoded in UTF-8 by default. A lot of the Web content is encoded in UTF-8, and most Linux distributions use UTF-8 as their default encoding.

Perl Unicode Basics

In July 2002, with the release of Perl 5.8, Unicode support was integrated into the language. Since everything was done in a nicely backward-compatible manner, the six lines under the "Better Unicode Support" heading in the release announcement went largely unnoticed. The good news is that, in the meantime, most people are actually using Perl 5.8 or a later version, so all the information in this article should be readily applicable in your real-life Perl setup.

The fundamental idea behind Perl's Unicode support is that every string is stored in Unicode internally. If the string consists only of characters with code points (numeric IDs) lower than `0x100`, the string is stored with one byte per character. Since the Unicode code points `0x0` to `0xff` are equivalent to the Latin1 character set, nothing much changed to the casual observer. If any characters with code points `0x100` or higher are present, the string is UTF-8 encoded and flagged appropriately.

On IBM mainframes, Perl uses the EBCDIC encoding, and thus a matching UTF-EBCDIC was chosen to go with it. For the purposes of this article, if you are using Perl on a mainframe just think "EBCDIC" when you read "Latin1."

Perl Unicode Internals

So the main new thing is that now strings can be stored either as sequences of characters with one byte per character or as UTF-8 encoded character sequences with a flag. The perlunifaq(1) suggests that you not even think about all these things, pointing out that they will just work. This is largely true, but I found that until I understood what was happening internally, I kept running into interesting corner cases, driving me nuts. So here is your chance to get your mental picture cleaned up as well.

But, as the perlunifaq suggests, the functions shown in the following examples are not normally required in everyday tasks.

uft8::is_utf8($string) tells whether the UTF-8 flag is set on a string or not:

```
#!/usr/bin/perl
my %s = (
  latin1    => chr(228), # latin1 ä;
  utf8      => chr(195).chr(164), # utf8 encoded ä char
  smiley    => "\x{263A}", # unicode smiley
);
for (keys %s){
       print "$_: >".utf8::is_utf8($s{$_})."< $s{$_}\n";
}
```

Terminals set to work in Latin1 encoding, will show:

```
latin1: >< ä
Wide character in print at p1.pl line 8.
smiley: >1< â☺º
utf8: >< Ã¤
```

Terminals running in UTF-8 mode will display:

```
latin1: ><
Wide character in print at p1.pl line 8.
smiley: >1< ☺
utf8: >< ä
```

The presentation of the characters is entirely up to the terminal, hence the different rendering. Perl assumes that your output device is in Latin1 single-byte mode and warns that it will have trouble displaying the smiley character, which has no equivalent representation in Latin1.

The example also shows that Perl will keep strings in single-byte mode unless it is forced into UTF-8 encoding by the content of the string. Also, the UTF-8 encoded string is not automatically recognized as such.

A few more functions help to get things sorted. The `utf8` namespace holds a bunch of utility functions that allow you to move strings between encodings:

utf8::upgrade($string) in-place converts from single byte to UTF-8 encoding while setting the UTF-8 flag. If the UTF-8 flag is already set, this is a no-op.

utf8::downgrade($string[, FAIL_OK]) in-place converts from UTF-8 to single byte while removing the UTF-8 flag. If the logical character sequence cannot be represented in single byte, this function will die unless FAIL_OK is set.

utf8::encode($string) in-place converts from internal encoding to a byte-sequence UTF-8 encoding, and removes the UTF-8 flag in the process. Since Unicode strings are internally represented as UTF-8 already, all this really does, is remove the UTF-8 flag from a string.

utf8::decode($string) checks if a single-byte (non-encoded) string contains a valid UTF-8 encoded character sequence and sets the UTF-8 flag if this is the case.

```
#!/usr/bin/perl
my %s = (
  latin1  => chr(228), # latin1 ä;
  utf8    => chr(195).chr(164), # utf8 encoded ä char
```

```
  smiley  => "\x{263A}", # unicode smiley
);
utf8::upgrade($s{latin1}); # latin1 ⟶ internal utf8
utf8::decode($s{utf8}); # set the utf8 flag
for (keys %s){
       print "$_: >".utf8::is_utf8($s{$_})."< $s{$_}\n";
}
```

A UTF-8 terminal now shows:

```
latin1: >1<
Wide character in print at p1.pl line 12.
smiley: >1< ☺
utf8: >1<
```

All strings are in UTF-8 mode internally, so all should be well, but only the smiley character gets printed; the ä character is lost. The Latin1 terminal, on the other hand, shows:

```
latin1: >1< ä
Wide character in print at p1.pl line 12.
smiley: >1< â˜º
utf8: >1< ä
```

The reason for this effect is Perl assuming that our output device (STDOUT) is working in single-byte mode. Perl is "doing what you want" by encoding all the output strings into Latin1, and only if there is no Latin1 representation will it resort to UTF-8 native encoding. This leads to the question of how to tell Perl about the encoding in use on `STDOUT`. The `binmode` function helps:

```
#!/usr/bin/perl
binmode(STDOUT,':utf8');
my $smile = "\x{263A}";
print "$smile\n";
```

When running in an UTF-8 enabled terminal you now get properly encoded data *and Perl will also not complain about wide characters anymore*:

```
☺
```

While this works fine if you are running on an UTF-8 terminal, it would not work well for sites still running in Latin1 mode. Normally the `LANG` environment variable gives an indication as to the encoding in use on the system. If you want Perl to take this into account, you can use the `open` pragma and the `:locale` encoding:

```
#!/usr/bin/perl
use open ':locale';
my $umlaut = chr(228);
utf8::upgrade($umlaut);
print "$umlaut\n";
```

which will always output an "ä", taking the default encoding into account when reading and writing data on the system.

When interpolating a string containing material with the UTF-8 flag set (the smiley gets an automatic UTF-8 promotion due to its content, which cannot be

represented in a single-byte encoding), then the resulting string will be upgraded to UTF-8 mode as well:

```
#!/usr/bin/perl
use open ':locale';
my $umlaut = chr(228);
my $smile = "\x{263A}";
print "$umlaut $smile\n";
```

Running this on a Latin1 system gives:

```
"\x{263a}" does not map to iso-8859-1 at p3.pl line 5.
ä \x{263a}
```

If you want to write your Perl scripts in UTF-8 encoding, you can use the UTF-8 pragma to tell Perl about this.

```
use utf8; # assume utf8 program text
no utf8; # assume native program text
```

Note, though, that this only affects how Perl treats the text of the program, so it will understand an UTF-8 encoded "ä" in the program text, but it will still store it in native encoding internally. The UTF-8 flag will only be set on strings that do contain Unicode characters with code points above 0xff.

When a string has the UTF-8 flag set, all string handling functions will continue to work in an intuitive manner, meaning they will act on characters and not on bytes. This might cause some interesting side effects, as the length command will, for example, not tell you anymore how many bytes are in a string, but how many characters. Using the bytes pragma, you can force Perl to still look at the bytes and not at the characters:

```
#!/usr/bin/perl
my $umlaut = "ä";
print 'plain: ',length($umlaut),"\n";
utf8::upgrade($umlaut);
print 'utf8: ',length($umlaut),"\n";
use bytes;
print 'byte length:', length($umlaut),"\n";
```

As expected the UTF-8 version of the string uses 2 bytes of memory.

```
plain: 1
utf8: 1
byte length:2
```

PerlIO and the Encoding Module

The real fun begins when interacting with data from outside the program. The PerlIO layer goes a long way toward making this process as simple as possible. It allows you to do elaborate data processing steps as you are working with file handles. Using the three-argument open syntax and an appropriate PerlIO layer definition is all it takes:

```
#!/usr/bin/perl
open my $fs, '<:encoding(latin15)','euro-test.txt';
my $data = <$fs>;
print 'utf8 flag: ',utf8::is_utf8($data),"\n";
```

With this setup, PerlIO takes care of decoding the Latin15 encoded input file and stores the result in UTF-8 mode internally:

```
utf8 flag: 1
```

Latin15 is another name for the ISO-8859-15 encoding. It is the single-byte encoding commonly used in Western Europe these days. Latin15 is very similar to the classic Latin1 encoding, but a few characters have been replaced. Most importantly, Latin15 includes the € (Euro) symbol.

Using the `binmode` command, the encoding of an open file handle can be changed:

```
#!/usr/bin/perl
open my $fs, '<','euro-test.txt';
binmode($fs,':encoding(latin15)');
my $data = <$fs>;
```

The open pragma allows you to define the default encoding for commands creating file handles:

```
#!/usr/bin/perl
use open IN => ':encoding(latin15)', OUT=>':utf8';
open my $fs, '<','euro-test.txt';
my $data = <$fs>;
```

The encoding and decoding process can also be controlled directly by using the Encode module:

```
#!/usr/bin/perl
use Encode;
$x = decode('iso-8859-1', chr(228));
print 'flag: ',utf8::is_utf8($x),
       ' - ',encode('utf8',$x),"\n";
```

The decode step turns the "ä" character in Latin1 encoding into a Perl UTF-8 character sequence with the UTF-8 flag set. The encode step turns it into a UTF-8 multi-byte sequence without the UTF-8 flag set. A Latin1 terminal will show:

```
flag: 1 - Ã¤
```

The Unicode Bug

The Unicode standard not only defines code points (numeric IDs) for its characters, but it also provides properties such as *Letter, Number, Uppercase_Letter, Space_Separator* . Perl has access to this information and can use it in regular expressions and other commands. The example below demonstrates the behavior of the "\w" (word characters) regular expression match:

```
#!/usr/bin/perl
my $a = 'aäa';
$a =~ /(\w+)/ and print "standard match: $1\n";
utf8::upgrade($a);
$a =~ /(\w+)/ and print "utf8 match: $1\n";
```

The result is a bit odd:

```
standard match: a
utf8 match: aäa
```

While Perl uses Unicode for its internal strings, it seems to use the Unicode meta-information only when the string is in UTF-8 encoding. The "\w" does not match "ä". This can be fixed by using the good old "locale" module, but that is from a time when Perl did not assume all strings to be in Unicode.

This behavior has become known as the "Unicode Bug"; it has been present for quite some time and therefore could not just be fixed without breaking existing code. Perl 5.12 therefore introduced the `unicode_strings` feature, which "fixes" the bug:

```
#!/usr/bin/perl-5.12.0
use feature 'unicode_strings';
my $a = 'aäa';
$a =~ /(\w+)/ and print "standard match: $1\n";
utf8::upgrade($a);
$a =~ /(\w+)/ and print "utf8 match: $1\n";
```

Now Perl uses the knowledge from the Unicode standard in all cases and the result looks fine:

```
standard match: aäa
utf8 match: aäa
```

But also note that the bug does not affect you if you are using UTF-8 encoded and flagged character strings, the format you would end up with when using the PerlIO layer or the Encoding module functions.

CPAN and Unicode

When using CPAN modules, make sure to check their documentation for Unicode support.

The DBD module for PostgreSQL (`DBD::Pg`), for example, will return all string data with the UTF-8 flag set and properly decoded, unless the database encoding is set to `SQL_ASCII`. You can use the `pg_utf8_strings` flag to override the automated decision. The MySQL DBD module can deal with UTF-8 encoded databases, but you have to tell it explicitly by setting the `mysql_enable_utf8` flag on the database handle.

`XML::LibXML` will also work with UTF-8 characters without further ado.

Mojolicious, Dancer, and other new kids on the CPAN block will, in general, work graciously with Unicode. The only problem I ever ran into was that I was incorrectly encoding data for output which had already been encoded by the framework, ending up with doubly encoded data.

About That Missing Character...

And now, on to resolving the mystery from the beginning of this article. The German text files were in Latin15 encoding, which is pretty similar to the Latin1 encoding Perl uses by default except for the Euro sign (and some other bits). Your script read the text in as if it was Latin1 encoded. As the data went via `DBI` into PostgreSQL, the `DBD::Pg` module took care of properly UTF-8 encoding the data, which worked fine except for the Euro sign, which is not in the Latin1 character set. The fix for the problem is simple, though: the text files have to be opened with the `:encoding(Latin15)` PerlIO layer and it all works.

Recap

The road to Perl Unicode bliss:

- Be aware that Perl internally treats everything as Unicode (and make sure to keep all text information encoded in UTF-8 with the UTF-8 flag set to avoid the Unicode Bug).
- Whenever data enters the program from the outside, *decode* it from its outside encoding.
- When data leaves the program, *encode* it according to the requirements of the next step of processing.
- Consider that the modules you are using to access data might already be taking care of all (or part) of the encoding and decoding business.

For further entertainment have a look at the Perl Unicode documentation on http://perldoc.perl.org/.