NAME

encoding - allows you to write your script in non-ascii or non-utf8

WARNING

This module is deprecated under perl 5.18. It uses a mechanism provided by perl that is deprecated under 5.18 and higher, and may be removed in a future version.

SYNOPSIS

```
  use encoding "greek";  # Perl like Greek to you?
  use encoding "euc-jp"; # Jperl!

  # or you can even do this if your shell supports your native encoding

  perl -Mencoding=latin2 -e'...' # Feeling centrally European?
  perl -Mencoding=euc-kr -e'...' # Or Korean?

  # more control

  # A simple euc-cn => utf-8 converter
  use encoding "euc-cn", STDOUT => "utf8";  while(<>){print};

  # "no encoding;" supported (but not scoped!)
  no encoding;

  # an alternate way, Filter
  use encoding "euc-jp", Filter=>1;
  # now you can use kanji identifiers -- in euc-jp!

  # switch on locale -
  # note that this probably means that unless you have a complete control
  # over the environments the application is ever going to be run, you
should
  # NOT use the feature of encoding pragma allowing you to write your
script
  # in any recognized encoding because changing locale settings will wreck
  # the script; you can of course still use the other features of the
pragma.
  use encoding ':locale';
```

ABSTRACT

Let's start with a bit of history: Perl 5.6.0 introduced Unicode support. You could apply `substr()` and regexes even to complex CJK characters -- so long as the script was written in UTF-8. But back then, text editors that supported UTF-8 were still rare and many users instead chose to write scripts in legacy encodings, giving up a whole new feature of Perl 5.6.

Rewind to the future: starting from perl 5.8.0 with the **encoding** pragma, you can write your script in any encoding you like (so long as the `Encode` module supports it) and still enjoy Unicode support. This pragma achieves that by doing the following:

- Internally converts all literals (`q//`,`qq//`,`qr//`,`qw///`, `qx//`) from the encoding specified to utf8. In Perl 5.8.1 and later, literals in `tr///` and `DATA` pseudo-filehandle are also converted.
- Changing PerlIO layers of `STDIN` and `STDOUT` to the encoding specified.

Literal Conversions

You can write code in EUC-JP as follows:

```
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
             #<-char-><-char->   # 4 octets
s/\bCamel\b/$Rakuda/;
```

And with `use encoding "euc-jp"` in effect, it is the same thing as the code in UTF-8:

```
my $Rakuda = "\x{99F1}\x{99DD}"; # two Unicode Characters
s/\bCamel\b/$Rakuda/;
```

PerlIO layers for STD(IN|OUT)

The **encoding** pragma also modifies the filehandle layers of STDIN and STDOUT to the specified encoding. Therefore,

```
use encoding "euc-jp";
my $message = "Camel is the symbol of perl.\n";
my $Rakuda = "\xF1\xD1\xF1\xCC"; # Camel in Kanji
$message =~ s/\bCamel\b/$Rakuda/;
print $message;
```

Will print "\xF1\xD1\xF1\xCC is the symbol of perl.\n", not "\x{99F1}\x{99DD} is the symbol of perl.\n".

You can override this by giving extra arguments; see below.

Implicit upgrading for byte strings

By default, if strings operating under byte semantics and strings with Unicode character data are concatenated, the new string will be created by decoding the byte strings as *ISO 8859-1 (Latin-1)*.

The **encoding** pragma changes this to use the specified encoding instead. For example:

```
use encoding 'utf8';
my $string = chr(20000); # a Unicode string
utf8::encode($string);   # now it's a UTF-8 encoded byte string
# concatenate with another Unicode string
print length($string . chr(20000));
```

Will print 2, because `$string` is upgraded as UTF-8. Without `use encoding 'utf8';`, it will print 4 instead, since `$string` is three octets when interpreted as Latin-1.

Side effects

If the `encoding` pragma is in scope then the lengths returned are calculated from the length of `$/` in Unicode characters, which is not always the same as the length of `$/` in the native encoding.

This pragma affects utf8::upgrade, but not utf8::downgrade.

FEATURES THAT REQUIRE 5.8.1

Some of the features offered by this pragma requires perl 5.8.1. Most of these are done by Inaba Hiroto. Any other features and changes are good for 5.8.0.

"NON-EUC" doublebyte encodings

> Because perl needs to parse script before applying this pragma, such encodings as Shift_JIS and Big-5 that may contain '\' (BACKSLASH; \x5c) in the second byte fails because the second byte may accidentally escape the quoting character that follows. Perl 5.8.1 or later fixes this problem.

tr//

`tr//` was overlooked by Perl 5 porters when they released perl 5.8.0 See the section below for details.

DATA pseudo-filehandle

Another feature that was overlooked was DATA.

USAGE

use encoding [*ENCNAME*] ;

Sets the script encoding to *ENCNAME*. And unless ${^UNICODE} exists and non-zero, PerlIO layers of STDIN and STDOUT are set to ":encoding(*ENCNAME*)".

Note that STDERR WILL NOT be changed.

Also note that non-STD file handles remain unaffected. Use `use open` or `binmode` to change layers of those.

If no encoding is specified, the environment variable *PERL_ENCODING* is consulted. If no encoding can be found, the error `Unknown encoding '`*ENCNAME*`'` will be thrown.

use encoding *ENCNAME* [STDIN => *ENCNAME_IN* ...] ;

You can also individually set encodings of STDIN and STDOUT via the `STDIN =>` *ENCNAME* form. In this case, you cannot omit the first *ENCNAME*. `STDIN => undef` turns the IO transcoding completely off.

When ${^UNICODE} exists and non-zero, these options will completely ignored. ${^UNICODE} is a variable introduced in perl 5.8.1. See *perlrun* see *"${^UNICODE}" in perlvar* and *"-C" in perlrun* for details (perl 5.8.1 and later).

use encoding *ENCNAME* Filter=>1;

This turns the encoding pragma into a source filter. While the default approach just decodes interpolated literals (in qq() and qr()), this will apply a source filter to the entire source code. See *The Filter Option* below for details.

no encoding;

Unsets the script encoding. The layers of STDIN, STDOUT are reset to ":raw" (the default unprocessed raw stream of bytes).

The Filter Option

The magic of `use encoding` is not applied to the names of identifiers. In order to make `${"\x{4eba}"}++` ($human++, where human is a single Han ideograph) work, you still need to write your script in UTF-8 -- or use a source filter. That's what 'Filter=>1' does.

What does this mean? Your source code behaves as if it is written in UTF-8 with 'use utf8' in effect. So even if your editor only supports Shift_JIS, for example, you can still try examples in Chapter 15 of `Programming Perl, 3rd Ed.`. For instance, you can use UTF-8 identifiers.

This option is significantly slower and (as of this writing) non-ASCII identifiers are not very stable WITHOUT this option and with the source code written in UTF-8.

Filter-related changes at Encode version 1.87

- The Filter option now sets STDIN and STDOUT like non-filter options. And `STDIN=>` *`ENCODING`* and `STDOUT=>`*`ENCODING`* work like non-filter version.
- `use utf8` is implicitly declared so you no longer have to `use utf8` to `${"\x{4eba}"}++`.

CAVEATS

NOT SCOPED

The pragma is a per script, not a per block lexical. Only the last `use encoding` or `no encoding` matters, and it affects **the whole script**. However, the <no encoding> pragma is supported and **use encoding** can appear as many times as you want in a given script. The multiple use of this pragma is discouraged.

By the same reason, the use this pragma inside modules is also discouraged (though not as strongly discouraged as the case above. See below).

If you still have to write a module with this pragma, be very careful of the load order. See the codes below;

```
# called module
package Module_IN_BAR;
use encoding "bar";
# stuff in "bar" encoding here
1;

# caller script
use encoding "foo"
use Module_IN_BAR;
# surprise! use encoding "bar" is in effect.
```

The best way to avoid this oddity is to use this pragma RIGHT AFTER other modules are loaded. i.e.

```
use Module_IN_BAR;
use encoding "foo";
```

DO NOT MIX MULTIPLE ENCODINGS

Notice that only literals (string or regular expression) having only legacy code points are affected: if you mix data like this

```
\xDF\x{100}
```

the data is assumed to be in (Latin 1 and) Unicode, not in your native encoding. In other words, this will match in "greek":

```
"\xDF" =~ /\x{3af}/
```

but this will not

```
"\xDF\x{100}" =~ /\x{3af}\x{100}/
```

since the `\xDF` (ISO 8859-7 GREEK SMALL LETTER IOTA WITH TONOS) on the left will **not** be upgraded to `\x{3af}` (Unicode GREEK SMALL LETTER IOTA WITH TONOS) because of the `\x{100}` on the left. You should not be mixing your legacy data and Unicode in the same string.

This pragma also affects encoding of the 0x80..0xFF code point range: normally characters in that range are left as eight-bit bytes (unless they are combined with characters with code points 0x100 or larger, in which case all characters need to become UTF-8 encoded), but if the `encoding` pragma is present, even the 0x80..0xFF range always gets UTF-8 encoded.

After all, the best thing about this pragma is that you don't have to resort to \x{....} just to spell your name in a native encoding. So feel free to put your strings in your encoding in quotes and regexes.

tr/// with ranges

The **encoding** pragma works by decoding string literals in `q//`, `qq//`, `qr//`, `qw///`, `qx//` and so forth. In perl 5.8.0, this does not apply to `tr///`. Therefore,

```
use encoding 'euc-jp';
#....
$kana =~ tr/\xA4\xA1-\xA4\xF3/\xA5\xA1-\xA5\xF3/;
#           -------- -------- -------- --------
```

Does not work as

```
$kana =~ tr/\x{3041}-\x{3093}/\x{30a1}-\x{30f3}/;
```

Legend of characters above

```
       utf8     euc-jp   charnames::viacode()
       ----------------------------------------
       \x{3041} \xA4\xA1 HIRAGANA LETTER SMALL A
       \x{3093} \xA4\xF3 HIRAGANA LETTER N
       \x{30a1} \xA5\xA1 KATAKANA LETTER SMALL A
       \x{30f3} \xA5\xF3 KATAKANA LETTER N
```

This counterintuitive behavior has been fixed in perl 5.8.1.

workaround to tr///;

In perl 5.8.0, you can work around as follows;

```
use encoding 'euc-jp';
#  ....
eval qq{ \$kana =~ tr/\xA4\xA1-\xA4\xF3/\xA5\xA1-\xA5\xF3/ };
```

Note the `tr//` expression is surrounded by `qq{}`. The idea behind is the same as classic idiom that makes `tr///` 'interpolate'.

```
tr/$from/$to/;            # wrong!
eval qq{ tr/$from/$to/ }; # workaround.
```

Nevertheless, in case of **encoding** pragma even `q//` is affected so `tr///` not being decoded was obviously against the will of Perl5 Porters so it has been fixed in Perl 5.8.1 or later.

EXAMPLE - Greekperl

```
use encoding "iso 8859-7";

# \xDF in ISO 8859-7 (Greek) is \x{3af} in Unicode.

$a = "\xDF";
$b = "\x{100}";

printf "%#x\n", ord($a); # will print 0x3af, not 0xdf

$c = $a . $b;

# $c will be "\x{3af}\x{100}", not "\x{df}\x{100}".
```

```
    # chr() is affected, and ...

    print "mega\n"  if ord(chr(0xdf)) == 0x3af;

    # ... ord() is affected by the encoding pragma ...

    print "tera\n" if ord(pack("C", 0xdf)) == 0x3af;

    # ... as are eq and cmp ...

    print "peta\n" if "\x{3af}" eq  pack("C", 0xdf);
    print "exa\n"  if "\x{3af}" cmp pack("C", 0xdf) == 0;

    # ... but pack/unpack C are not affected, in case you still
    # want to go back to your native encoding

    print "zetta\n" if unpack("C", (pack("C", 0xdf))) == 0xdf;
```

KNOWN PROBLEMS

literals in regex that are longer than 127 bytes

For native multibyte encodings (either fixed or variable length), the current implementation of the regular expressions may introduce recoding errors for regular expression literals longer than 127 bytes.

EBCDIC

The encoding pragma is not supported on EBCDIC platforms. (Porters who are willing and able to remove this limitation are welcome.)

format

This pragma doesn't work well with format because PerlIO does not get along very well with it. When format contains non-ascii characters it prints funny or gets "wide character warnings". To understand it, try the code below.

```
  # Save this one in utf8
  # replace *non-ascii* with a non-ascii string
  my $camel;
  format STDOUT =
  *non-ascii*@>>>>>>>
  $camel
  .
  $camel = "*non-ascii*";
  binmode(STDOUT=>':encoding(utf8)'); # bang!
  write;              # funny
  print $camel, "\n"; # fine
```

Without binmode this happens to work but without binmode, print() fails instead of write().

At any rate, the very use of format is questionable when it comes to unicode characters since you have to consider such things as character width (i.e. double-width for ideographs) and directions (i.e. BIDI for Arabic and Hebrew).

Thread safety

`use encoding ...` is not thread-safe (i.e., do not use in threaded applications).

The Logic of :locale

The logic of `:locale` is as follows:

1. If the platform supports the langinfo(CODESET) interface, the codeset returned is used as the default encoding for the open pragma.

2. If 1. didn't work but we are under the locale pragma, the environment variables LC_ALL and LANG (in that order) are matched for encodings (the part after `.`, if any), and if any found, that is used as the default encoding for the open pragma.

3. If 1. and 2. didn't work, the environment variables LC_ALL and LANG (in that order) are matched for anything looking like UTF-8, and if any found, `:utf8` is used as the default encoding for the open pragma.

If your locale environment variables (LC_ALL, LC_CTYPE, LANG) contain the strings 'UTF-8' or 'UTF8' (case-insensitive matching), the default encoding of your STDIN, STDOUT, and STDERR, and of **any subsequent file open**, is UTF-8.

HISTORY

This pragma first appeared in Perl 5.8.0. For features that require 5.8.1 and better, see above.

The `:locale` subpragma was implemented in 2.01, or Perl 5.8.6.

SEE ALSO

perlunicode, *Encode*, *open*, *Filter::Util::Call*,

Ch. 15 of `Programming Perl (3rd Edition)` by Larry Wall, Tom Christiansen, Jon Orwant; O'Reilly & Associates; ISBN 0-596-00027-8