NAME

perluniintro - Perl Unicode introduction

DESCRIPTION

This document gives a general idea of Unicode and how to use Unicode in Perl. See ‘‘Further Resources’’ for references to more in-depth treatments of Unicode.

Unicode

Unicode is a character set standard which plans to codify all of the writing systems of the world, plus many other symbols.

Unicode and ISO/IEC 10646 are coordinated standards that unify almost all other modern character set standards, covering more than 80 writing systems and hundreds of languages, including all commercially-important modern languages. All characters in the largest Chinese, Japanese, and Korean dictionaries are also encoded. The standards will eventually cover almost all characters in more than 250 writing systems and thousands of languages. Unicode 1.0 was released in October 1991, and 6.0 in October 2010.

A Unicode *character* is an abstract entity. It is not bound to any particular integer width, especially not to the C language `char`. Unicode is language-neutral and display-neutral: it does not encode the language of the text, and it does not generally define fonts or other graphical layout details. Unicode operates on characters and on text built from those characters.

Unicode defines characters like `LATIN CAPITAL LETTER A` or `GREEK SMALL LETTER ALPHA` and unique numbers for the characters, in this case 0x0041 and 0x03B1, respectively. These unique numbers are called *code points*. A code point is essentially the position of the character within the set of all possible Unicode characters, and thus in Perl, the term *ordinal* is often used interchangeably with it.

The Unicode standard prefers using hexadecimal notation for the code points. If numbers like `0x0041` are unfamiliar to you, take a peek at a later section, ‘‘Hexadecimal Notation’’. The Unicode standard uses the notation `U+0041 LATIN CAPITAL LETTER A`, to give the hexadecimal code point and the normative name of the character.

Unicode also defines various *properties* for the characters, like ‘‘uppercase’’ or ‘‘lowercase’’, ‘‘decimal digit’’, or ‘‘punctuation’’; these properties are independent of the names of the characters. Furthermore, various operations on the characters like uppercasing, lowercasing, and collating (sorting) are defined.

A Unicode *logical* ‘‘character’’ can actually consist of more than one internal *actual* ‘‘character’’ or code point. For Western languages, this is adequately modelled by a *base character* (like `LATIN CAPITAL LETTER A`) followed by one or more *modifiers* (like `COMBINING ACUTE ACCENT`). This sequence of base character and modifiers is called a *combining character sequence*. Some non-western languages require more complicated models, so Unicode created the *grapheme cluster* concept, which was later further refined into the *extended grapheme cluster*. For example, a Korean Hangul syllable is considered a single logical character, but most often consists of three actual Unicode characters: a leading consonant followed by an interior vowel followed by a trailing consonant.

Whether to call these extended grapheme clusters ‘‘characters’’ depends on your point of view. If you are a programmer, you probably would tend towards seeing each element in the sequences as one unit, or ‘‘character’’. However from the user’s point of view, the whole sequence could be seen as one ‘‘character’’ since that’s probably what it looks like in the context of the user’s language. In this document, we take the programmer’s point of view: one ‘‘character’’ is one Unicode code point.

For some combinations of base character and modifiers, there are *precomposed* characters. There is a single character equivalent, for example, for the sequence `LATIN CAPITAL LETTER A` followed by `COMBINING ACUTE ACCENT`. It is called `LATIN CAPITAL LETTER A WITH ACUTE`. These precomposed characters are, however, only available for some combinations, and are mainly meant to support round-trip conversions between Unicode and legacy standards (like ISO 8859). Using sequences, as Unicode does, allows for needing fewer basic building blocks (code points) to express many more potential grapheme clusters. To support conversion between equivalent forms, various *normalization forms* are also defined. Thus, `LATIN CAPITAL LETTER A WITH ACUTE` is in *Normalization Form Composed*, (abbreviated NFC), and the sequence `LATIN CAPITAL LETTER A` followed by `COMBINING ACUTE`

`ACCENT` represents the same character in *Normalization Form Decomposed* (NFD).

Because of backward compatibility with legacy encodings, the ''a unique number for every character'' idea breaks down a bit: instead, there is ''at least one number for every character''. The same character could be represented differently in several legacy encodings. The converse is not true: some code points do not have an assigned character. Firstly, there are unallocated code points within otherwise used blocks. Secondly, there are special Unicode control characters that do not represent true characters.

When Unicode was first conceived, it was thought that all the world's characters could be represented using a 16-bit word; that is a maximum of `0x10000` (or 65,536) characters would be needed, from `0x0000` to `0xFFFF`. This soon proved to be wrong, and since Unicode 2.0 (July 1996), Unicode has been defined all the way up to 21 bits (`0x10FFFF`), and Unicode 3.1 (March 2001) defined the first characters above `0xFFFF`. The first`0x10000` characters are called the *Plane 0*, or the *Basic Multilingual Plane* (BMP). With Unicode 3.1, 17 (yes, seventeen) planes in all were defined — but they are nowhere near full of defined characters, yet.

When a new language is being encoded, Unicode generally will choose a `block` of consecutive unallocated code points for its characters. So far, the number of code points in these blocks has always been evenly divisible by 16. Extras in a block, not currently needed, are left unallocated, for future growth. But there have been occasions when a later release needed more code points than the available extras, and a new block had to allocated somewhere else, not contiguous to the initial one, to handle the overflow. Thus, it became apparent early on that ''block'' wasn't an adequate organizing principal, and so the `Script` property was created. (Later an improved script property was added as well, the `Script_Extensions` property.) Those code points that are in overflow blocks can still have the same script as the original ones. The script concept fits more closely with natural language: there is `Latin` script, `Greek` script, and so on; and there are several artificial scripts, like `Common` for characters that are used in multiple scripts, such as mathematical symbols. Scripts usually span varied parts of several blocks. For more information about scripts, see ''Scripts'' in perlunicode. The division into blocks exists, but it is almost completely accidental — an artifact of how the characters have been and still are allocated. (Note that this paragraph has oversimplified things for the sake of this being an introduction. Unicode doesn't really encode languages, but the writing systems for them — their scripts; and one script can be used by many languages. Unicode also encodes things that aren't really about languages, such as symbols like `BAGGAGE CLAIM`.)

The Unicode code points are just abstract numbers. To input and output these abstract numbers, the numbers must be *encoded* or *serialised* somehow. Unicode defines several *character encoding forms*, of which *UTF-8* is the most popular. UTF-8 is a variable length encoding that encodes Unicode characters as 1 to 4 bytes. Other encodings include UTF-16 and UTF-32 and their big- and little-endian variants (UTF-8 is byte-order independent). The ISO/IEC 10646 defines the UCS-2 and UCS-4 encoding forms.

For more information about encodings — for instance, to learn what *surrogates* and *byte order marks* (BOMs) are — see perlunicode.

Perl's Unicode Support

Starting from Perl v5.6.0, Perl has had the capacity to handle Unicode natively. Perl v5.8.0, however, is the first recommended release for serious Unicode work. The maintenance release 5.6.1 fixed many of the problems of the initial Unicode implementation, but for example regular expressions still do not work with Unicode in 5.6.1. Perl v5.14.0 is the first release where Unicode support is (almost) seamlessly integrable without some gotchas (the exception being some differences in quotemeta, and that is fixed starting in Perl 5.16.0). To enable this seamless support, you should `use feature 'unicode_strings'` (which is automatically selected if you `use 5.012` or higher). See feature. (5.14 also fixes a number of bugs and departures from the Unicode standard.)

Before Perl v5.8.0, the use of `use utf8` was used to declare that operations in the current block or file would be Unicode-aware. This model was found to be wrong, or at least clumsy: the ''Unicodeness'' is now carried with the data, instead of being attached to the operations. Starting with Perl v5.8.0, only one case remains where an explicit `use utf8` is needed: if your Perl script itself is encoded in UTF-8, you can use UTF-8 in your identifier names, and in string and regular expression literals, by saying `use utf8`. This is not the default because scripts with legacy 8-bit data in them would break. See utf8.

Perl's Unicode Model

Perl supports both pre-5.6 strings of eight-bit native bytes, and strings of Unicode characters. The general principle is that Perl tries to keep its data as eight-bit bytes for as long as possible, but as soon as Unicodeness cannot be avoided, the data is transparently upgraded to Unicode. Prior to Perl v5.14.0, the upgrade was not completely transparent (see "The "Unicode Bug"" in perlunicode), and for backwards compatibility, full transparency is not gained unless `use feature 'unicode_strings'` (see feature) or `use 5.012` (or higher) is selected.

Internally, Perl currently uses either whatever the native eight-bit character set of the platform (for example Latin-1) is, defaulting to UTF-8, to encode Unicode strings. Specifically, if all code points in the string are `0xFF` or less, Perl uses the native eight-bit character set. Otherwise, it uses UTF-8.

A user of Perl does not normally need to know nor care how Perl happens to encode its internal strings, but it becomes relevant when outputting Unicode strings to a stream without a PerlIO layer (one with the "default" encoding). In such a case, the raw bytes used internally (the native character set or UTF-8, as appropriate for each string) will be used, and a "Wide character" warning will be issued if those strings contain a character beyond 0x00FF.

For example,

```
perl -e 'print "\x{DF}\n", "\x{0100}\x{DF}\n"'
```

produces a fairly useless mixture of native bytes and UTF-8, as well as a warning:

```
Wide character in print at ...
```

To output UTF-8, use the `:encoding` or `:utf8` output layer. Prepending

```
binmode(STDOUT, ":utf8");
```

to this sample program ensures that the output is completely UTF-8, and removes the program's warning.

You can enable automatic UTF-8-ification of your standard file handles, default `open()` layer, and `@ARGV` by using either the `-C` command line switch or the `PERL_UNICODE` environment variable, see perlrun(1) for the documentation of the `-C` switch.

Note that this means that Perl expects other software to work the same way: if Perl has been led to believe that STDIN should be UTF-8, but then STDIN coming in from another command is not UTF-8, Perl will likely complain about the malformed UTF-8.

All features that combine Unicode and I/O also require using the new PerlIO feature. Almost all Perl 5.8 platforms do use PerlIO, though: you can see whether yours is by running "perl -V" and looking for `useperlio=define`.

Unicode and EBCDIC

Perl 5.8.0 added support for Unicode on EBCDIC platforms. This support was allowed to lapse in later releases, but was revived in 5.22. Unicode support is somewhat more complex to implement since additional conversions are needed. See perlebcdic(1) for more information.

On EBCDIC platforms, the internal Unicode encoding form is UTF-EBCDIC instead of UTF-8. The difference is that as UTF-8 is "ASCII-safe" in that ASCII characters encode to UTF-8 as-is, while UTF-EBCDIC is "EBCDIC-safe", in that all the basic characters (which includes all those that have ASCII equivalents (like `"A"`, `"0"`, `"%"`, *etc.*) are the same in both EBCDIC and UTF-EBCDIC. Often, documentation will use the term "UTF-8" to mean UTF-EBCDIC as well. This is the case in this document.

Creating Unicode

This section applies fully to Perls starting with v5.22. Various caveats for earlier releases are in the "Earlier releases caveats" subsection below.

To create Unicode characters in literals, use the `\N{...}` notation in double-quoted strings:

```
my $smiley_from_name = "\N{WHITE SMILING FACE}";
my $smiley_from_code_point = "\N{U+263a}";
```

Similarly, they can be used in regular expression literals

```
$smiley =~ /\N{WHITE SMILING FACE}/;
$smiley =~ /\N{U+263a}/;
```

At run-time you can use:

```
use charnames ();
my $hebrew_alef_from_name
= charnames::string_vianame("HEBREW LETTER ALEF");
my $hebrew_alef_from_code_point = charnames::string_vianame("U+05D0");
```

Naturally, `ord()` will do the reverse: it turns a character into a code point.

There are other runtime options as well. You can use `pack()`:

```
my $hebrew_alef_from_code_point = pack("U", 0x05d0);
```

Or you can use `chr()`, though it is less convenient in the general case:

```
$hebrew_alef_from_code_point = chr(utf8::unicode_to_native(0x05d0));
utf8::upgrade($hebrew_alef_from_code_point);
```

The `utf8::unicode_to_native()` and `utf8::upgrade()` aren't needed if the argument is above 0xFF, so the above could have been written as

```
$hebrew_alef_from_code_point = chr(0x05d0);
```

since 0x5d0 is above 255.

`\x{}` and `\o{}` can also be used to specify code points at compile time in double-quotish strings, but, for backward compatibility with older Perls, the same rules apply as with `chr()` for code points less than 256.

`utf8::unicode_to_native()` is used so that the Perl code is portable to EBCDIC platforms. You can omit it if you're *really* sure no one will ever want to use your code on a non-ASCII platform. Starting in Perl v5.22, calls to it on ASCII platforms are optimized out, so there's no performance penalty at all in adding it. Or you can simply use the other constructs that don't require it.

See ''Further Resources'' for how to find all these names and numeric codes.

Earlier releases caveats

On EBCDIC platforms, prior to v5.22, using `\N{U+...}` doesn't work properly.

Prior to v5.16, using `\N{...}` with a character name (as opposed to a `U+...` code point) required a `usecharnames:full`.

Prior to v5.14, there were some bugs in `\N{...}` with a character name (as opposed to a `U+...` code point).

`charnames::string_vianame()` was introduced in v5.14. Prior to that, `charnames::vianame()` should work, but only if the argument is of the form `"U+..."`. Your best bet there for runtime Unicode by character name is probably:

```
use charnames ();
my $hebrew_alef_from_name
= pack("U", charnames::vianame("HEBREW LETTER ALEF"));
```

Handling Unicode

Handling Unicode is for the most part transparent: just use the strings as usual. Functions like `index()`, `length()`, and `substr()` will work on the Unicode characters; regular expressions will work on the Unicode characters (see perlunicode(1) and perlretut).

Note that Perl considers grapheme clusters to be separate characters, so for example

```
print length("\N{LATIN CAPITAL LETTER A}\N{COMBINING ACUTE ACCENT}"),
"\n";
```

will print 2, not 1. The only exception is that regular expressions have `\X` for matching an extended grapheme cluster. (Thus `\X` in a regular expression would match the entire sequence of both the example characters.)

Life is not quite so transparent, however, when working with legacy encodings, I/O, and certain special cases:

Legacy Encodings

When you combine legacy data and Unicode, the legacy data needs to be upgraded to Unicode. Normally the legacy data is assumed to be ISO 8859-1 (or EBCDIC, if applicable).

The `Encode` module knows about many encodings and has interfaces for doing conversions between those encodings:

```
use Encode 'decode';
$data = decode("iso-8859-3", $data); # convert from legacy to utf-8
```

Unicode I/O

Normally, writing out Unicode data

```
print FH $some_string_with_unicode, "\n";
```

produces raw bytes that Perl happens to use to internally encode the Unicode string. Perl's internal encoding depends on the system as well as what characters happen to be in the string at the time. If any of the characters are at code points `0x100` or above, you will get a warning. To ensure that the output is explicitly rendered in the encoding you desire—and to avoid the warning—open the stream with the desired encoding. Some examples:

```
open FH, ">:utf8", "file";

open FH, ">:encoding(ucs2)", "file";
open FH, ">:encoding(UTF-8)", "file";
open FH, ">:encoding(shift_jis)", "file";
```

and on already open streams, use `binmode()`:

```
binmode(STDOUT, ":utf8");

binmode(STDOUT, ":encoding(ucs2)");
binmode(STDOUT, ":encoding(UTF-8)");
binmode(STDOUT, ":encoding(shift_jis)");
```

The matching of encoding names is loose: case does not matter, and many encodings have several aliases. Note that the `:utf8` layer must always be specified exactly like that; it is *not* subject to the loose matching of encoding names. Also note that currently `:utf8` is unsafe for input, because it accepts the data without validating that it is indeed valid UTF-8; you should instead use `:encoding(utf-8)` (with or without a hyphen).

See PerlIO for the `:utf8` layer, PerlIO::encoding and Encode::PerlIO for the `:encoding()` layer, and Encode::Supported for many encodings supported by the `Encode` module.

Reading in a file that you know happens to be encoded in one of the Unicode or legacy encodings does not magically turn the data into Unicode in Perl's eyes. To do that, specify the appropriate layer when opening files

```
open(my $fh,'<:encoding(utf8)', 'anything');
my $line_of_unicode = <$fh>;

open(my $fh,'<:encoding(Big5)', 'anything');
my $line_of_unicode = <$fh>;
```

The I/O layers can also be specified more flexibly with the `open` pragma. See open, or look at the following example.

```
use open ':encoding(utf8)'; # input/output default encoding will be
# UTF-8
open X, ">file";
print X chr(0x100), "\n";
close X;
open Y, "<file";
printf "%#x\n", ord(<Y>); # this should print 0x100
close Y;
```

With the `open` pragma you can use the `:locale` layer

```
BEGIN { $ENV{LC_ALL} = $ENV{LANG} = 'ru_RU.KOI8-R' }
# the :locale will probe the locale environment variables like
# LC_ALL
use open OUT => ':locale'; # russki parusski
open(O, ">koi8");
print O chr(0x430); # Unicode CYRILLIC SMALL LETTER A = KOI8-R 0xc1
close O;
open(I, "<koi8");
printf "%#x\n", ord(<I>), "\n"; # this should print 0xc1
close I;
```

These methods install a transparent filter on the I/O stream that converts data from the specified encoding when it is read in from the stream. The result is always Unicode.

The open pragma affects all the `open()` calls after the pragma by setting default layers. If you want to affect only certain streams, use explicit layers directly in the `open()` call.

You can switch encodings on an already opened stream by using `binmode()`; see "binmode" in perlfunc.

The `:locale` does not currently work with `open()` and `binmode()`, only with the `open` pragma. The `:utf8` and `:encoding(...)` methods do work with all of `open()`, `binmode()`, and the `open` pragma.

Similarly, you may use these I/O layers on output streams to automatically convert Unicode to the specified encoding when it is written to the stream. For example, the following snippet copies the contents of the file "text.jis" (encoded as ISO-2022-JP, aka JIS) to the file "text.utf8", encoded as UTF-8:

```
open(my $nihongo, '<:encoding(iso-2022-jp)', 'text.jis');
open(my $unicode, '>:utf8', 'text.utf8');
while (<$nihongo>) { print $unicode $_ }
```

The naming of encodings, both by the `open()` and by the `open` pragma allows for flexible names: `koi8-r` and `KOI8R` will both be understood.

Common encodings recognized by ISO, MIME, IANA, and various other standardisation organisations are recognised; for a more detailed list see Encode::Supported.

`read()` reads characters and returns the number of characters. `seek()` and `tell()` operate on byte counts, as do `sysread()` and `sysseek()`.

Notice that because of the default behaviour of not doing any conversion upon input if there is no default layer, it is easy to mistakenly write code that keeps on expanding a file by repeatedly encoding the data:

```
# BAD CODE WARNING
open F, "file";
local $/; ## read in the whole file of 8-bit characters
$t = <F>;
close F;
open F, ">:encoding(utf8)", "file";
print F $t; ## convert to UTF-8 on output
close F;
```

If you run this code twice, the contents of the *file* will be twice UTF-8 encoded. A `use open ':encoding(utf8)'` would have avoided the bug, or explicitly opening also the *file* for input as UTF-8.

NOTE: the `:utf8` and `:encoding` features work only if your Perl has been built with PerlIO, which is the default on most systems.

Displaying Unicode As Text

Sometimes you might want to display Perl scalars containing Unicode as simple ASCII (or EBCDIC) text. The following subroutine converts its argument so that Unicode characters with code points greater than 255 are displayed as `\x{...}`, control characters (like `\n`) are displayed as `\x..`, and the rest of the characters as themselves:

```
sub nice_string {
join("",
map { $_ > 255 # if wide character...
? sprintf("\\x{%04X}", $_) # \x{...}
: chr($_) =~ /[[:cntrl:]]/ # else if control character...
? sprintf("\\x%02X", $_) # \x..
: quotemeta(chr($_)) # else quoted or as themselves
} unpack("W*", $_[0])); # unpack Unicode characters
}
```

For example,

```
nice_string("foo\x{100}bar\n")
```

returns the string

```
'foo\x{0100}bar\x0A'
```

which is ready to be printed.

(`\\x{}` is used here instead of `\\N{}`, since it's most likely that you want to see what the native values are.)

Special Cases

- Bit Complement Operator ~ And *vec()*

  The bit complement operator ~ may produce surprising results if used on strings containing characters with ordinal values above 255. In such a case, the results are consistent with the internal encoding of the characters, but not with much else. So don't do that. Similarly for `vec()`: you will be operating on the internally-encoded bit patterns of the Unicode characters, not on the code point values, which is very probably not what you want.

- Peeking At Perl's Internal Encoding

  Normal users of Perl should never care how Perl encodes any particular Unicode string (because the normal ways to get at the contents of a string with Unicode — via input and output — should always be via explicitly-defined I/O layers). But if you must, there are two ways of looking behind the scenes.

  One way of peeking inside the internal encoding of Unicode characters is to use `unpack("C*", ...` to get the bytes of whatever the string encoding happens to be, or `unpack("U0..", ...)` to get the bytes of the UTF-8 encoding:

```
# this prints c4 80 for the UTF-8 bytes 0xc4 0x80
print join(" ", unpack("U0(H2)*", pack("U", 0x100))), "\n";
```

Yet another way would be to use the Devel::Peek module:

```
perl -MDevel::Peek -e 'Dump(chr(0x100))'
```

That shows the `UTF8` flag in FLAGS and both the UTF-8 bytes and Unicode characters in `PV`. See also later in this document the discussion about the `utf8::is_utf8()` function.

Advanced Topics

- String Equivalence

  The question of string equivalence turns somewhat complicated in Unicode: what do you mean by "equal"?

  (Is `LATIN CAPITAL LETTER A WITH ACUTE` equal to `LATIN CAPITAL LETTER A`?)

  The short answer is that by default Perl compares equivalence (`eq`, `ne`) based only on code points of the characters. In the above case, the answer is no (because 0x00C1 != 0x0041). But sometimes, any CAPITAL LETTER A's should be considered equal, or even A's of any case.

  The long answer is that you need to consider character normalization and casing issues: see Unicode::Normalize, Unicode Technical Report #15, Unicode Normalization Forms <http://www.unicode.org/unicode/reports/tr15> and sections on case mapping in the Unicode Standard <http://www.unicode.org>.

  As of Perl 5.8.0, the "Full" case-folding of *Case Mappings/SpecialCasing* is implemented, but bugs remain in `qr//i` with them, mostly fixed by 5.14, and essentially entirely by 5.18.

- String Collation

  People like to see their strings nicely sorted — or as Unicode parlance goes, collated. But again, what do you mean by collate?

  (Does `LATIN CAPITAL LETTER A WITH ACUTE` come before or after `LATIN CAPITAL LETTER A WITH GRAVE`?)

  The short answer is that by default, Perl compares strings (`lt`, `le`, `cmp`, `ge`, `gt`) based only on the code points of the characters. In the above case, the answer is "after", since `0x00C1` > `0x00C0`.

  The long answer is that "it depends", and a good answer cannot be given without knowing (at the very least) the language context. See Unicode::Collate, and *Unicode Collation Algorithm* <http://www.unicode.org/unicode/reports/tr10/>

Miscellaneous

- Character Ranges and Classes

  Character ranges in regular expression bracketed character classes (e.g., `/[a-z]/`) and in the `tr///` (also known as `y///`) operator are not magically Unicode-aware. What this means is that `[A-Za-z]` will not magically start to mean "all alphabetic letters" (not that it does mean that even for 8-bit characters; for those, if you are using locales (perllocale), use `/[[:alpha:]]/`; and if not, use the 8-bit-aware property `\p{alpha}`).

  All the properties that begin with `\p` (and its inverse `\P`) are actually character classes that are Unicode-aware. There are dozens of them, see perluniprops.

  Starting in v5.22, you can use Unicode code points as the end points of regular expression pattern character ranges, and the range will include all Unicode code points that lie between those end points, inclusive.

  ```
  qr/ [\N{U+03]-\N{U+20}] /x
  ```

  includes the code points `\N{U+03}`, `\N{U+04}`, ..., `\N{U+20}`.

(It is planned to extend this behavior to ranges in `tr///` in Perl v5.24.)

- String-To-Number Conversions

  Unicode does define several other decimal — and numeric — characters besides the familiar 0 to 9, such as the Arabic and Indic digits. Perl does not support string-to-number conversion for digits other than ASCII `0` to `9` (and ASCII `a` to `f` for hexadecimal). To get safe conversions from any Unicode string, use "*num()*" in Unicode::UCD.

Questions With Answers

- Will My Old Scripts Break?

  Very probably not. Unless you are generating Unicode characters somehow, old behaviour should be preserved. About the only behaviour that has changed and which could start generating Unicode is the old behaviour of `chr()` where supplying an argument more than 255 produced a character modulo 255. `chr(300)`, for example, was equal to `chr(45)` or "-" (in ASCII), now it is LATIN CAPITAL LETTER I WITH BREVE.

- How Do I Make My Scripts Work With Unicode?

  Very little work should be needed since nothing changes until you generate Unicode data. The most important thing is getting input as Unicode; for that, see the earlier I/O discussion. To get full seamless Unicode support, add `use feature 'unicode_strings'` (or `use 5.012` or higher) to your script.

- How Do I Know Whether My String Is In Unicode?

  You shouldn't have to care. But you may if your Perl is before 5.14.0 or you haven't specified `use feature 'unicode_strings'` or `use 5.012` (or higher) because otherwise the rules for the code points in the range 128 to 255 are different depending on whether the string they are contained within is in Unicode or not. (See "When Unicode Does Not Happen" in perlunicode.)

  To determine if a string is in Unicode, use:

  ```
  print utf8::is_utf8($string) ? 1 : 0, "\n";
  ```

  But note that this doesn't mean that any of the characters in the string are necessary UTF-8 encoded, or that any of the characters have code points greater than 0xFF (255) or even 0x80 (128), or that the string has any characters at all. All the `is_utf8()` does is to return the value of the internal "utf8ness" flag attached to the `$string`. If the flag is off, the bytes in the scalar are interpreted as a single byte encoding. If the flag is on, the bytes in the scalar are interpreted as the (variable-length, potentially multi-byte) UTF-8 encoded code points of the characters. Bytes added to a UTF-8 encoded string are automatically upgraded to UTF-8. If mixed non-UTF-8 and UTF-8 scalars are merged (double-quoted interpolation, explicit concatenation, or printf/sprintf parameter substitution), the result will be UTF-8 encoded as if copies of the byte strings were upgraded to UTF-8: for example,

  ```
  $a = "ab\x80c";
  $b = "\x{100}";
  print "$a = $b\n";
  ```

  the output string will be UTF-8-encoded `ab\x80c = \x{100}\n`, but `$a` will stay byte-encoded.

  Sometimes you might really need to know the byte length of a string instead of the character length. For that use either the `Encode::encode_utf8()` function or the `bytes` pragma and the `length()` function:

```
my $unicode = chr(0x100);
print length($unicode), "\n"; # will print 1
require Encode;
print length(Encode::encode_utf8($unicode)),"\n"; # will print 2
use bytes;
print length($unicode), "\n"; # will also print 2
# (the 0xC4 0x80 of the UTF-8)
no bytes;
```

- How Do I Find Out What Encoding a File Has?

  You might try Encode::Guess, but it has a number of limitations.

- How Do I Detect Data That's Not Valid In a Particular Encoding?

  Use the `Encode` package to try converting it. For example,

  ```
  use Encode 'decode_utf8';

  if (eval { decode_utf8($string, Encode::FB_CROAK); 1 }) {
  # $string is valid utf8
  } else {
  # $string is not valid utf8
  }
  ```

  Or use `unpack` to try decoding it:

  ```
  use warnings;
  @chars = unpack("C0U*", $string_of_bytes_that_I_think_is_utf8);
  ```

  If invalid, a `Malformed UTF-8 character` warning is produced. The "C0" means "process the string character per character". Without that, the `unpack("U*", ...)` would work in `U0` mode (the default if the format string starts with `U`) and it would return the bytes making up the UTF-8 encoding of the target string, something that will always work.

- How Do I Convert Binary Data Into a Particular Encoding, Or Vice Versa?

  This probably isn't as useful as you might think. Normally, you shouldn't need to.

  In one sense, what you are asking doesn't make much sense: encodings are for characters, and binary data are not "characters", so converting "data" into some encoding isn't meaningful unless you know in what character set and encoding the binary data is in, in which case it's not just binary data, now is it?

  If you have a raw sequence of bytes that you know should be interpreted via a particular encoding, you can use `Encode`:

  ```
  use Encode 'from_to';
  from_to($data, "iso-8859-1", "utf-8"); # from latin-1 to utf-8
  ```

  The call to `from_to()` changes the bytes in `$data`, but nothing material about the nature of the string has changed as far as Perl is concerned. Both before and after the call, the string `$data` contains just a bunch of 8-bit bytes. As far as Perl is concerned, the encoding of the string remains as "system-native 8-bit bytes".

  You might relate this to a fictional 'Translate' module:

  ```
  use Translate;
  my $phrase = "Yes";
  Translate::from_to($phrase, 'english', 'deutsch');
  ## phrase now contains "Ja"
  ```

  The contents of the string changes, but not the nature of the string. Perl doesn't know any more after the call than before that the contents of the string indicates the affirmative.

Back to converting data. If you have (or want) data in your system's native 8-bit encoding (e.g. Latin-1, EBCDIC, etc.), you can use pack/unpack to convert to/from Unicode.

```
$native_string = pack("W*", unpack("U*", $Unicode_string));
$Unicode_string = pack("U*", unpack("W*", $native_string));
```

If you have a sequence of bytes you **know** is valid UTF-8, but Perl doesn't know it yet, you can make Perl a believer, too:

```
use Encode 'decode_utf8';
$Unicode = decode_utf8($bytes);
```

or:

```
$Unicode = pack("U0a*", $bytes);
```

You can find the bytes that make up a UTF-8 sequence with

```
@bytes = unpack("C*", $Unicode_string)
```

and you can create well-formed Unicode with

```
$Unicode_string = pack("U*", 0xff, ...)
```

- How Do I Display Unicode? How Do I Input Unicode?

  See <http://www.alanwood.net/unicode/> and <http://www.cl.cam.ac.uk/~mgk25/unicode.html>

- How Does Unicode Work With Traditional Locales?

  If your locale is a UTF-8 locale, starting in Perl v5.20, Perl works well for all categories except `LC_COLLATE` dealing with sorting and the `cmp` operator.

  For other locales, starting in Perl 5.16, you can specify

  ```
  use locale ':not_characters';
  ```

  to get Perl to work well with them. The catch is that you have to translate from the locale character set to/from Unicode yourself. See "Unicode I/O" above for how to

  ```
  use open ':locale';
  ```

  to accomplish this, but full details are in "Unicode and UTF-8" in perllocale(1), including gotchas that happen if you don't specify `:not_characters`.

Hexadecimal Notation

The Unicode standard prefers using hexadecimal notation because that more clearly shows the division of Unicode into blocks of 256 characters. Hexadecimal is also simply shorter than decimal. You can use decimal notation, too, but learning to use hexadecimal just makes life easier with the Unicode standard. The `U+HHHH` notation uses hexadecimal, for example.

The `0x` prefix means a hexadecimal number, the digits are 0-9 *and* a-f (or A-F, case doesn't matter). Each hexadecimal digit represents four bits, or half a byte. `print 0x..., "\n"` will show a hexadecimal number in decimal, and `printf "%x\n", $decimal` will show a decimal number in hexadecimal. If you have just the "hex digits" of a hexadecimal number, you can use the `hex()` function.

```
print 0x0009, "\n"; # 9
print 0x000a, "\n"; # 10
print 0x000f, "\n"; # 15
print 0x0010, "\n"; # 16
print 0x0011, "\n"; # 17
print 0x0100, "\n"; # 256

print 0x0041, "\n"; # 65
```

```
    printf "%x\n", 65; # 41
    printf "%#x\n", 65; # 0x41

    print hex("41"), "\n"; # 65
```

Further Resources

- Unicode Consortium

  <http://www.unicode.org/>
- Unicode FAQ

  <http://www.unicode.org/unicode/faq/>
- Unicode Glossary

  <http://www.unicode.org/glossary/>
- Unicode Recommended Reading List

  The Unicode Consortium has a list of articles and books, some of which give a much more in depth treatment of Unicode: <http://unicode.org/resources/readinglist.html>
- Unicode Useful Resources

  <http://www.unicode.org/unicode/onlinedat/resources.html>
- Unicode and Multilingual Support in HTML, Fonts, Web Browsers and Other Applications

  <http://www.alanwood.net/unicode/>
- UTF-8 and Unicode FAQ for Unix/Linux

  <http://www.cl.cam.ac.uk/~mgk25/unicode.html>
- Legacy Character Sets

  <http://www.czyborra.com/> <http://www.eki.ee/letter/>
- You can explore various information from the Unicode data files using the `Unicode::UCD` module.

UNICODE IN OLDER PERLS

If you cannot upgrade your Perl to 5.8.0 or later, you can still do some Unicode processing by using the modules `Unicode::String` `Unicode::Map8` and `Unicode::Map` available from CPAN. If you have the GNU recode installed, you can also use the Perl front-end `Convert::Recode` for character conversions.

The following are fast conversions from ISO 8859-1 (Latin-1) bytes to UTF-8 bytes and back, the code works even with older Perl 5 versions.

```
    # ISO 8859-1 to UTF-8
    s/([\x80-\xFF])/chr(0xC0|ord($1)>>6).chr(0x80|ord($1)&0x3F)/eg;

    # UTF-8 to ISO 8859-1
    s/([\xC2\xC3])([\x80-\xBF])/chr(ord($1)<<6&0xC0|ord($2)&0x3F)/eg;
```

SEE ALSO

perlunitut(1), perlunicode(1), Encode, open, utf8, bytes, perlretut(1), perlrun(1), Unicode::Collate, Unicode::Normalize, Unicode::UCD

ACKNOWLEDGMENTS

Thanks to the kind readers of the perl5-porters@perl.org, perl-unicode@perl.org, linux-utf8@nl.linux.org, and unicore@unicode.org mailing lists for their valuable feedback.

AUTHOR, COPYRIGHT, AND LICENSE