

**NAME**

perllocale - Perl locale handling (internationalization and localization)

**DESCRIPTION**

In the beginning there was ASCII, the “American Standard Code for Information Interchange”, which works quite well for Americans with their English alphabet and dollar-denominated currency. But it doesn’t work so well even for other English speakers, who may use different currencies, such as the pound sterling (as the symbol for that currency is not in ASCII); and it’s hopelessly inadequate for many of the thousands of the world’s other languages.

To address these deficiencies, the concept of locales was invented (formally the ISO C, XPG4, POSIX 1.c “locale system”). And applications were and are being written that use the locale mechanism. The process of making such an application take account of its users’ preferences in these kinds of matters is called **internationalization** (often abbreviated as **i18n**); telling such an application about a particular set of preferences is known as **localization** (**l10n**).

Perl has been extended to support the locale system. This is controlled per application by using one pragma, one function call, and several environment variables.

Unfortunately, there are quite a few deficiencies with the design (and often, the implementations) of locales. Unicode was invented (see perlunitut for an introduction to that) in part to address these design deficiencies, and nowadays, there is a series of “UTF-8 locales”, based on Unicode. These are locales whose character set is Unicode, encoded in UTF-8. Starting in v5.20, Perl fully supports UTF-8 locales, except for sorting and string comparisons like `lt` and `ge`. (Use `Unicode::Collate` for these.) Perl continues to support the old non UTF-8 locales as well. There are currently no UTF-8 locales for EBCDIC platforms.

(Unicode is also creating CLDR, the “Common Locale Data Repository”, <<http://cldr.unicode.org/>> which includes more types of information than are available in the POSIX locale system. At the time of this writing, there was no CPAN module that provides access to this XML-encoded data. However, it is possible to compute the POSIX locale data from them, and earlier CLDR versions had these already extracted for you as UTF-8 locales <<http://unicode.org/Public/cldr/2.0.1/>>.)

**WHAT IS A LOCALE**

A locale is a set of data that describes various aspects of how various communities in the world categorize their world. These categories are broken down into the following types (some of which include a brief note here):

Category `LC_NUMERIC`: Numeric formatting

This indicates how numbers should be formatted for human readability, for example the character used as the decimal point.

Category `LC_MONETARY`: Formatting of monetary amounts

Category `LC_TIME`: Date/Time formatting

Category `LC_MESSAGES`: Error and other messages

This is used by Perl itself only for accessing operating system error messages via `$!` and `$^E`.

Category `LC_COLLATE`: Collation

This indicates the ordering of letters for comparison and sorting. In Latin alphabets, for example, “b”, generally follows “a”.

Category `LC_CTYPE`: Character Types

This indicates, for example if a character is an uppercase letter.

Other categories

Some platforms have other categories, dealing with such things as measurement units and paper sizes. None of these are used directly by Perl, but outside operations that Perl interacts with may use these. See “Not within the scope of ”use locale” below.

More details on the categories used by Perl are given below in “LOCALE CATEGORIES”.

Together, these categories go a long way towards being able to customize a single program to run in many different locations. But there are deficiencies, so keep reading.

## PREPARING TO USE LOCALES

Perl itself (outside the POSIX module) will not use locales unless specifically requested to (but again note that Perl may interact with code that does use them). Even if there is such a request, **all** of the following must be true for it to work properly:

- **Your operating system must support the locale system.** If it does, you should find that the `setlocale()` function is a documented part of its C library.
- **Definitions for locales that you use must be installed.** You, or your system administrator, must make sure that this is the case. The available locales, the location in which they are kept, and the manner in which they are installed all vary from system to system. Some systems provide only a few, hard-wired locales and do not allow more to be added. Others allow you to add “canned” locales provided by the system supplier. Still others allow you or the system administrator to define and add arbitrary locales. (You may have to ask your supplier to provide canned locales that are not delivered with your operating system.) Read your system documentation for further illumination.
- **Perl must believe that the locale system is supported.** If it does, `perl -V:d_setlocale` will say that the value for `d_setlocale` is `define`.

If you want a Perl application to process and present your data according to a particular locale, the application code should include the `uselocale` pragma (see “The `uselocale` pragma”) where appropriate, and **at least one** of the following must be true:

1. **The locale-determining environment variables (see “ENVIRONMENT”) must be correctly set up** at the time the application is started, either by yourself or by whomever set up your system account; or
2. **The application must set its own locale** using the method described in “The `setlocale` function”.

## USING LOCALES

### The `uselocale` pragma

WARNING! Do NOT use this pragma in scripts that have multiple threads active. The locale is not local to a single thread. Another thread may change the locale at any time, which could cause at a minimum that a given thread is operating in a locale it isn’t expecting to be in. On some platforms, segfaults can also occur. The locale change need not be explicit; some operations cause perl to change the locale itself. You are vulnerable simply by having done a `uselocale`.

By default, Perl itself (outside the POSIX module) ignores the current locale. The `uselocale` pragma tells Perl to use the current locale for some operations. Starting in v5.16, there are optional parameters to this pragma, described below, which restrict which operations are affected by it.

The current locale is set at execution time by `setlocale()` described below. If that function hasn’t yet been called in the course of the program’s execution, the current locale is that which was determined by the “ENVIRONMENT” in effect at the start of the program. If there is no valid environment, the current locale is whatever the system default has been set to. On POSIX systems, it is likely, but not necessarily, the “C” locale. On Windows, the default is set via the computer’s `ControlPanel->RegionalandLanguageOptions` (or its current equivalent).

The operations that are affected by locale are:

### Not within the scope of `uselocale`

Only certain operations originating outside Perl should be affected, as follows:

- The current locale is used when going outside of Perl with operations like `system()` or `qx//`, if those operations are locale-sensitive.
- Also Perl gives access to various C library functions through the POSIX module. Some of those functions are always affected by the current locale. For example, `POSIX::strftime()` uses `LC_TIME`; `POSIX::strtod()` uses `LC_NUMERIC`; `POSIX::strcoll()` and `POSIX::strxfrm()` use `LC_COLLATE`. All such functions will behave according to the current underlying locale, even if that locale isn’t exposed to Perl space.

- XS modules for all categories but `LC_NUMERIC` get the underlying locale, and hence any C library functions they call will use that underlying locale. For more discussion, see “CAVEATS” in `perlxs`.

Note that all C programs (including the perl interpreter, which is written in C) always have an underlying locale. That locale is the “C” locale unless changed by a call to `setlocale()`. When Perl starts up, it changes the underlying locale to the one which is indicated by the “ENVIRONMENT”. When using the POSIX module or writing XS code, it is important to keep in mind that the underlying locale may be something other than “C”, even if the program hasn’t explicitly changed it.

### Lingering effects of `uselocale`

Certain Perl operations that are set-up within the scope of a `use locale` retain that effect even outside the scope. These include:

- The output format of a `write()` is determined by an earlier format declaration (“format” in `perlfunc`), so whether or not the output is affected by locale is determined by if the `format()` is within the scope of a `use locale`, not whether the `write()` is.
- Regular expression patterns can be compiled using `qr//` with actual matching deferred to later. Again, it is whether or not the compilation was done within the scope of `use locale` that determines the match behavior, not if the matches are done within such a scope or not.

### Under “`use locale`”;

- All the above operations
- **Format declarations** (“format” in `perlfunc`) and hence any subsequent `write()`s use `LC_NUMERIC`.
- **stringification and output** use `LC_NUMERIC`. These include the results of `print()`, `printf()`, `say()`, and `sprintf()`.
- **The comparison operators** (`lt`, `le`, `cmp`, `ge`, and `gt`) use `LC_COLLATE`. `sort()` is also affected if used without an explicit comparison function, because it uses `cmp` by default.  
**Note:** `eq` and `ne` are unaffected by locale: they always perform a char-by-char comparison of their scalar operands. What’s more, if `cmp` finds that its operands are equal according to the collation sequence specified by the current locale, it goes on to perform a char-by-char comparison, and only returns 0 (equal) if the operands are char-for-char identical. If you really want to know whether two strings — which `eq` and `cmp` may consider different — are equal as far as collation in the locale is concerned, see the discussion in “Category `LC_COLLATE`: Collation”.
- **Regular expressions and case-modification functions** (`uc()`, `lc()`, `ucfirst()`, and `lcfirst()`) use `LC_CTYPE`
- **The variables \$!** (and its synonyms `$ERRNO` and `$OS_ERROR`) **and \$^E** (and its synonym `$EXTENDED_OS_ERROR`) when used as strings use `LC_MESSAGES`.

The default behavior is restored with the `nolocale` pragma, or upon reaching the end of the block enclosing `use locale`. Note that `use locale` calls may be nested, and that what is in effect within an inner scope will revert to the outer scope’s rules at the end of the inner scope.

The string result of any operation that uses locale information is tainted, as it is possible for a locale to be untrustworthy. See “SECURITY”.

Starting in Perl v5.16 in a very limited way, and more generally in v5.22, you can restrict which category or categories are enabled by this particular instance of the pragma by adding parameters to it. For example,

```
use locale qw(:ctype :numeric);
```

enables locale awareness within its scope of only those operations (listed above) that are affected by `LC_CTYPE` and `LC_NUMERIC`.

The possible categories are: `:collate`, `:ctype`, `:messages`, `:monetary`, `:numeric`, `:time`, and the pseudo category `:characters` (described below).

Thus you can say

```
use locale ':messages';
```

and only `$!` and `$^E` will be locale aware. Everything else is unaffected.

Since Perl doesn't currently do anything with the `LC_MONETARY` category, specifying `:monetary` does effectively nothing. Some systems have other categories, such as `LC_PAPER_SIZE`, but Perl also doesn't know anything about them, and there is no way to specify them in this pragma's arguments.

You can also easily say to use all categories but one, by either, for example,

```
use locale '!ctype';
use locale ':not_ctype';
```

both of which mean to enable locale awareness of all categories but `LC_CTYPE`. Only one category argument may be specified in a `uselocale` if it is of the negated form.

Prior to v5.22 only one form of the pragma with arguments is available:

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

(and you have to say `not_`; you can't use the bang `!` form). This pseudo category is a shorthand for specifying both `:collate` and `:ctype`. Hence, in the negated form, it is nearly the same thing as saying

```
use locale qw(:messages :monetary :numeric :time);
```

We use the term “nearly”, because `:not_characters` also turns on `usefeature 'unicode_strings'` within its scope. This form is less useful in v5.20 and later, and is described fully in “Unicode and UTF-8”, but briefly, it tells Perl to not use the character portions of the locale definition, that is the `LC_CTYPE` and `LC_COLLATE` categories. Instead it will use the native character set (extended by Unicode). When using this parameter, you are responsible for getting the external character set translated into the native/Unicode one (which it already will be if it is one of the increasingly popular UTF-8 locales). There are convenient ways of doing this, as described in “Unicode and UTF-8”.

### The `setlocale` function

WARNING! Do NOT use this function in a thread. The locale will change in all other threads at the same time, and should your thread get paused by the operating system, and another started, that thread will not have the locale it is expecting. On some platforms, there can be a race leading to segfaults if two threads call this function nearly simultaneously.

You can switch locales as often as you wish at run time with the `POSIX::setlocale()` function:

```
# Import locale-handling tool set from POSIX module.
# This example uses: setlocale -- the function call
# LC_CTYPE -- explained below
# (Showing the testing for success/failure of operations is
# omitted in these examples to avoid distracting from the main
# point)

use POSIX qw(locale_h);
use locale;
my $old_locale;

# query and save the old locale
$old_locale = setlocale(LC_CTYPE);

setlocale(LC_CTYPE, "fr_CA.ISO8859-1");
# LC_CTYPE now in locale "French, Canada, codeset ISO 8859-1"
```

```

setlocale(LC_CTYPE, "");
# LC_CTYPE now reset to the default defined by the
# LC_ALL/LC_CTYPE/LANG environment variables, or to the system
# default. See below for documentation.

# restore the old locale
setlocale(LC_CTYPE, $old_locale);

```

The first argument of `setlocale()` gives the **category**, the second the **locale**. The category tells in what aspect of data processing you want to apply locale-specific rules. Category names are discussed in “LOCALE CATEGORIES” and “ENVIRONMENT”. The locale is the name of a collection of customization information corresponding to a particular combination of language, country or territory, and codeset. Read on for hints on the naming of locales: not all systems name locales as in the example.

If no second argument is provided and the category is something other than `LC_ALL`, the function returns a string naming the current locale for the category. You can use this value as the second argument in a subsequent call to `setlocale()`, **but** on some platforms the string is opaque, not something that most people would be able to decipher as to what locale it means.

If no second argument is provided and the category is `LC_ALL`, the result is implementation-dependent. It may be a string of concatenated locale names (separator also implementation-dependent) or a single locale name. Please consult your [setlocale\(3\)](#) man page for details.

If a second argument is given and it corresponds to a valid locale, the locale for the category is set to that value, and the function returns the now-current locale value. You can then use this in yet another call to `setlocale()`. (In some implementations, the return value may sometimes differ from the value you gave as the second argument—think of it as an alias for the value you gave.)

As the example shows, if the second argument is an empty string, the category’s locale is returned to the default specified by the corresponding environment variables. Generally, this results in a return to the default that was in force when Perl started up: changes to the environment made by the application after startup may or may not be noticed, depending on your system’s C library.

Note that when a form of `use locale` that doesn’t include all categories is specified, Perl ignores the excluded categories.

If `set_locale()` fails for some reason (for example, an attempt to set to a locale unknown to the system), the locale for the category is not changed, and the function returns `undef`.

For further information about the categories, consult [setlocale\(3\)](#).

### Finding locales

For locales available in your system, consult also [setlocale\(3\)](#) to see whether it leads to the list of available locales (search for the *SEE ALSO* section). If that fails, try the following command lines:

```

locale -a

nlsinfo

ls /usr/lib/nls/loc

ls /usr/lib/locale

ls /usr/lib/nls

ls /usr/share/locale

```

and see whether they list something resembling these

```

en_US.ISO8859-1 de_DE.ISO8859-1 ru_RU.ISO8859-5
en_US.iso88591 de_DE.iso88591 ru_RU.iso88595
en_US de_DE ru_RU
en de ru
english german russian
english.iso88591 german.iso88591 russian.iso88595
english.roman8 russian.koi8r

```

Sadly, even though the calling interface for `setlocale()` has been standardized, names of locales and the directories where the configuration resides have not been. The basic form of the name is *language\_territory.codeset*, but the latter parts after *language* are not always present. The *language* and *country* are usually from the standards **ISO 3166** and **ISO 639**, the two-letter abbreviations for the countries and the languages of the world, respectively. The *codeset* part often mentions some **ISO 8859** character set, the Latin codesets. For example, ISO 8859-1 is the so-called “Western European codeset” that can be used to encode most Western European languages adequately. Again, there are several ways to write even the name of that one standard. Lamentably.

Two special locales are worth particular mention: “C” and “POSIX”. Currently these are effectively the same locale: the difference is mainly that the first one is defined by the C standard, the second by the POSIX standard. They define the **default locale** in which every program starts in the absence of locale information in its environment. (The *default* default locale, if you will.) Its language is (American) English and its character codeset ASCII or, rarely, a superset thereof (such as the “DEC Multinational Character Set (DEC-MCS)”). **Warning.** The C locale delivered by some vendors may not actually exactly match what the C standard calls for. So beware.

**NOTE:** Not all systems have the “POSIX” locale (not all systems are POSIX-conformant), so use “C” when you need explicitly to specify this default locale.

## LOCALE PROBLEMS

You may encounter the following warning message at Perl startup:

```

perl: warning: Setting locale failed.
perl: warning: Please check that your locale settings:
LC_ALL = "En_US",
LANG = (unset)
are supported and installed on your system.
perl: warning: Falling back to the standard locale ("C").

```

This means that your locale settings had `LC_ALL` set to “En\_US” and `LANG` exists but has no value. Perl tried to believe you but could not. Instead, Perl gave up and fell back to the “C” locale, the default locale that is supposed to work no matter what. (On Windows, it first tries falling back to the system default locale.) This usually means your locale settings were wrong, they mention locales your system has never heard of, or the locale installation in your system has problems (for example, some system files are broken or missing). There are quick and temporary fixes to these problems, as well as more thorough and lasting fixes.

### Testing for broken locales

If you are building Perl from source, the Perl test suite file *lib/locale.t* can be used to test the locales on your system. Setting the environment variable `PERL_DEBUG_FULL_TEST` to 1 will cause it to output detailed results. For example, on Linux, you could say

```
PERL_DEBUG_FULL_TEST=1 ./perl -T -Ilib lib/locale.t > locale.log 2>&1
```

Besides many other tests, it will test every locale it finds on your system to see if they conform to the POSIX standard. If any have errors, it will include a summary near the end of the output of which locales passed all its tests, and which failed, and why.

### Temporarily fixing locale problems

The two quickest fixes are either to render Perl silent about any locale inconsistencies or to run Perl under the default locale “C”.

Perl's moaning about locale problems can be silenced by setting the environment variable `PERL_BADLANG` to `"0"` or `""`. This method really just sweeps the problem under the carpet: you tell Perl to shut up even when Perl sees that something is wrong. Do not be surprised if later something locale-dependent misbehaves.

Perl can be run under the `"C"` locale by setting the environment variable `LC_ALL` to `"C"`. This method is perhaps a bit more civilized than the `PERL_BADLANG` approach, but setting `LC_ALL` (or other locale variables) may affect other programs as well, not just Perl. In particular, external programs run from within Perl will see these changes. If you make the new settings permanent (read on), all programs you run see the changes. See `"ENVIRONMENT"` for the full list of relevant environment variables and `"USING LOCALES"` for their effects in Perl. Effects in other programs are easily deducible. For example, the variable `LC_COLLATE` may well affect your `sort` program (or whatever the program that arranges "records" alphabetically in your system is called).

You can test out changing these variables temporarily, and if the new settings seem to help, put those settings into your shell startup files. Consult your local documentation for the exact details. For Bourne-like shells (**sh**, **ksh**, **bash**, **zsh**):

```
LC_ALL=en_US.ISO8859-1
export LC_ALL
```

This assumes that we saw the locale `"en_US.ISO8859-1"` using the commands discussed above. We decided to try that instead of the above faulty locale `"En_US"`--and in Cshish shells (**csh**, **tcsh**)

```
setenv LC_ALL en_US.ISO8859-1
```

or if you have the `"env"` application you can do (in any shell)

```
env LC_ALL=en_US.ISO8859-1 perl ...
```

If you do not know what shell you have, consult your local helpdesk or the equivalent.

### Permanently fixing locale problems

The slower but superior fixes are when you may be able to yourself fix the misconfiguration of your own environment variables. The mis(sing)configuration of the whole system's locales usually requires the help of your friendly system administrator.

First, see earlier in this document about `"Finding locales"`. That tells how to find which locales are really supported—and more importantly, installed—on your system. In our example error message, environment variables affecting the locale are listed in the order of decreasing importance (and unset variables do not matter). Therefore, having `LC_ALL` set to `"En_US"` must have been the bad choice, as shown by the error message. First try fixing locale settings listed first.

Second, if using the listed commands you see something **exactly** (prefix matches do not count and case usually counts) like `"En_US"` without the quotes, then you should be okay because you are using a locale name that should be installed and available in your system. In this case, see `"Permanently fixing your system's locale configuration"`.

### Permanently fixing your system's locale configuration

This is when you see something like:

```
perl: warning: Please check that your locale settings:
LC_ALL = "En_US",
LANG = (unset)
are supported and installed on your system.
```

but then cannot see that `"En_US"` listed by the above-mentioned commands. You may see things like `"en_US.ISO8859-1"`, but that isn't the same. In this case, try running under a locale that you can list and which somehow matches what you tried. The rules for matching locale names are a bit vague because standardization is weak in this area. See again the `"Finding locales"` about general rules.

**Fixing system locale configuration**

Contact a system administrator (preferably your own) and report the exact error message you get, and ask them to read this same documentation you are now reading. They should be able to check whether there is something wrong with the locale configuration of the system. The “Finding locales” section is unfortunately a bit vague about the exact commands and places because these things are not that standardized.

**The localeconv function**

The `POSIX::localeconv()` function allows you to get particulars of the locale-dependent numeric formatting information specified by the current underlying `LC_NUMERIC` and `LC_MONETARY` locales (regardless of whether called from within the scope of `uselocale` or not). (If you just want the name of the current locale for a particular category, use `POSIX::setlocale()` with a single parameter—see “The `setlocale` function”.)

```
use POSIX qw(locale_h);

# Get a reference to a hash of locale-dependent info
$locale_values = localeconv();

# Output sorted list of the values
for (sort keys %$locale_values) {
    printf "%-20s = %s\n", $_, $locale_values->{$_}
}
```

`localeconv()` takes no arguments, and returns **a reference** to a hash. The keys of this hash are variable names for formatting, such as `decimal_point` and `thousands_sep`. The values are the corresponding, er, values. See “`localeconv`” in POSIX for a longer example listing the categories an implementation might be expected to provide; some provide more and others fewer. You don’t need an explicit `use locale`, because `localeconv()` always observes the current locale.

Here’s a simple-minded example program that rewrites its command-line parameters as integers correctly formatted in the current locale:

```
use POSIX qw(locale_h);

# Get some of locale's numeric formatting parameters
my ($thousands_sep, $grouping) =
    @{localeconv()}{'thousands_sep', 'grouping'};

# Apply defaults if values are missing
$thousands_sep = ',' unless $thousands_sep;

# grouping and mon_grouping are packed lists
# of small integers (characters) telling the
# grouping (thousand_seps and mon_thousand_seps
# being the group dividers) of numbers and
# monetary quantities. The integers' meanings:
# 255 means no more grouping, 0 means repeat
# the previous grouping, 1-254 means use that
# as the current grouping. Grouping goes from
# right to left (low to high digits). In the
# below we cheat slightly by never using anything
# else than the first grouping (whatever that is).
if ($grouping) {
    @grouping = unpack("C*", $grouping);
} else {
    @grouping = (3);
}
```

```

}

# Format command line params for current locale
for (@ARGV) {
    $_ = int; # Chop non-integer part
    1 while
    s/(\d)(\d{${$grouping[0]}}($|${thousands_sep}))/${1}${thousands_sep}${2}/;
    print "$_";
}
print "\n";

```

Note that if the platform doesn't have LC\_NUMERIC and/or LC\_MONETARY available or enabled, the corresponding elements of the hash will be missing.

### I18N::Langinfo

Another interface for querying locale-dependent information is the `I18N::Langinfo::langinfo()` function, available at least in Unix-like systems and VMS.

The following example will import the `langinfo()` function itself and three constants to be used as arguments to `langinfo()`: a constant for the abbreviated first day of the week (the numbering starts from Sunday = 1) and two more constants for the affirmative and negative answers for a yes/no question in the current locale.

```

use I18N::Langinfo qw(langinfo ABDAY_1 YESSTR NOSTR);

my ($abday_1, $yesstr, $nostr)
= map { langinfo } qw(ABDAY_1 YESSTR NOSTR);

print "$abday_1? [$yesstr/$nostr] ";

```

In other words, in the “C” (or English) locale the above will probably print something like:

```
Sun? [yes/no]
```

See [I18N::Langinfo](#) for more information.

## LOCALE CATEGORIES

The following subsections describe basic locale categories. Beyond these, some combination categories allow manipulation of more than one basic category at a time. See “ENVIRONMENT” for a discussion of these.

### Category LC\_COLLATE: Collation

In the scope of a `uselocale` form that includes collation, Perl looks to the LC\_COLLATE environment variable to determine the application's notions on collation (ordering) of characters. For example, “b” follows “a” in Latin alphabets, but where do “á” and “â” belong? And while “color” follows “chocolate” in English, what about in traditional Spanish?

The following collations all make sense and you may meet any of them if you `use locale`.

```

A B C D E a b c d e
A a B b C c D d E e
a A b B c C d D e E
a b c d e A B C D E

```

Here is a code snippet to tell what “word” characters are in the current locale, in that locale's order:

```

use locale;
print +(sort grep /\w/, map { chr } 0..255), "\n";

```

Compare this with the characters that you see and their order if you state explicitly that the locale should be ignored:

```
no locale;
print +(sort grep /\w/, map { chr } 0..255), "\n";
```

This machine-native collation (which is what you get unless `uselocale` has appeared earlier in the same block) must be used for sorting raw binary data, whereas the locale-dependent collation of the first example is useful for natural text.

As noted in “USING LOCALES”, `cmp` compares according to the current collation locale when `uselocale` is in effect, but falls back to a char-by-char comparison for strings that the locale says are equal. You can use `POSIX::strcoll()` if you don’t want this fall-back:

```
use POSIX qw(strcoll);
$equal_in_locale =
!strcoll("space and case ignored", "SpaceAndCaseIgnored");
```

`$equal_in_locale` will be true if the collation locale specifies a dictionary-like ordering that ignores space characters completely and which folds case.

Perl currently only supports single-byte locales for `LC_COLLATE`. This means that a UTF-8 locale likely will just give you machine-native ordering. Use [Unicode::Collate](#) for the full implementation of the Unicode Collation Algorithm.

If you have a single string that you want to check for “equality in locale” against several others, you might think you could gain a little efficiency by using `POSIX::strxfrm()` in conjunction with `eq`:

```
use POSIX qw(strxfrm);
$xfrm_string = strxfrm("Mixed-case string");
print "locale collation ignores spaces\n"
if $xfrm_string eq strxfrm("Mixed-casestring");
print "locale collation ignores hyphens\n"
if $xfrm_string eq strxfrm("Mixedcase string");
print "locale collation ignores case\n"
if $xfrm_string eq strxfrm("mixed-case string");
```

`strxfrm()` takes a string and maps it into a transformed string for use in char-by-char comparisons against other transformed strings during collation. “Under the hood”, locale-affected Perl comparison operators call `strxfrm()` for both operands, then do a char-by-char comparison of the transformed strings. By calling `strxfrm()` explicitly and using a non locale-affected comparison, the example attempts to save a couple of transformations. But in fact, it doesn’t save anything: Perl magic (see “Magic Variables” in `perlguts`) creates the transformed version of a string the first time it’s needed in a comparison, then keeps this version around in case it’s needed again. An example rewritten the easy way with `cmp` runs just about as fast. It also copes with null characters embedded in strings; if you call `strxfrm()` directly, it treats the first null it finds as a terminator. don’t expect the transformed strings it produces to be portable across systems—or even from one revision of your operating system to the next. In short, don’t call `strxfrm()` directly: let Perl do it for you.

Note: `uselocale` isn’t shown in some of these examples because it isn’t needed: `strcoll()` and `strxfrm()` are POSIX functions which use the standard system-supplied `libc` functions that always obey the current `LC_COLLATE` locale.

### Category `LC_CTYPE`: Character Types

In the scope of a `uselocale` form that includes `LC_CTYPE`, Perl obeys the `LC_CTYPE` locale setting. This controls the application’s notion of which characters are alphabetic, numeric, punctuation, *etc.* This affects Perl’s `\w` regular expression metanotation, which stands for alphanumeric characters—that is, alphabetic, numeric, and the platform’s native underscore. (Consult [perlre\(1\)](#) for more information about regular expressions.) Thanks to `LC_CTYPE`, depending on your locale setting, characters like “æ”, “ð”, “ß”, and “ø” may be understood as `\w` characters. It also affects things like `\s`, `\D`, and the POSIX character classes, like `[[:graph:]]`. (See [perlrecharclass\(1\)](#) for more information on all these.)

The `LC_CTYPE` locale also provides the map used in transliterating characters between lower and uppercase. This affects the case-mapping functions—`fc()`, `lc()`, `lcfirst()`, `uc()`, and

`ucfirst()`; case-mapping interpolation with `\F`, `\l`, `\L`, `\u`, or `\U` in double-quoted strings and `s///` substitutions; and case-independent regular expression pattern matching using the `i` modifier.

Starting in v5.20, Perl supports UTF-8 locales for `LC_CTYPE`, but otherwise Perl only supports single-byte locales, such as the ISO 8859 series. This means that wide character locales, for example for Asian languages, are not well-supported. (If the platform has the capability for Perl to detect such a locale, starting in Perl v5.22, Perl will warn, default enabled, using the `locale` warning category, whenever such a locale is switched into.) The UTF-8 locale support is actually a superset of POSIX locales, because it is really full Unicode behavior as if no `LC_CTYPE` locale were in effect at all (except for tainting; see “SECURITY”). POSIX locales, even UTF-8 ones, are lacking certain concepts in Unicode, such as the idea that changing the case of a character could expand to be more than one character. Perl in a UTF-8 locale, will give you that expansion. Prior to v5.20, Perl treated a UTF-8 locale on some platforms like an ISO 8859-1 one, with some restrictions, and on other platforms more like the “C” locale. For releases v5.16 and v5.18, `uselocale 'not_characters'` could be used as a workaround for this (see “Unicode and UTF-8”).

Note that there are quite a few things that are unaffected by the current locale. Any literal character is the native character for the given platform. Hence `'A'` means the character at code point 65 on ASCII platforms, and 193 on EBCDIC. That may or may not be an `'A'` in the current locale, if that locale even has an `'A'`. Similarly, all the escape sequences for particular characters, `\n` for example, always mean the platform’s native one. This means, for example, that `\N` in regular expressions (every character but new-line) works on the platform character set.

Starting in v5.22, Perl will by default warn when switching into a locale that redefines any ASCII printable character (plus `\t` and `\n`) into a different class than expected. This is likely to happen on modern locales only on EBCDIC platforms, where, for example, a CCSID 0037 locale on a CCSID 1047 machine moves `" | "`, but it can happen on ASCII platforms with the ISO 646 and other 7-bit locales that are essentially obsolete. Things may still work, depending on what features of Perl are used by the program. For example, in the example from above where `" | "` becomes a `\w`, and there are no regular expressions where this matters, the program may still work properly. The warning lists all the characters that it can determine could be adversely affected.

**Note:** A broken or malicious `LC_CTYPE` locale definition may result in clearly ineligible characters being considered to be alphanumeric by your application. For strict matching of (mundane) ASCII letters and digits—for example, in command strings—locale-aware applications should use `\w` with the `/a` regular expression modifier. See “SECURITY”.

### Category `LC_NUMERIC`: Numeric Formatting

After a proper `POSIX::setlocale()` call, and within the scope of a `use locale` form that includes numerics, Perl obeys the `LC_NUMERIC` locale information, which controls an application’s idea of how numbers should be formatted for human readability. In most implementations the only effect is to change the character used for the decimal point—perhaps from `“.”` to `“,”`. The functions aren’t aware of such niceties as thousands separation and so on. (See “The `localeconv` function” if you care about these things.)

```
use POSIX qw(strtod setlocale LC_NUMERIC);
use locale;

setlocale LC_NUMERIC, "";

$N = 5/2; # Assign numeric 2.5 to $N

$a = " $N"; # Locale-dependent conversion to string

print "half five is $N\n"; # Locale-dependent output

printf "half five is %g\n", $N; # Locale-dependent output
```

```
print "DECIMAL POINT IS COMMA\n"
if $n == (strtod("2,5"))[0]; # Locale-dependent conversion
```

See also [I18N::Langinfo](#) and RADIXCHAR.

### Category LC\_MONETARY: Formatting of monetary amounts

The C standard defines the LC\_MONETARY category, but not a function that is affected by its contents. (Those with experience of standards committees will recognize that the working group decided to punt on the issue.) Consequently, Perl essentially takes no notice of it. If you really want to use LC\_MONETARY, you can query its contents—see “The localeconv function”—and use the information that it returns in your application’s own formatting of currency amounts. However, you may well find that the information, voluminous and complex though it may be, still does not quite meet your requirements: currency formatting is a hard nut to crack.

See also [I18N::Langinfo](#) and CRNCYSTR.

### LC\_TIME

Output produced by `POSIX::strftime()`, which builds a formatted human-readable date/time string, is affected by the current LC\_TIME locale. Thus, in a French locale, the output produced by the `%B` format element (full month name) for the first month of the year would be “janvier”. Here’s how to get a list of long month names in the current locale:

```
use POSIX qw(strftime);
for (0..11) {
    $long_month_name[$_] =
    strftime("%B", 0, 0, 0, 1, $_, 96);
}
```

Note: `use locale` isn’t needed in this example: `strftime()` is a POSIX function which uses the standard system-supplied `libc` function that always obeys the current LC\_TIME locale.

See also [I18N::Langinfo](#) and `ABDAY_1..ABDAY_7`, `DAY_1..DAY_7`, `ABMON_1..ABMON_12`, and `ABMON_1..ABMON_12`.

### Other categories

The remaining locale categories are not currently used by Perl itself. But again note that things Perl interacts with may use these, including extensions outside the standard Perl distribution, and by the operating system and its utilities. Note especially that the string value of `$!` and the error messages given by external utilities may be changed by LC\_MESSAGES. If you want to have portable error codes, use `%!`. See `Errno`.

## SECURITY

Although the main discussion of Perl security issues can be found in [perlsec\(1\)](#), a discussion of Perl’s locale handling would be incomplete if it did not draw your attention to locale-dependent security issues. Locales—particularly on systems that allow unprivileged users to build their own locales—are untrustworthy. A malicious (or just plain broken) locale can make a locale-aware application give unexpected results. Here are a few possibilities:

- Regular expression checks for safe file names or mail addresses using `\w` may be spoofed by an LC\_CTYPE locale that claims that characters such as `">"` and `"|"` are alphanumeric.
- String interpolation with case-mapping, as in, say, `$dest = "C:\U$name.$ext"`, may produce dangerous results if a bogus LC\_CTYPE case-mapping table is in effect.
- A sneaky LC\_COLLATE locale could result in the names of students with “D” grades appearing ahead of those with “A”s.
- An application that takes the trouble to use information in LC\_MONETARY may format debits as if they were credits and vice versa if that locale has been subverted. Or it might make payments in US dollars instead of Hong Kong dollars.
- The date and day names in dates formatted by `strftime()` could be manipulated to advantage by a malicious user able to subvert the LC\_DATE locale. (“Look—it says I wasn’t in the building on

Sunday.”)

Such dangers are not peculiar to the locale system: any aspect of an application’s environment which may be modified maliciously presents similar challenges. Similarly, they are not specific to Perl: any programming language that allows you to write programs that take account of their environment exposes you to these issues.

Perl cannot protect you from all possibilities shown in the examples—there is no substitute for your own vigilance—but, when `use locale` is in effect, Perl uses the tainting mechanism (see `perlsec`) to mark string results that become locale-dependent, and which may be untrustworthy in consequence. Here is a summary of the tainting behavior of operators and functions that may be affected by the locale:

- **Comparison operators** (`lt`, `le`, `ge`, `gt` and `cmp`):

Scalar true/false (or less/equal/greater) result is never tainted.

- **Case-mapping interpolation** (with `\l`, `\L`, `\u`, `\U`, or `\F`)

The result string containing interpolated material is tainted if a `use locale` form that includes `LC_CTYPE` is in effect.

- **Matching operator** (`m//`):

Scalar true/false result never tainted.

All subpatterns, either delivered as a list-context result or as `$1` *etc.*, are tainted if a `use locale` form that includes `LC_CTYPE` is in effect, and the subpattern regular expression contains a locale-dependent construct. These constructs include `\w` (to match an alphanumeric character), `\W` (non-alphanumeric character), `\b` and `\B` (word-boundary and non-boundary, which depend on what `\w` and `\W` match), `\s` (whitespace character), `\S` (non whitespace character), `\d` and `\D` (digits and non-digits), and the POSIX character classes, such as `[:alpha:]` (see “POSIX Character Classes” in `perlrecharclass`).

Tainting is also likely if the pattern is to be matched case-insensitively (via `/i`). The exception is if all the code points to be matched this way are above 255 and do not have folds under Unicode rules to below 256. Tainting is not done for these because Perl only uses Unicode rules for such code points, and those rules are the same no matter what the current locale.

The matched-pattern variables, `$&`, `$`` (pre-match), `$'` (post-match), and `$+` (last match) also are tainted.

- **Substitution operator** (`s///`):

Has the same behavior as the match operator. Also, the left operand of `=~` becomes tainted when a `use locale` form that includes `LC_CTYPE` is in effect, if modified as a result of a substitution based on a regular expression match involving any of the things mentioned in the previous item, or of case-mapping, such as `\l`, `\L`, `\u`, `\U`, or `\F`.

- **Output formatting functions** (`printf()` and `write()`):

Results are never tainted because otherwise even output from `print`, for example `print(1/7)`, should be tainted if `use locale` is in effect.

- **Case-mapping functions** (`lc()`, `lcfirst()`, `uc()`, `ucfirst()`):

Results are tainted if a `use locale` form that includes `LC_CTYPE` is in effect.

- **POSIX locale-dependent functions** (`localeconv()`, `strcoll()`, `strftime()`, `strxfrm()`):

Results are never tainted.

Three examples illustrate locale-dependent tainting. The first program, which ignores its locale, won’t run: a value taken directly from the command line may not be used to name an output file when taint checks are enabled.

```

/usr/local/bin/perl -T
# Run with taint checking

# Command line sanity check omitted...
$tainted_output_file = shift;

open(F, ">$tainted_output_file")
or warn "Open of $tainted_output_file failed: $!\n";

```

The program can be made to run by “laundering” the tainted value through a regular expression: the second example — which still ignores locale information — runs, creating the file named on its command line if it can.

```

/usr/local/bin/perl -T

$tainted_output_file = shift;
$tainted_output_file =~ m%[\w/]+%;
$suntainted_output_file = $&;

open(F, ">$suntainted_output_file")
or warn "Open of $suntainted_output_file failed: $!\n";

```

Compare this with a similar but locale-aware program:

```

/usr/local/bin/perl -T

$tainted_output_file = shift;
use locale;
$tainted_output_file =~ m%[\w/]+%;
$localized_output_file = $&;

open(F, ">$localized_output_file")
or warn "Open of $localized_output_file failed: $!\n";

```

This third program fails to run because `$&` is tainted: it is the result of a match involving `\w` while `use locale` is in effect.

## ENVIRONMENT

### PERL\_SKIP\_LOCALE\_INIT

This environment variable, available starting in Perl v5.20, if set (to any value), tells Perl to not use the rest of the environment variables to initialize with. Instead, Perl uses whatever the current locale settings are. This is particularly useful in embedded environments, see “Using embedded Perl with POSIX locales” in `perlembed`.

### PERL\_BADLANG

A string that can suppress Perl’s warning about failed locale settings at startup. Failure can occur if the locale support in the operating system is lacking (broken) in some way — or if you mistyped the name of a locale when you set up your environment. If this environment variable is absent, or has a value other than “0” or “”, Perl will complain about locale setting failures.

**NOTE:** `PERL_BADLANG` only gives you a way to hide the warning message. The message tells about some problem in your system’s locale support, and you should investigate what the problem is.

### DPKG\_RUNNING\_VERSION

On Debian systems, if the `DPKG_RUNNING_VERSION` environment variable is set (to any value), the locale failure warnings will be suppressed just like with a zero `PERL_BADLANG` setting. This is done to avoid floods of spurious warnings during system upgrades. See <http://bugs.debian.org/508764>.

The following environment variables are not specific to Perl: They are part of the standardized (ISO C, XPG4, POSIX 1.c) `setlocale()` method for controlling an application's opinion on data. Windows is non-POSIX, but Perl arranges for the following to work as described anyway. If the locale given by an environment variable is not valid, Perl tries the next lower one in priority. If none are valid, on Windows, the system default locale is then tried. If all else fails, the "C" locale is used. If even that doesn't work, something is badly broken, but Perl tries to forge ahead with whatever the locale settings might be.

**LC\_ALL** LC\_ALL is the "override-all" locale environment variable. If set, it overrides all the rest of the locale environment variables.

**LANGUAGE** **NOTE:** LANGUAGE is a GNU extension, it affects you only if you are using the GNU libc. This is the case if you are using e.g. Linux. If you are using "commercial" Unixes you are most probably *not* using GNU libc and you can ignore LANGUAGE.

However, in the case you are using LANGUAGE: it affects the language of informational, warning, and error messages output by commands (in other words, it's like LC\_MESSAGES) but it has higher priority than LC\_ALL. Moreover, it's not a single value but instead a "path" (":"-separated list) of *languages* (not locales). See the GNU `gettext` library documentation for more information.

**LC\_CTYPE** In the absence of LC\_ALL, LC\_CTYPE chooses the character type locale. In the absence of both LC\_ALL and LC\_CTYPE, LANG chooses the character type locale.

**LC\_COLLATE** In the absence of LC\_ALL, LC\_COLLATE chooses the collation (sorting) locale. In the absence of both LC\_ALL and LC\_COLLATE, LANG chooses the collation locale.

**LC\_MONETARY** In the absence of LC\_ALL, LC\_MONETARY chooses the monetary formatting locale. In the absence of both LC\_ALL and LC\_MONETARY, LANG chooses the monetary formatting locale.

**LC\_NUMERIC** In the absence of LC\_ALL, LC\_NUMERIC chooses the numeric format locale. In the absence of both LC\_ALL and LC\_NUMERIC, LANG chooses the numeric format.

**LC\_TIME** In the absence of LC\_ALL, LC\_TIME chooses the date and time formatting locale. In the absence of both LC\_ALL and LC\_TIME, LANG chooses the date and time formatting locale.

**LANG** LANG is the "catch-all" locale environment variable. If it is set, it is used as the last resort after the overall LC\_ALL and the category-specific `LC_foo`.

### Examples

The LC\_NUMERIC controls the numeric output:

```
use locale;
use POSIX qw(locale_h); # Imports setlocale() and the LC_ constants.
setlocale(LC_NUMERIC, "fr_FR") or die "Pardon";
printf "%g\n", 1.23; # If the "fr_FR" succeeded, probably shows 1,23.
```

and also how strings are parsed by `POSIX::strtod()` as numbers:

```
use locale;
use POSIX qw(locale_h strtod);
setlocale(LC_NUMERIC, "de_DE") or die "Entschuldigung";
my $x = strtod("2,34") + 5;
print $x, "\n"; # Probably shows 7,34.
```

### NOTES

#### String eval and LC\_NUMERIC

A string `eval` parses its expression as standard Perl. It is therefore expecting the decimal point to be a dot. If LC\_NUMERIC is set to have this be a comma instead, the parsing will be confused, perhaps silently.

```

use locale;
use POSIX qw(locale_h);
setlocale(LC_NUMERIC, "fr_FR") or die "Pardon";
my $a = 1.2;
print eval "$a + 1.5";
print "\n";

```

prints 13,5. This is because in that locale, the comma is the decimal point character. The `eval` thus expands to:

```
eval "1,2 + 1.5"
```

and the result is not what you likely expected. No warnings are generated. If you do string `eval`'s within the scope of `uselocale`, you should instead change the `eval` line to do something like:

```
print eval "no locale; $a + 1.5";
```

This prints 2.7.

You could also exclude `LC_NUMERIC`, if you don't need it, by

```
use locale '!:numeric';
```

### Backward compatibility

Versions of Perl prior to 5.004 **mostly** ignored locale information, generally behaving as if something similar to the "C" locale were always in force, even if the program environment suggested otherwise (see "The `setlocale` function"). By default, Perl still behaves this way for backward compatibility. If you want a Perl application to pay attention to locale information, you **must** use the `uselocale` pragma (see "The "use locale" pragma") or, in the unlikely event that you want to do so for just pattern matching, the `/l` regular expression modifier (see "Character set modifiers" in `perlre`) to instruct it to do so.

Versions of Perl from 5.002 to 5.003 did use the `LC_CTYPE` information if available; that is, `\w` did understand what were the letters according to the locale environment variables. The problem was that the user had no control over the feature: if the C library supported locales, Perl used them.

### I18N:Collate obsolete

In versions of Perl prior to 5.004, per-locale collation was possible using the `I18N::Collate` library module. This module is now mildly obsolete and should be avoided in new applications. The `LC_COLLATE` functionality is now integrated into the Perl core language: One can use locale-specific scalar data completely normally with `use locale`, so there is no longer any need to juggle with the scalar references of `I18N::Collate`.

### Sort speed and memory use impacts

Comparing and sorting by locale is usually slower than the default sorting; slow-downs of two to four times have been observed. It will also consume more memory: once a Perl scalar variable has participated in any string comparison or sorting operation obeying the locale collation rules, it will take 3-15 times more memory than before. (The exact multiplier depends on the string's contents, the operating system and the locale.) These downsides are dictated more by the operating system's implementation of the locale system than by Perl.

### Freely available locale definitions

The Unicode CLDR project extracts the POSIX portion of many of its locales, available at

<http://unicode.org/Public/cldr/2.0.1/>

(Newer versions of CLDR require you to compute the POSIX data yourself. See <http://unicode.org/Public/cldr/latest/>.)

There is a large collection of locale definitions at:

<http://std.dkuug.dk/i18n/WG15-collection/locales/>

You should be aware that it is unsupported, and is not claimed to be fit for any purpose. If your system allows installation of arbitrary locales, you may find the definitions useful as they are, or as a basis for the development of your own locales.

**I18n and I10n**

“Internationalization” is often abbreviated as **i18n** because its first and last letters are separated by eighteen others. (You may guess why the internalin ... internaliti ... i18n tends to get abbreviated.) In the same way, “localization” is often abbreviated to **I10n**.

**An imperfect standard**

Internationalization, as defined in the C and POSIX standards, can be criticized as incomplete, ungainly, and having too large a granularity. (Locales apply to a whole process, when it would arguably be more useful to have them apply to a single thread, window group, or whatever.) They also have a tendency, like standards groups, to divide the world into nations, when we all know that the world can equally well be divided into bankers, bikers, gamers, and so on.

**Unicode and UTF-8**

The support of Unicode is new starting from Perl version v5.6, and more fully implemented in versions v5.8 and later. See `perluniintro`.

Starting in Perl v5.20, UTF-8 locales are supported in Perl, except for `LC_COLLATE` (use `Unicode::Collate` instead). If you have Perl v5.16 or v5.18 and can't upgrade, you can use

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

When this form of the pragma is used, only the non-character portions of locales are used by Perl, for example `LC_NUMERIC`. Perl assumes that you have translated all the characters it is to operate on into Unicode (actually the platform's native character set (ASCII or EBCDIC) plus Unicode). For data in files, this can conveniently be done by also specifying

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

This pragma arranges for all inputs from files to be translated into Unicode from the current locale as specified in the environment (see “ENVIRONMENT”), and all outputs to files to be translated back into the locale. (See `open`). On a per-filehandle basis, you can instead use the `PerlIO::locale` module, or the `Encode::Locale` module, both available from CPAN. The latter module also has methods to ease the handling of `ARGV` and environment variables, and can be used on individual strings. If you know that all your locales will be UTF-8, as many are these days, you can use the `-C` command line switch.

This form of the pragma allows essentially seamless handling of locales with Unicode. The collation order will be by Unicode code point order. It is strongly recommended that when you need to order and sort strings that you use the standard module `Unicode::Collate` which gives much better results in many instances than you can get with the old-style locale handling.

All the modules and switches just described can be used in v5.20 with just plain `use locale`, and, should the input locales not be UTF-8, you'll get the less than ideal behavior, described below, that you get with pre-v5.16 Perls, or when you use the locale pragma without the `:not_characters` parameter in v5.16 and v5.18. If you are using exclusively UTF-8 locales in v5.20 and higher, the rest of this section does not apply to you.

There are two cases, multi-byte and single-byte locales. First multi-byte:

The only multi-byte (or wide character) locale that Perl is ever likely to support is UTF-8. This is due to the difficulty of implementation, the fact that high quality UTF-8 locales are now published for every area of the world (<http://unicode.org/Public/cldr/2.0.1/> for ones that are already set-up, but from an earlier version; <http://unicode.org/Public/cldr/latest/> for the most up-to-date, but you have to extract the POSIX information yourself), and that failing all that you can use the `Encode` module to translate to/from your locale. So, you'll have to do one of those things if you're using one of these locales, such as `Big5` or `Shift JIS`. For UTF-8 locales, in Perls (pre v5.20) that don't have full UTF-8 locale support, they may work reasonably well (depending on your C library implementation) simply because both they and Perl store characters that take up multiple bytes the same way. However, some, if not most, C library implementations may not process the characters in the upper half of the Latin-1 range (128 - 255) properly under `LC_CTYPE`. To see if a character is a particular type under a locale, Perl uses the functions like `isalnum()`. Your C library may not work for UTF-8 locales with those functions, instead only working under the newer wide library functions like `iswalnum()`, which Perl does not use. These multi-byte

locales are treated like single-byte locales, and will have the restrictions described below. Starting in Perl v5.22 a warning message is raised when Perl detects a multi-byte locale that it doesn't fully support.

For single-byte locales, Perl generally takes the tack to use locale rules on code points that can fit in a single byte, and Unicode rules for those that can't (though this isn't uniformly applied, see the note at the end of this section). This prevents many problems in locales that aren't UTF-8. Suppose the locale is ISO8859-7, Greek. The character at 0xD7 there is a capital Chi. But in the ISO8859-1 locale, Latin1, it is a multiplication sign. The POSIX regular expression character class `[[:alpha:]]` will magically match 0xD7 in the Greek locale but not in the Latin one.

However, there are places where this breaks down. Certain Perl constructs are for Unicode only, such as `\p{Alpha}`. They assume that 0xD7 always has its Unicode meaning (or the equivalent on EBCDIC platforms). Since Latin1 is a subset of Unicode and 0xD7 is the multiplication sign in both Latin1 and Unicode, `\p{Alpha}` will never match it, regardless of locale. A similar issue occurs with `\N{...}`. Prior to v5.20, it is therefore a bad idea to use `\p{}` or `\N{}` under plain `use locale--unless` you can guarantee that the locale will be ISO8859-1. Use POSIX character classes instead.

Another problem with this approach is that operations that cross the single byte/multiple byte boundary are not well-defined, and so are disallowed. (This boundary is between the codepoints at 255/256.) For example, lower casing LATIN CAPITAL LETTER Y WITH DIAERESIS (U+0178) should return LATIN SMALL LETTER Y WITH DIAERESIS (U+00FF). But in the Greek locale, for example, there is no character at 0xFF, and Perl has no way of knowing what the character at 0xFF is really supposed to represent. Thus it disallows the operation. In this mode, the lowercase of U+0178 is itself.

The same problems ensue if you enable automatic UTF-8-ification of your standard file handles, default `open()` layer, and `@ARGV` on non-ISO8859-1, non-UTF-8 locales (by using either the `-C` command line switch or the `PERL_UNICODE` environment variable; see `perlrun`). Things are read in as UTF-8, which would normally imply a Unicode interpretation, but the presence of a locale causes them to be interpreted in that locale instead. For example, a 0xD7 code point in the Unicode input, which should mean the multiplication sign, won't be interpreted by Perl that way under the Greek locale. This is not a problem *provided* you make certain that all locales will always and only be either an ISO8859-1, or, if you don't have a deficient C library, a UTF-8 locale.

Still another problem is that this approach can lead to two code points meaning the same character. Thus in a Greek locale, both U+03A7 and U+00D7 are GREEK CAPITAL LETTER CHI.

Because of all these problems, starting in v5.22, Perl will raise a warning if a multi-byte (hence Unicode) code point is used when a single-byte locale is in effect. (Although it doesn't check for this if doing so would unreasonably slow execution down.)

Vendor locales are notoriously buggy, and it is difficult for Perl to test its locale-handling code because this interacts with code that Perl has no control over; therefore the locale-handling code in Perl may be buggy as well. (However, the Unicode-supplied locales should be better, and there is a feed back mechanism to correct any problems. See "Freely available locale definitions".)

If you have Perl v5.16, the problems mentioned above go away if you use the `:not_characters` parameter to the locale pragma (except for vendor bugs in the non-character portions). If you don't have v5.16, and you *do* have locales that work, using them may be worthwhile for certain specific purposes, as long as you keep in mind the gotchas already mentioned. For example, if the collation for your locales works, it runs faster under locales than under `Unicode::Collate`; and you gain access to such things as the local currency symbol and the names of the months and days of the week. (But to hammer home the point, in v5.16, you get this access without the downsides of locales by using the `:not_characters` form of the pragma.)

Note: The policy of using locale rules for code points that can fit in a byte, and Unicode rules for those that can't is not uniformly applied. Pre-v5.12, it was somewhat haphazard; in v5.12 it was applied fairly consistently to regular expression matching except for bracketed character classes; in v5.14 it was extended to all regex matches; and in v5.16 to the casing operations such as `\L` and `uc()`. For collation, in all releases so far, the system's `strxfrm()` function is called, and whatever it does is what you get.

## BUGS

### Broken systems

In certain systems, the operating system's locale support is broken and cannot be fixed or used by Perl. Such deficiencies can and will result in mysterious hangs and/or Perl core dumps when `use locale` is in effect. When confronted with such a system, please report in excruciating detail to [<perlbug@perl.org>](mailto:perlbug@perl.org), and also contact your vendor: bug fixes may exist for these problems in your operating system. Sometimes such bug fixes are called an operating system upgrade. If you have the source for Perl, include in the [perlbug\(1\)](#) email the output of the test described above in "Testing for broken locales".

### SEE ALSO

[I18N::Langinfo](#), [perluniintro\(1\)](#), [perlunicode\(1\)](#), `open`, "isalnum" in POSIX, "isalpha" in POSIX, "isdigit" in POSIX, "isgraph" in POSIX, "islower" in POSIX, "isprint" in POSIX, "ispunct" in POSIX, "isspace" in POSIX, "isupper" in POSIX, "isxdigit" in POSIX, "localeconv" in POSIX, "setlocale" in POSIX, "strcoll" in POSIX, "strftime" in POSIX, "strtod" in POSIX, "strxfrm" in POSIX.

For special considerations when Perl is embedded in a C program, see "Using embedded Perl with POSIX locales" in `perlembed`.

### HISTORY

Jarkko Hietaniemi's original *perl18n.pod* heavily hacked by Dominic Dunlop, assisted by the perl5-porters. Prose worked over a bit by Tom Christiansen, and updated by Perl 5 porters.