Organizing character and code sets

ALPHABET SOUP

When foreign characters occur in program code or data, Perl programmers need a solution that avoids the tribulations of Babel.

BY MICHAEL SCHILLI

In the beginning was the ASCII table – 128 characters that let users compose English-language texts, including a couple of foreign characters that were on any typewriter, such as % or $, and of course a couple of control characters, such as line break, page feed, or the bell. It was just a matter of time until non-English speakers started looking for ways to add the accented characters and umlauts their native languages needed, and the first approach was to squash them into the next group of 128 characters. All 256 characters were numbered 0 through 255 and encoded on computers with 8 bits (1 byte) of data.

This was the birth of the ISO 8859 standard (also known as Latin 1).

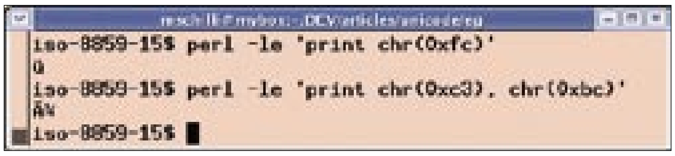
Different Languages

It all started with ISO-8859-1, but over the course of time other variants were added until the current count reached ISO-8859-15, which also includes the Euro character. Incidentally, most of today's web browsers do not use the ISO-8859-1 standard to decode ISO-8859-1 content sent by web servers. Instead, the browsers rely on the Windows-1252 standard, which includes a couple of extra characters, such as the Euro character. Of course, the rest of the world didn't want to get left behind, and the race to display far more complex character sets was on.

Encoding schemes for Asian languages, such as Shift-JIS and BIG5, were invented. But developers soon realized that this was getting them nowhere, and Unicode, an enormous table containing all the characters of common languages in the world, was invented.

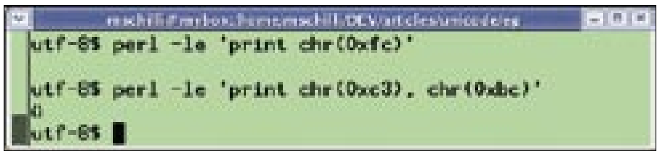
UTF 8

The UTF 8 standard provides one approach to encoding the Unicode table effectively on a computer. If the sudden need had arisen to encode legacy ASCII characters with 2 or 4 bytes throughout, memory requirements would have skyrocketed. To retain the ability to render the legacy ASCII table with a single byte, the UTF 8 table was designed so that the first 128 characters are exactly the same as the ASCII table.



```
iso-8859-15$ perl -le 'print chr(0xfc)'
ü
iso-8859-15$ perl -le 'print chr(0xc3), chr(0xbc)'
Ã¼
iso-8859-15$ █
```

Figure 1: In a terminal set to ISO-8859-15, Latin 1 output is fine, but output in UTF 8 looks like Kashubian.



```
utf-8$ perl -le 'print chr(0xfc)'

utf-8$ perl -le 'print chr(0xc3), chr(0xbc)'
ü
utf-8$ █
```

Figure 2: In a terminal set to UTF 8, UTF 8 output works fine, but ISO-8859-15-encoded characters are not displayed.

However, the next group of 128 characters is made up of special masking codes that indicate that a specific number of additional codes follow to uniquely identify which character in the Unicode table to display. For example, the German umlaut (*ü*) is stored as number 252 (*0xFC*) in the ISO-8859-15 table. If you have an ISO-8859-15 text that contains a byte with a value of *0xFC*, the character is obviously a *ü*.

A With a Wavy Line

In UTF 8 encoding, the umlaut *ü* is represented by 2 bytes – 195 and 188 (*0xC3* and *0xBC*). If you have a UTF 8 text and the computer first sees a byte value of *0xC3* followed by *0xBC*, it is equally as clear that this must be the letter *ü*.

On the other hand, if the encoding is unclear and the computer sees a byte with a value of *0xC3*, the question is whether this is the first byte of a Western European umlaut in UTF 8 format, or whether it is an ISO-8859-15-encoded byte that represents a complete character. If the introductory byte, *0xC3*, in a UTF 8 sequence is incorrectly interpreted as ISO-8859-15, a character that can be the bane of programmers who wander between the encoding worlds is displayed – an *A* with a wavy line.

Listing 1: latin1

```
#!/usr/bin/perl -w
use strict;
my $s = "ü";
print "umlaut=$s", "\n";
```

Listing 2: utf8

```
#!/usr/bin/perl -w
use strict;
use utf8;
my $s = "ü";

binmode STDOUT, ":utf8";
print "umlaut=$s", "\n";
```

This character, which is shown in Figure 1, occurs in Portuguese, Vietnamese, or Kashubian writing [2]. If these languages are outside your typical domain, but you see an *A* with a wavy line, you are probably looking at UTF 8-encoded text that has mistakenly been interpreted as ISO-8859-15.

Terminal Display

If you want your terminal to display the text output from a program on screen, the terminal needs to know how to interpret the bytestream output by the program to locate the right characters for the display.

To launch an X terminal with UTF 8 output, you could run *xterm* with the *-u8 +lc* options. The *-u8* option enables UTF 8, and *+lc* disables the interpretation of environmental variables, such as *LANG*, to prevent them interfering.

To launch a terminal in ISO-8859-15 mode, run *xterm* with the *LANG* environment variable set:

```
LANG=en_US.ISO-8859-15 xterm
```

Figures 1 and 2 show the output from two Perl scripts in an ISO and a UTF 8 terminal, respectively. A single byte with a value of *0xFC* is correctly interpreted as *ü* by the red ISO terminal.

However, the UTF 8 sequence *0xC3BC* is rendered as an *A* with a tilde and the ISO representation of *0xBC*, which is the *1/4* character.

In contrast, the green UTF 8 terminal does not display a single byte with a value of *0xFC*, but the UTF 8 *0xC3BC* sequence displays a *ü* as expected.

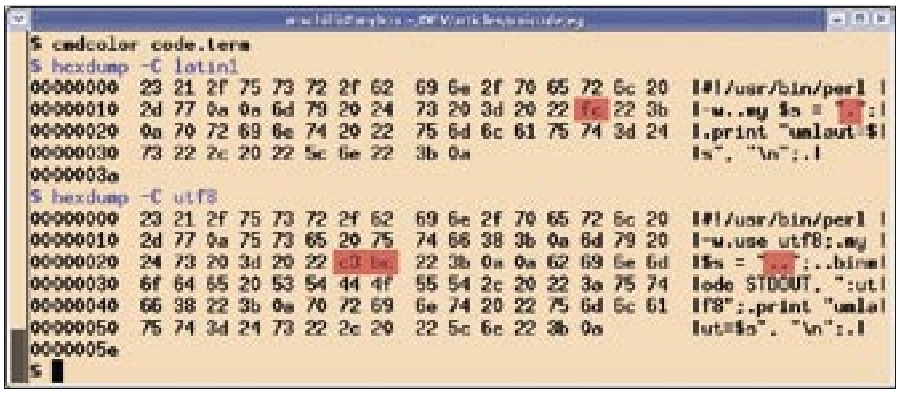
Latin as the Standard Case

Unless you tell Perl to do otherwise, it will interpret the source code of a script, including any strings, regular expressions, variables, and function names, as ISO-8859-1 encoded.

If you use an editor set to ISO-8859-1 to program the code in Listing 1, the *ü* in the program text will be rendered by a code of *0xFC*, as the *hexdump* utility goes on to prove (Figure 3).

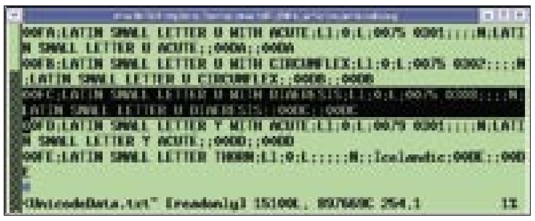
In contrast, Listing 2 was created in the vim editor with a setting of *set encoding=utf-8*. The red markings in the hexdump screen in Figure 3 show that the umlaut in the string in the program code really has been encoded by two bytes: *C3* and *BC*.

You might have noticed the two extra lines shown in Listing 2. First, I set the pragma *use utf8* to tell Perl to interpret the source code of the script as UTF 8. This ensures that the string "*ü*", which is represented by *0xC3BC* in the source code, holds a single character – the Unicode character *ü*. Reflecting this, *length($s)* won't return a value of 2, but just 1.



```
$ cmdcolor code.term
$ hexdump -C latin1
00000000  23 21 2f 75 73 72 2f 62  69 6e 2f 70 65 72 6c 20  |#!/usr/bin/perl |
00000010  2d 77 0a 0a 6d 79 20 24  73 20 3d 20 22 fc 22 3b  |-w..my $s = ".";|
00000020  0a 70 72 69 6e 74 20 22  75 6d 6c 61 75 74 3d 24  |.print "umlaut=$|
00000030  73 22 2c 20 22 5c 6e 22  3b 0a                    |s", "\n";.|
0000003a
$ hexdump -C utf8
00000000  23 21 2f 75 73 72 2f 62  69 6e 2f 70 65 72 6c 20  |#!/usr/bin/perl |
00000010  2d 77 0a 75 73 65 20 75  74 66 38 3b 0a 6d 79 20  |-w.use utf8;.my |
00000020  24 73 20 3d 20 22 c3 bc  22 3b 0a 0a 62 69 6e 6d  |$s = "..";..binm|
00000030  6f 64 65 20 53 54 44 4f  55 54 2c 20 22 3a 75 74  |ode STDOUT, ":ut|
00000040  66 38 22 3b 0a 70 72 69  6e 74 20 22 75 6d 6c 61  |f8";.print "umla|
00000050  75 74 3d 24 73 22 2c 20  22 5c 6e 22 3b 0a        |ut=$s", "\n";.|
0000005e
$ █
```

Figure 3: A Perl script written in ISO-8859 will use a code of 0xFC to represent the ü in the source code. If the source code is written in UTF 8, the ü is represented by a byte sequence of 0xC3BC instead.



```
00FA;LATIN SMALL LETTER U WITH ACUTE;Ll;0;L;0075 0301;;;;N;LATI
N SMALL LETTER U ACUTE;;00DA;;00DA
00FB;LATIN SMALL LETTER U WITH CIRCUMFLEX;Ll;0;L;0075 0302;;;;N
;LATIN SMALL LETTER U CIRCUMFLEX;;00DB;;00DB
00FC;LATIN SMALL LETTER U WITH DIAERESIS;Ll;0;L;0075 0308;;;;N;
LATIN SMALL LETTER U DIAERESIS;;00DC;;00DC
00FD;LATIN SMALL LETTER Y WITH ACUTE;Ll;0;L;0079 0301;;;;N;LATI
N SMALL LETTER Y ACUTE;;00DD;;00DD
00FE;LATIN SMALL LETTER THORN;Ll;0;L;;;;;N;;Icelandic;00DE;;00D
E
"UnicodeData.txt" [readonly] 15100L, 897668C 254,1            1%
```

Figure 4: The entry for ü in Perl's Unicode table.

Next, Listing 2 sets the line discipline for standard output to UTF 8 mode by calling *binmode(STDOUT, ":utf8")*. This makes sure that Perl will output Unicode strings to standard output in UTF 8 format and, thus, that the terminal will receive the UTF 8 data it expects and render it properly. Without the call to *binmode*, Perl would attempt to convert the output string to Latin 1. This makes sense for an ISO-8859-1 terminal, but it is exactly what you want to avoid with a UTF 8 terminal.

And things go completely wrong if the Unicode character cannot be converted to Latin 1, such as the Japanese Katakana character "me," which has a Unicode number of *30E1*. In this case, you see a "Wide character in print" warning. The use of *binmode(STDOUT, ":utf8")* to set the line discipline for the output file handle prevents Perl from trying to convert the Unicode string and tells it to output raw UTF 8 instead. If you have a UTF 8 terminal, this is exactly the strategy you need.

Reference

The *man iso-8859-1* man page details Latin 1 standard encoding. If you refer to the octal number 374, or the hex value of *FC*, you will see an entry for *LATIN SMALL LETTER U WITH DIAERESIS*, or the umlaut *ü*.

If you need to refer to the Unicode table, the *unicore/UnicodeData.txt* file, below *lib* in your Perl distribution (typically *ls/usr/lib/perl5/5.8.x*), is where to look. Again, you will find an entry for the number *00FC*: *LATIN SMALL LETTER U WITH DIAERESIS*.

If you look carefully at Figure 4, you will notice that the sequence number for the small *ü* in the Unicode table corresponds to the number for the small *ü* in the ISO-8859-1 table. It is *FC* in both cases (*FC* in ISO-8859-1 and *00FC* in Unicode) because the creators of the Unicode table modeled the first 256 bytes following the ISO-8859-1 standard.

Note that the Unicode number does not represent the UTF 8 encoding of the character. For example, *ü*, the Unicode character with the number *00FC*, is represented as *C3BC* in UTF8. As I mentioned previously, UTF 8 is just a more efficient approach to encoding the Unicode table.

Keyboard without Foreign Characters

If you have a keyboard with umlauts (like the one I use in San Francisco), it is easy to type a *ü* in a string by entering its Unicode number (Figure 5):

```
my $s = "\x{00FC}";
```

As an alternative, some editors support keyboard shortcuts. The keyboard shortcut for a lowercase *ü* in the vim editor's input mode is Ctrl + k *u:*. Here, note that vim is set to UTF 8 via *set encoding = utf-8*.

In and Out

When a Perl program reads or outputs data, the programmer has to specify the input or output format for the data. To read the lines in a UTF 8-encoded text file, you can either resort to the *binmode* trick I demonstrated previously to set the *FILE* filehandle to *:utf8* or set the line discipline using an *open* command with three parameters: *open FILE, "< :utf8", "file.txt"* … . If the program then reads a line of the file with *< FILE >* and assigns the results to a scalar, you can be sure that the string is a Unicode string, and Perl will make a note of this fact internally.

Figure 5: With a keyboard that has an umlaut, it is easy to type an umlaut in a string by entering its Unicode number,

Listing 3: peek

```
#!/usr/bin/perl -w
use strict;
use utf8;
use Data::Hexdumper;
use Encode qw(_utf8_off is_
  utf8);

my $s = "ü";

if( is_utf8($s) ) {
    print "UTF-8 flag^^is
'on'.\n";
}

print "Len: ", length($s), "\
  n";
_utf8_off($s);
print "Len: ", length($s), "\
  n";

print hexdump(data => $s), "\
  n";
```

Without line discipline, the input would be interpreted as ISO-8859-1, and Perl would cram the raw bytes into a string scalar without marking it as UTF 8. The same principle applies to output. A *> :utf8* or *> > :utf8* as the second parameter with *open* sets the line discipline for the output to UTF 8 mode, and a *print FILE $string* will output the UTF 8-encoded string without modifications. As an alternative, you could use *binmode* to modify the filehandle.

Dropping the Last Veil

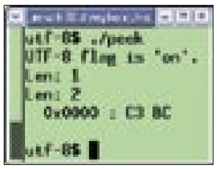
Listing 3 shows how Perl manages Unicode strings internally. Because of the *use utf8* pragma that I previously set, the *"ü"* string (the one that I input in UTF 8 with vim) is identified and managed as a Unicode string.

To allow this to happen, Perl set an internal flag, which you can query (*is_utf8()*) and manipulate (*_utf8_off()*) using the Encode module.

The output from *peek* in Figure 6 shows that the UTF 8 string really has a length of 1. If you delete the flag by setting *_utf8_off()*, the length of the string suddenly grows to two characters.

The output from the CPAN *Data::Hexdumper* module shows that the string is now stored internally as *0xC3BC* – and, presto, that really is UTF 8.

Listing 4 shows how a CGI script promises the browser ISO-8859-1-encoded text but then sends a Euro character with a code of *0x80*, which complies with the Windows-1252 standard [3]. As you can see in Figure 7, the browser generously agrees to display the Euro character.

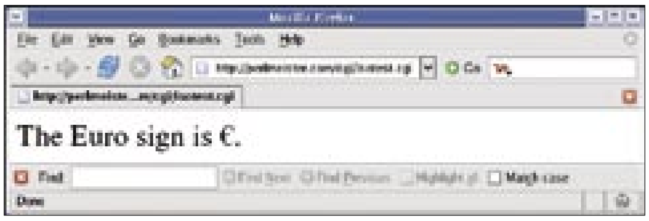


```
utf-8$ ./peek
UTF-8 flag is 'on'.
Len: 1
Len: 2
  0x0000 : C3 BC
utf-8$
```

Figure 6: In a Unicode string, a multi-byte character really has a length of 1. If you remove the Unicode property from the string, Perl will interpret it bytewise.

If the server script were to specify ISO-8859-15 in its header, you would see a black question mark instead of the Euro sign in the browser's rendering of the page.

The Euro sign has a code of *0xA4* in the ISO-8859-15 table. If the code is modified to reflect this, the browser again displays the Euro sign correctly.



The Euro sign is €.

Figure 7: The browser shows the Euro character.

Listing 4: isotest.cg

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:all);

print header(
  -type     => 'text/html',
  -charset  => 'iso-8859-1');

print "The Euro sign is ",
      chr(0x80), ".\n";
```

Listing 5: isotest2.cgi

```
#!/usr/bin/perl -w
use strict;
use CGI qw(:all);

print header(
  -type    => 'text/html',
  -charset => 'utf-8');

binmode STDOUT, ":utf8";
print "The Euro sign is ",
      "\x{20AC}.\n";
```

Not So Generous

Perl's web-client library, LWP, is not so generous. Listing 5 shows an example that outputs text as UTF 8 and even sets the response header correctly.

The Euro character in the string is represented by its Unicode serial number *\x{20AC}* in the string. However, there are a number of things to watch out for on the client side of the web application, if the web page text is UTF 8-encoded server-side. The idea is to use the LWP library to retrieve the page from the web server and, if everything works out okay, to store a Unicode string in Perl. Listing 6 shows the approach.

Because of a known bug in the LWP library (or in *HTML::HeadParser*, to be more precise), Perl throws a nasty *Parsing of undecoded UTF-8 will give garbage when decoding entities* warning when UTF 8 is returned, although you should be able to avoid this by setting the *parse_head = > 0* option in the call to the *UserAgent*'s constructor [4].

Listing 6: webclient

```
#!/usr/bin/perl -w
use strict;
use LWP::UserAgent;

my $ua = LWP::UserAgent->new(
   parse_head => 0
);

my $resp = $ua->get(
  "http://perlmeister.com/cgi/isotest.cgi");

if($resp->is_success()) {
    my $text = $resp->decoded_content();
    binmode STDOUT, ":utf8";
    print "$text\n";
}
```

To allow Perl to store the returned UTF 8 text in a Unicode string, you need to avoid using the typical *content()* method to extract the text for the page from the *HTTP::Response* object and rely on *decoded_content()* instead.

This method uses the *charset* field in the the web server's response to figure out how to decode content. As long as the client continues to honor the line discipline for STDOUT, there is nothing to prevent correct rendering in a terminal set to UTF 8 mode.

Conclusions

Wandering between the encoding worlds has always been a problem. But if you prefer not to restrict availability of your software to a fraction of the market, it makes sense to work hard on an internationalization strategy. ■

THE AUTHOR

Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at *mschilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/81*

[2] "The Mystery of the Double-Encoded UTF8": *http://blog.360.yahoo.com/blog-8_S91Lc7dKj4rz8iueEaVlexawc_ZFVpd4JK?p=16*

[3] Windows-1252: *http://en.wikipedia.org/wiki/Windows-1252*

[4] Known LWP bug in UTF 8-encoded web pages: *http://www.mail-archive.com/libwww@perl.org/msg06330.html*