NAME

perllocale - Perl locale handling (internationalization and localization)

DESCRIPTION

Perl supports language-specific notions of data such as "is this a letter", "what is the uppercase equivalent of this letter", and "which of these letters comes first". These are important issues, especially for languages other than English--but also for English: it would be naïve to imagine that `A-Za-z` defines all the "letters" needed to write in English. Perl is also aware that some character other than '.' may be preferred as a decimal point, and that output date representations may be language-specific. The process of making an application take account of its users' preferences in such matters is called **internationalization** (often abbreviated as **i18n**); telling such an application about a particular set of preferences is known as **localization** (**l10n**).

Perl can understand language-specific data via the standardized (ISO C, XPG4, POSIX 1.c) method called "the locale system". The locale system is controlled per application using one pragma, one function call, and several environment variables.

NOTE: This feature is new in Perl 5.004, and does not apply unless an application specifically requests it--see *Backward compatibility*. The one exception is that write() now **always** uses the current locale - see *NOTES*.

PREPARING TO USE LOCALES

If Perl applications are to understand and present your data correctly according a locale of your choice, **all** of the following must be true:

- **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 `use locale` pragma (see *The use locale pragma*) where appropriate, and **at least one** of the following must be true:

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

USING LOCALES

The use locale pragma

By default, Perl ignores the current locale. The `use locale` pragma tells Perl to use the current locale for some operations:

- **The comparison operators** (`lt`, `le`, `cmp`, `ge`, and `gt`) and the POSIX string collation functions strcoll() and strxfrm() 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 formatting functions** (printf(), sprintf() and write()) use LC_NUMERIC
- **The POSIX date formatting function** (strftime()) uses LC_TIME.

LC_COLLATE, LC_CTYPE, and so on, are discussed further in *LOCALE CATEGORIES*.

The default behavior is restored with the no locale pragma, or upon reaching the end of block enclosing use locale.

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

The setlocale function

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

```
# This functionality not usable prior to Perl 5.004
require 5.004;

# Import locale-handling tool set from POSIX module.
# This example uses: setlocale -- the function call
#                    LC_CTYPE -- explained below
use POSIX qw(locale_h);

# 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 default defined by LC_ALL/LC_CTYPE/LANG
# environment variables.  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 else 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().

If no second argument is provided and the category is LC_ALL, the result is implementation-dependent. It may be a string of concatenated locales 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.

If the second argument does not correspond to a valid locale, 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.

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. 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.

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 a zero value, for example "0". 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 in 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 `LC_NUMERIC` and `LC_MONETARY` locales. (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:

```
        # See comments in previous example
        require 5.004;
        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";
```

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 `use locale`, 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 'å' belong? And while 'color' follows 'chocolate' in English, what about in 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 `use locale` 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 `use locale` 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.

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: `use locale` isn't shown in some of these examples because it isn't needed: strcoll() and strxfrm() exist only to generate locale-dependent results, and so always obey the current `LC_COLLATE` locale.

Category LC_CTYPE: Character Types

In the scope of `use locale`, Perl obeys the `LC_CTYPE` locale setting. This controls the application's notion of which characters are alphabetic. This affects Perl's `\w` regular expression metanotation, which stands for alphanumeric characters--that is, alphabetic, numeric, and including other special characters such as the underscore or hyphen. (Consult *perlre* 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.

The `LC_CTYPE` locale also provides the map used in transliterating characters between lower and uppercase. This affects the case-mapping functions--lc(), lcfirst, uc(), and ucfirst(); case-mapping interpolation with `\l`, `\L`, `\u`, or `\U` in double-quoted strings and `s///` substitutions; and case-independent regular expression pattern matching using the `i` modifier.

Finally, `LC_CTYPE` affects the POSIX character-class test functions--isalpha(), islower(), and so on. For example, if you move from the "C" locale to a 7-bit Scandinavian one, you may find--possibly to your surprise--that "|" moves from the ispunct() class to isalpha().

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) letters and digits--for example, in command strings--locale-aware applications should use `\w` inside a `no locale` block. See *SECURITY*.

Category LC_NUMERIC: Numeric Formatting

After a proper POSIX::setlocale() call, Perl obeys the `LC_NUMERIC` locale information, which controls an application's idea of how numbers should be formatted for human readability by the printf(), sprintf(), and write() functions. String-to-numeric conversion by the POSIX::strtod() function is also affected. In most implementations the only effect is to change the character used for the decimal point--perhaps from '.' to ','. These functions aren't aware of such niceties as thousands separation and so on. (See *The localeconv function* if you care about these things.)

Output produced by print() is also affected by the current locale: it corresponds to what you'd get from printf() in the "C" locale. The same is true for Perl's internal conversions between numeric and string formats:

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

 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 no 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 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: as a function that exists only to generate locale-dependent results, strftime() 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 category, LC_MESSAGES (possibly supplemented by others in particular implementations) is not currently used by Perl--except possibly to affect the behavior of library functions called by 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*, 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 or \U)

  Result string containing interpolated material is tainted if use locale is in effect.

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

  Scalar true/false result never tainted.

  Subpatterns, either delivered as a list-context result or as $1 etc. are tainted if use locale is in effect, and the subpattern regular expression contains \w (to match an alphanumeric character), \W (non-alphanumeric character), \s (whitespace character), or \S (non whitespace character). The matched-pattern variable, $&, $` (pre-match), $' (post-match), and $+ (last match) are also tainted if use locale is in effect and the regular expression contains \w, \W, \s, or \S.

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

  Has the same behavior as the match operator. Also, the left operand of =~ becomes tainted when use locale in effect if modified as a result of a substitution based on a regular expression match involving \w, \W, \s, or \S; or of case-mapping with \l, \L,\u or \U.

- **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 `use locale` is in effect.

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

  Results are never tainted.

- **POSIX character class tests** (isalnum(), isalpha(), isdigit(), isgraph(), islower(), isprint(), ispunct(), isspace(), isupper(), isxdigit()):

  True/false 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 $untainted_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/]+%;
$untainted_output_file = $&;

open(F, ">$untainted_output_file")
    or warn "Open of $untainted_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_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 that does not evaluate to integer zero--that is, "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.

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.

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_....

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

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 use locale pragma (see *The use locale pragma*) 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.

write() and LC_NUMERIC

Formats are the only part of Perl that unconditionally use information from a program's locale; if a program's environment specifies an LC_NUMERIC locale, it is always used to specify the decimal point character in formatted output. Formatted output cannot be controlled by use locale because the pragma is tied to the block structure of the program, and, for historical reasons, formats exist outside that block structure.

Freely available locale definitions

There is a large collection of locale definitions at ftp://dkuug.dk/i18n/WG15-collection . 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 l10n

"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 **l10n**.

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. But, for now, it's the only standard we've got. This may be construed as a bug.

Unicode and UTF-8

The support of Unicode is new starting from Perl version 5.6, and more fully implemented in the version 5.8. See *perluniintro* and *perlunicode* for more details.

Usually locale settings and Unicode do not affect each other, but there are exceptions, see *"Locales" in perlunicode* for examples.

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 the `use locale` is in effect. When confronted with such a system, please report in excruciating detail to <*perlbug@perl.org*>, and complain to your vendor: bug fixes may exist for these problems in your operating system. Sometimes such bug fixes are called an operating system upgrade.

SEE ALSO

I18N::Langinfo, *perluniintro*, *perlunicode*, *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*.

HISTORY

Jarkko Hietaniemi's original *perli18n.pod* heavily hacked by Dominic Dunlop, assisted by the perl5-porters. Prose worked over a bit by Tom Christiansen.

Last update: Thu Jun 11 08:44:13 MDT 1998