NAME

perldocstyle - A style guide for writing Perl’s documentation

DESCRIPTION

This document is a guide for the authorship and maintenance of the documentation that ships with Perl. This includes the following:

- The several dozen manual sections whose filenames begin with ""perl"", such as "perlobj", "perlre", and "perlintro". (And, yes, "perl".)

- The documentation for all the modules included with Perl (as listed by "perlmodlib").

- The hundreds of individually presented reference sections derived from the "perlfunc" file.

This guide will hereafter refer to user-manual section files as *man pages*, per Unix convention.

Purpose of this guide

This style guide aims to establish standards, procedures, and philosophies applicable to Perl’s core documentation.

Adherence to these standards will help ensure that any one part of Perl’s manual has a tone and style consistent with that of any other. As with the rest of the Perl project, the language’s documentation collection is an open-source project authored over a long period of time by many people. Maintaining consistency across such a wide swath of work presents a challenge; this guide provides a foundation to help mitigate this difficulty.

This will help its readers--especially those new to Perl--to feel more welcome and engaged with Perl’s documentation, and this in turn will help the Perl project itself grow stronger through having a larger, more diverse, and more confident population of knowledgeable users.

Intended audience

Anyone interested in contributing to Perl’s core documentation should familiarize themselves with the standards outlined by this guide.

Programmers documenting their own work apart from the Perl project itself may also find this guide worthwhile, especially if they wish their work to extend the tone and style of Perl’s own manual.

Status of this document

This guide was initially drafted in late 2020, drawing from the documentation style guides of several open-source technologies contemporary with Perl. This has included Python, Raku, Rust, and the Linux

kernel.

The author intends to see this guide used as starting place from which to launch a review of Perl’s reams of extant documentation, with the expectation that those conducting this review should grow and modify this guide as needed to account for the requirements and quirks particular to Perl’s programming manual.

FUNDAMENTALS

Choice of markup: Pod

All of Perl’s core documentation uses Pod ("Plain Old Documentation"), a simple markup language, to format its source text. Pod is similar in spirit to other contemporary lightweight markup technologies, such as Markdown and reStructuredText, and has a decades-long shared history with Perl itself.

For a comprehensive reference to Pod syntax, see "perlpod". For the sake of reading this guide, familiarity with the Pod syntax for section headers ("=head2", et cetera) and for inline text formatting ("C<like this>") should suffice.

Perl programmers also use Pod to document their own scripts, libraries, and modules. This use of Pod has its own style guide, outlined by "perlpodstyle".

Choice of language: American English

Perl’s core documentation is written in English, with a preference for American spelling of words and expression of phrases. That means "color" over "colour", "math" versus "maths", "the team has decided" and not "the team have decided", and so on.

We name one style of English for the sake of consistency across Perl’s documentation, much as a software project might declare a four-space indentation standard--even when that doesn’t affect how well the code compiles. Both efforts result in an easier read by avoiding jarring, mid-document changes in format or style.

Contributors to Perl’s documentation should note that this rule describes the ultimate, published output of the project, and does not prescribe the dialect used within community contributions. The documentation team enthusiastically welcomes any English-language contributions, and will actively assist in Americanizing spelling and style when warranted.

Other languages and translations

Community-authored translations of Perl’s documentation do exist, covering a variety of languages. While the Perl project appreciates these translation efforts and promotes them when applicable, it does not officially support or maintain any of them.

That said, keeping Perl's documentation clear, simple, and short has a welcome side effect of aiding any such translation project.

(Note that the Chinese, Japanese, and Korean-language README files included with Perl's source distributions provide an exception to this choice of language--but these documents fall outside the scope of this guide.)

Choice of encoding: UTF-8

Perl's core documentation files are encoded in UTF-8, and can make use of the full range of characters this encoding allows.

As such, every core doc file (or the Pod section of every core module) should commence with an "=encoding utf8" declaration.

Choice of underlying style guide: CMOS

Perl's documentation uses the Chicago Manual of Style <https://www.chicagomanualofstyle.org> (CMOS), 17th Edition, as its baseline guide for style and grammar. While the document you are currently reading endeavors to serve as an adequate stand-alone style guide for the purposes of documenting Perl, authors should consider CMOS the fallback authority for any pertinent topics not covered here.

Because CMOS is not a free resource, access to it is not a prerequisite for contributing to Perl's documentation; the doc team will help contributors learn about and apply its guidelines as needed. However, we do encourage anyone interested in significant doc contributions to obtain or at least read through CMOS. (Copies are likely available through most public libraries, and CMOS-derived fundamentals can be found online as well.)

Contributing to Perl's documentation

Perl, like any programming language, is only as good as its documentation. Perl depends upon clear, friendly, and thorough documentation in order to welcome brand-new users, teach and explain the language's various concepts and components, and serve as a lifelong reference for experienced Perl programmers. As such, the Perl project welcomes and values all community efforts to improve the language's documentation.

Perl accepts documentation contributions through the same open-source project pipeline as code contributions. See "perlhack" for more information.

FORMATTING AND STRUCTURE

This section details specific Pod syntax and style that all core Perl documentation should adhere to, in the interest of consistency and readability.

Document structure

Each individual work of core Perl documentation, whether contained within a ".pod" file or in the Pod section of a standard code module, patterns its structure after a number of long-time Unix man page conventions. (Hence this guide's use of "man page" to refer to any one self-contained part of Perl's documentation.)

Adhering to these conventions helps Pod formatters present a Perl man page's content in different contexts--whether a terminal, the web, or even print. Many of the following requirements originate with "perlpodstyle", which derives its recommendations in turn from these well-established practices.

Name

After its "=encoding utf8" declaration, a Perl man page *must* present a level-one header named "NAME" (literally), followed by a paragraph containing the page's name and a very brief description.

The first few lines of a notional page named "perlpodexample":

```
=encoding utf8

=head1 NAME

perlpodexample - An example of formatting a manual page's title line
```

Description and synopsis

Most Perl man pages also contain a DESCRIPTION section featuring a summary of, or introduction to, the document's content and purpose.

This section should also, one way or another, clearly identify the audience that the page addresses, especially if it has expectations about the reader's prior knowledge. For example, a man page that dives deep into the inner workings of Perl's regular expression engine should state its assumptions up front--and quickly redirect readers who are instead looking for a more basic reference or tutorial.

Reference pages, when appropriate, can precede the DESCRIPTION with a SYNOPSIS section that lists, within one or more code blocks, some very brief examples of the referenced feature's use. This section should show a handful of common-case and best-practice examples, rather than an exhaustive list of every obscure method or alternate syntax available.

Other sections and subsections

Pages should conclude, when appropriate, with a SEE ALSO section containing hyperlinks to relevant sections of Perl's manual, other Unix man pages, or appropriate web pages. Hyperlink each such cross-reference via "L<...>".

What other sections to include depends entirely upon the topic at hand. Authors should feel free to include further "=head1"-level sections, whether other standard ones listed by "perlpodstyle", or ones specific to the page's topic; in either case, render these top-level headings in all-capital letters.

You may then include as many subsections beneath them as needed to meet the standards of clarity, accessibility, and cross-reference affinity suggested elsewhere in this guide.

Author and copyright

In most circumstances, Perl's stand-alone man pages--those contained within ".pod" files--do not need to include any copyright or license information about themselves. Their source Pod files are part of Perl's own core software repository, and that already covers them under the same copyright and license terms as Perl itself. You do not need to include additional "LICENSE" or "COPYRIGHT" sections of your own.

These man pages may optionally credit their primary author, or include a list of significant contributors, under "AUTHOR" or "CONTRIBUTORS" headings. Note that the presence of authors' names does not preclude a given page from writing in a voice consistent with the rest of Perl's documentation.

Note that these guidelines do not apply to the core software modules that ship with Perl. These have their own standards for authorship and copyright statements, as found in "perlpodstyle".

Formatting rules

Line length and line wrap

Each line within a Perl man page's Pod source file should measure 72 characters or fewer in length.

Please break paragraphs up into blocks of short lines, rather than "soft wrapping" paragraphs across hundreds of characters with no line breaks.

Code blocks

Just like the text around them, all code examples should be as short and readable as possible, displaying no more complexity than absolutely necessary to illustrate the concept at hand.

For the sake of consistency within and across Perl's man pages, all examples must adhere to the code-layout principles set out by "perlstyle".

Sample code should deviate from these standards only when necessary: during a demonstration of how Perl disregards whitespace, for example, or to temporarily switch to two-column indentation for an unavoidably verbose illustration.

You may include comments within example code to further clarify or label the code's behavior in-line. You may also use comments as placeholder for code normally present but not relevant to the current topic, like so:

```
while (my $line = <$fh>) {
    #
    # (Do something interesting with $line here.)
    #
}
```

Even the simplest code blocks often require the use of example variables and subroutines, whose names you should choose with care.

Inline code and literals

Within a paragraph of text, use "C<...>" when quoting or referring to any bit of Perl code--even if it is only one character long.

For instance, when referring within an explanatory paragraph to Perl's operator for adding two numbers together, you'd write ""C<+>"".

Function names

Use "C<...>" to render all Perl function names in monospace, whenever they appear in text.

Unless you need to specifically quote a function call with a list of arguments, do not follow a function's name in text with a pair of empty parentheses. That is, when referring in general to Perl's "print" function, write it as ""print"", not ""print()"".

Function arguments

Represent functions' expected arguments in all-caps, with no sigils, and using "C<...>" to render them in monospace. These arguments should have short names making their nature and purpose clear.

Convention specifies a few ones commonly seen throughout Perl’s documentation:

- EXPR

  The "generic" argument: any scalar value, or a Perl expression that evaluates to one.

- ARRAY

  An array, stored in a named variable.

- HASH

  A hash, stored in a named variable.

- BLOCK

  A curly-braced code block, or a subroutine reference.

- LIST

  Any number of values, stored across any number of variables or expressions, which the function will "flatten" and treat as a single list. (And because it can contain any number of variables, it must be the *last* argument, when present.)

When possible, give scalar arguments names that suggest their purpose among the arguments. See, for example, "substr"’s documentation, whose listed arguments include "EXPR", "OFFSET", "LENGTH", and "REPLACEMENT".

Apostrophes, quotes, and dashes

In Pod source, use straight quotes, and not "curly quotes": "Like this", not Xlike thisX. The same goes for apostrophes: Here’s a positive example, and hereXs a negative one.

Render em dashes as two hyphens--like this:

```
Render em dashes as two hyphens--like this.
```

Leave it up to formatters to reformat and reshape these punctuation marks as best fits their respective target media.

Unix programs and C functions

When referring to a Unix program or C function with its own man page (outside of Perl's documentation), include its manual section number in parentheses. For example: malloc(3), or mkdir(1).

If mentioning this program for the first time within a man page or section, make it a cross reference, e.g. "L<malloc(3)>".

Do not otherwise style this text.

Cross-references and hyperlinks

Make generous use of Pod's "L<...>" syntax to create hyperlinks to other parts of the current man page, or to other documents entirely -- whether elsewhere on the reader's computer, or somewhere on the internet, via URL.

Use "L<...>" to link to another section of the current man page when mentioning it, and make use of its page-and-section syntax to link to the most specific section of a separate page within Perl's documentation. Generally, the first time you refer to a specific function, program, or concept within a certain page or section, consider linking to its full documentation.

Hyperlinks do not supersede other formatting required by this guide; Pod allows nested text formats, and you should use this feature as needed.

Here is an example sentence that mentions Perl's "say" function, with a link to its documentation section within the "perlfunc" man page:

```
In version 5.10, Perl added support for the
L<C<say>|perlfunc/say FILEHANDLE LIST> function.
```

Note the use of the vertical pipe (""|"") to separate how the link will appear to readers (""C<say>"") from the full page-and-section specifier that the formatter links to.

Tables and diagrams

Pod does not officially support tables. To best present tabular data, include the table as both HTML and plain-text representations--the latter as an indented code block. Use "=begin" / "=end" directives to target these tables at "html" and "text" Pod formatters, respectively. For example:

```
=head2 Table of fruits

=begin text

 Name          Shape          Color
 ===================================
 Apple         Round          Red
 Banana         Long          Yellow
 Pear          Pear-shaped     Green

=end text

=begin html

<table>
<tr><th>Name</th><th>Shape</th><th>Color</th></tr>
<tr><td>Apple</td><td>Round</td><td>Red</td></tr>
<tr><td>Banana</td><td>Long</td><td>Yellow</td></tr>
<tr><td>Pear</td><td>Pear-shaped</td><td>Green</td></tr>
</table>

=end html
```

The same holds true for figures and graphical illustrations. Pod does not natively support inline graphics, but you can mix HTML "<img>" tags with monospaced text-art representations of those images' content.

Due in part to these limitations, most Perl man pages use neither tables nor diagrams. Like any other tool in your documentation toolkit, however, you may consider their inclusion when they would improve an explanation's clarity without adding to its complexity.

Adding comments

Like any other kind of source code, Pod lets you insert comments visible only to other people reading the source directly, and ignored by the formatting programs that transform Pod into various human-friendly output formats (such as HTML or PDF).

To comment Pod text, use the "=for" and "=begin" / "=end" Pod directives, aiming them at a (notional) formatter called ""comment"". A couple of examples:

```
=for comment Using "=for comment" like this is good for short,
```

single-paragraph comments.

=begin comment

If you need to comment out more than one paragraph, use a
=begin/=end block, like this.

None of the text or markup in this whole example would be visible to
someone reading the documentation through normal means, so it's
great for leaving notes, explanations, or suggestions for your
fellow documentation writers.

=end comment

In the tradition of any good open-source project, you should make free but judicious use of comments to leave in-line "meta-documentation" as needed for other Perl documentation writers (including your future self).

Perlfunc has special rules

The "perlfunc" man page, an exhaustive reference of every Perl built-in function, has a handful of formatting rules not seen elsewhere in Perl's documentation.

Software used during Perl's build process (Pod::Functions) parses this page according to certain rules, in order to build separate man pages for each of Perl's functions, as well as achieve other indexing effects. As such, contributors to perlfunc must know about and adhere to its particular rules.

Most of the perfunc man page comprises a single list, found under the header "Alphabetical Listing of Perl Functions". Each function reference is an entry on that list, made of three parts, in order:

1. A list of "=item" lines which each demonstrate, in template format, a way to call this function. One line should exist for every combination of arguments that the function accepts (including no arguments at all, if applicable).

   If modern best practices prefer certain ways to invoke the function over others, then those ways should lead the list.

   The first item of the list should be immediately followed by one or more "X<...>" terms listing index-worthy topics; if nothing else, then the name of the function, with no arguments.

2. A "=for" line, directed at "Pod::Functions", containing a one-line description of what the function

does. This is written as a phrase, led with an imperative verb, with neither leading capitalization nor ending punctuation. Examples include "quote a list of words" and "change a filename".

3. The function’s definition and reference material, including all explanatory text and code examples.

Complex functions that need their text divided into subsections (under the principles of "Apply section-breaks and examples generously") may do so by using sublists, with "=item" elements as header text.

A fictional function ""myfunc"", which takes a list as an optional argument, might have an entry in perlfunc shaped like this:

```
=item myfunc LIST
X<myfunc>

=item myfunc

=for Pod::Functions demonstrate a function’s perlfunc section

[ Main part of function definition goes here, with examples ]

=over

=item Legacy uses

[ Examples of deprecated syntax still worth documenting ]

=item Security considerations

[ And so on... ]

=back
```

TONE AND STYLE

Apply one of the four documentation modes

Aside from "meta" documentation such as "perlhist" or "perlartistic", each of Perl’s man pages should conform to one of the four documentation "modes" suggested by *The Documentation System* by Daniele Procida <https://documentation.divio.com>. These include tutorials, cookbooks, explainers, and references--terms that we define in further detail below.

Each mode of documentation speaks to a different audience--not just people of different backgrounds and skill levels, but individual readers whose needs from language documentation can shift depending upon context. For example, a programmer with plenty of time to learn a new concept about Perl can ease into a tutorial about it, and later expand their knowledge further by studying an explainer. Later, that same programmer, wading knee-deep in live code and needing only to look up some function's exact syntax, will want to reach for a reference page instead.

Perl's documentation must strive to meet these different situational expectations by limiting each man page to a single mode. This helps writers ensure they provide readers with the documentation needed or expected, despite ever-evolving situations.

Tutorial

A tutorial man page focuses on **learning**, ideally by *doing*. It presents the reader with small, interesting examples that allow them to follow along themselves using their own Perl interpreter. The tutorial inspires comprehension by letting its readers immediately experience (and experiment on) the concept in question. Examples include "perlxstut", "perlpacktut", and "perlretut".

Tutorial man pages must strive for a welcoming and reassuring tone from their outset; they may very well be the first things that a newcomer to Perl reads, playing a significant role in whether they choose to stick around. Even an experienced programmer can benefit from the sense of courage imparted by a strong tutorial about a more advanced topic. After completing a tutorial, a reader should feel like they've been led from zero knowledge of its topic to having an invigorating spark of basic understanding, excited to learn more and experiment further.

Tutorials can certainly use real-world examples when that helps make for clear, relatable demonstrations, so long as they keep the focus on teaching--more practical problem-solving should be left to the realm of cookbooks (as described below). Tutorials also needn't concern themselves with explanations into why or how things work beneath the surface, or explorations of alternate syntaxes and solutions; these are better handled by explainers and reference pages.

Cookbook

A cookbook man page focuses on **results**. Just like its name suggests, it presents succinct, step-by-step solutions to a variety of real-world problems around some topic. A cookbook's code examples serve less to enlighten and more to provide quick, paste-ready solutions that the reader can apply immediately to the situation facing them.

A Perl cookbook demonstrates ways that all the tools and techniques explained elsewhere can work together in order to achieve practical results. Any explanation deeper than that belongs in explainers

and reference pages, instead. (Certainly, a cookbook can cross-reference other man pages in order to satisfy the curiosity of readers who, with their immediate problems solved, wish to learn more.)

The most prominent cookbook pages that ship with Perl itself are its many FAQ pages, in particular "perlfaq4" and up, which provide short solutions to practical questions in question-and-answer style. "perlunicook" shows another example, containing a bevy of practical code snippets for a variety of internationally minded text manipulations.

(An aside: *The Documentation System* calls this mode "how-to", but Perl's history of creative cuisine prefers the more kitchen-ready term that we employ here.)

Reference

A reference page focuses on **description**. Austere, uniform, and succinct, reference pages--often arranged into a whole section of mutually similar subpages--lend themselves well to "random access" by a reader who knows precisely what knowledge they need, requiring only the minimum amount of information before returning to the task at hand.

Perl's own best example of a reference work is "perlfunc", the sprawling man page that details the operation of every function built into Perl, with each function's documentation presenting the same kinds of information in the same order as every other. For an example of a shorter reference on a single topic, look at "perlreref".

Module documentation--including that of all the modules listed in "perlmodlib"--also counts as reference. They follow precepts similar to those laid down by the "perlpodstyle" man page, such as opening with an example-laden "SYNOPSIS" section, or featuring a "METHODS" section that succinctly lists and defines an object-oriented module's public interface.

Explainer

Explainer pages focus on **discussion**. Each explainer dives as deep as needed into some Perl-relevant topic, taking all the time and space needed to give the reader a thorough understanding of it. Explainers mean to impart knowledge through study. They don't assume that the student has a Perl interpreter fired up and hungry for immediate examples (as with a tutorial), or specific Perl problems that they need quick answers for (which cookbooks and reference pages can help with).

Outside of its reference pages, most of Perl's manual belongs to this mode. This includes the majority of the man pages whose names start with ""perl"". A fine example is "perlsyn", the Perl Syntax page, which explores the whys and wherefores of Perl's unique syntax in a wide-ranging discussion laden with many references to the language's history, culture, and driving philosophies.

Perl’s explainer pages give authors a chance to explore Perl’s penchant for TMTOWTDI, illustrating alternate and even obscure ways to use the language feature under discussion. However, as the remainder of this guide discusses, the ideal Perl documentation manages to deliver its message clearly and concisely, and not confuse mere wordiness for completeness.

Further notes on documentation modes

Keep in mind that the purpose of this categorization is not to dictate content--a very thorough explainer might contain short reference sections of its own, for example, or a reference page about a very complex function might resemble an explainer in places (e.g. "open"). Rather, it makes sure that the authors and contributors of any given man page agree on what sort of audience that page addresses.

If a new or otherwise uncategorized man page presents itself as resistant to fitting into only one of the four modes, consider breaking it up into separate pages. That may mean creating a new ""perl[...]"" man page, or (in the case of module documentation) making new packages underneath that module’s namespace that serve only to hold additional documentation. For instance, "Example::Module"’s reference documentation might include a see-also link to "Example::Module::Cookbook".

Perl’s several man pages about Unicode--comprising a short tutorial, a thorough explainer, a cookbook, and a FAQ--provide a fine example of spreading a complicated topic across several man pages with different and clearly indicated purposes.

Assume readers’ intelligence, but not their knowledge

Perl has grown a great deal from its humble beginnings as a tool for people already well versed in C programming and various Unix utilities. Today, a person learning Perl might come from any social or technological background, with a range of possible motivations stretching far beyond system administration.

Perl’s core documentation must recognize this by making as few assumptions as possible about the reader’s prior knowledge. While you should assume that readers of Perl’s documentation are smart, curious, and eager to learn, you should not confuse this for pre-existing knowledge about any other technology, or even programming in general--especially in tutorial or introductory material.

Keep Perl’s documentation about Perl

Outside of pages tasked specifically with exploring Perl’s relationship with other programming languages, the documentation should keep the focus on Perl. Avoid drawing analogies to other technologies that the reader may not have familiarity with.

For example, when documenting one of Perl’s built-in functions, write as if the reader is now learning

about that function for the first time, in any programming language.

Choosing to instead compare it to an equivalent or underlying C function will probably not illuminate much understanding in a contemporary reader. Worse, this can risk leaving readers unfamiliar with C feeling locked out from fully understanding of the topic--to say nothing of readers new to computer programming altogether.

If, however, that function's ties to its C roots can lead to deeper understanding with practical applications for a Perl programmer, you may mention that link after its more immediately useful documentation. Otherwise, omit this information entirely, leaving it for other documentation or external articles more concerned with examining Perl's underlying implementation details.

Deploy jargon when needed, but define it as well

Domain-specific jargon has its place, especially within documentation. However, if a man page makes use of jargon that a typical reader might not already know, then that page should make an effort to define the term in question early-on--either explicitly, or via cross reference.

For example, Perl loves working with filehandles, and as such that word appears throughout its documentation. A new Perl programmer arriving at a man page for the first time is quite likely to have no idea what a "filehandle" is, though. Any Perl man page mentioning filehandles should, at the very least, hyperlink that term to an explanation elsewhere in Perl's documentation. If appropriate--for example, in the lead-in to "open" function's detailed reference--it can also include a very short in-place definition of the concept for the reader's convenience.

Use meaningful variable and symbol names in examples

When quickly sketching out examples, English-speaking programmers have a long tradition of using short nonsense words as placeholders for variables and other symbols--such as the venerable "foo", "bar", and "baz". Example code found in a programming language's official, permanent documentation, however, can and should make an effort to provide a little more clarity through specificity.

Whenever possible, code examples should give variables, classes, and other programmer-defined symbols names that clearly demonstrate their function and their relationship to one another. For example, if an example requires that one class show an "is-a" relationship with another, consider naming them something like "Apple" and "Fruit", rather than "Foo" and "Bar". Similarly, sample code creating an instance of that class would do better to name it $apple, rather than $baz.

Even the simplest examples benefit from clear language using concrete words. Prefer a construct like "for my $item (@items) { ... }" over "for my $blah (@blah) { ... }".

Write in English, but not just for English-speakers

While this style guide does specify American English as the documentation's language for the sake of internal consistency, authors should avoid cultural or idiomatic references available only to English-speaking Americans (or any other specific culture or society). As much as possible, the language employed by Perl's core documentation should strive towards cultural universality, if not neutrality. Regional turns of phrase, examples drawing on popular-culture knowledge, and other rhetorical techniques of that nature should appear sparingly, if at all.

Authors should feel free to let more freewheeling language flourish in "second-order" documentation about Perl, like books, blog entries, and magazine articles, published elsewhere and with a narrower readership in mind. But Perl's own docs should use language as accessible and welcoming to as wide an audience as possible.

Omit placeholder text or commentary

Placeholder text does not belong in the documentation that ships with Perl. No section header should be followed by text reading only "Watch this space", "To be included later", or the like. While Perl's source files may shift and alter as much as any other actively maintained technology, each released iteration of its technology should feel complete and self-contained, with no such future promises or other loose ends visible.

Take advantage of Perl's regular release cycle. Instead of cluttering the docs with flags promising more information later--the presence of which do not help readers at all today--the documentation's maintenance team should treat any known documentation absences as an issue to address like any other in the Perl project. Let Perl's contributors, testers, and release engineers address that need, and resist the temptation to insert apologies, which have all the utility in documentation as undeleted debug messages do in production code.

Apply section-breaks and examples generously

No matter how accessible their tone, the sight of monolithic blocks of text in technical documentation can present a will-weakening challenge for the reader. Authors can improve this situation through breaking long passages up into subsections with short, meaningful headers.

Since every section-header in Pod also acts as a potential end-point for a cross-reference (made via Pod's "L<...>" syntax), putting plenty of subsections in your documentation lets other man pages more precisely link to a particular topic. This creates hyperlinks directly to the most appropriate section rather than to the whole page in general, and helps create a more cohesive sense of a rich, consistent, and interrelated manual for readers.

Among the four documentation modes, sections belong more naturally in tutorials and explainers. The step-by-step instructions of cookbooks, or the austere definitions of reference pages, usually have no

room for them. But authors can always make exceptions for unusually complex concepts that require further breakdown for clarity's sake.

Example code, on the other hand, can be a welcome addition to any mode of documentation. Code blocks help break up a man page visually, reassuring the reader that no matter how deep the textual explanation gets, they are never far from another practical example showing how it all comes together using a small, easy-to-read snippet of tested Perl code.

Lead with common cases and best practices

Perl famously gives programmers more than one way to do things. Like any other long-lived programming language, Perl has also built up a large, community-held notion of best practices, blessing some ways to do things as better than others, usually for the sake of more maintainable code.

Show the better ways first

Whenever it needs to show the rules for a technique which Perl provides many avenues for, the documentation should always lead with best practices. And when discussing some part of the Perl toolkit with many applications, the docs should begin with a demonstration of its application to the most common cases.

The "open" function, for example, has myriad potential uses within Perl programs, but *most of the time* programmers--and especially those new to Perl--turn to this reference because they simply wish to open a file for reading or writing. For this reason, "open"'s documentation begins there, and only descends into the function's more obscure uses after thoroughly documenting and demonstrating how it works in the common case. Furthermore, while engaging in this demonstration, the "open" documentation does not burden the reader right away with detailed explanations about calling "open" via any route other than the best-practice, three-argument style.

Show the lesser ways when needed

Sometimes, thoroughness demands documentation of deprecated techniques. For example, a certain Perl function might have an alternate syntax now considered outmoded and no longer best-practice, but which a maintainer of a legacy project might quite reasonably encounter when exploring old code. In this case, these features deserve documentation, but couched in clarity that modern Perl avoids such structures, and does not recommend their use in new projects.

Another way to look at this philosophy (and one borrowed from our friends <https://devguide.python.org/documenting/#affirmative-tone> on Python's documentation team) involves writing while sympathizing with a programmer new to Perl, who may feel uncertain about learning a complex concept. By leading that concept's main documentation with clear, positive

examples, we can immediately give these readers a simple and true picture of how it works in Perl, and boost their own confidence to start making use of this new knowledge. Certainly we should include alternate routes and admonitions as reasonably required, but we needn't emphasize them. Trust the reader to understand the basics quickly, and to keep reading for a deeper understanding if they feel so driven.

Document Perl's present

Perl's documentation should stay focused on Perl's present behavior, with a nod to future directions.

Recount the past only when necessary

When some Perl feature changes its behavior, documentation about that feature should change too, and just as definitively. The docs have no obligation to keep descriptions of past behavior hanging around, even if attaching clauses like "Prior to version 5.10, [...]".

Since Perl's core documentation is part of Perl's source distribution, it enjoys the same benefits of versioning and version-control as the source code of Perl itself. Take advantage of this, and update the text boldly when needed. Perl's history remains safe, even when you delete or replace outdated information from the current version's docs.

Perl's docs can acknowledge or discuss former behavior when warranted, including notes that some feature appeared in the language as of some specific version number. Authors should consider applying principles similar to those for deprecated techniques, as described above: make the information present, but not prominent.

Otherwise, keep the past in the past. A manual uncluttered with outdated instruction stays more succinct and relevant.

Describe the uncertain future with care

Perl features marked as "experimental"--those that generate warnings when used in code not invoking the "experimental" pragma--deserve documentation, but only in certain contexts, and even then with caveats. These features represent possible new directions for Perl, but they have unstable interfaces and uncertain future presence.

The documentation should take both implications of "experimental" literally. It should not discourage these features' use by programmers who wish to try out new features in projects that can risk their inherent instability; this experimentation can help Perl grow and improve. By the same token, the docs should downplay these features' use in just about every other context.

Introductory or overview material should omit coverage of experimental features altogether.

More thorough reference materials or explanatory articles can include experimental features, but needs to clearly mark them as such, and not treat them with the same prominence as Perl's stable features. Using unstable features seldom coincides with best practices, and documentation that puts best practices first should reflect this.

The documentation speaks with one voice

Even though it comes from many hands and minds, criss-crossing through the many years of Perl's lifetime, the language's documentation should speak with a single, consistent voice. With few exceptions, the docs should avoid explicit first-person-singular statements, or similar self-reference to any individual's contributor's philosophies or experiences.

Perl did begin life as a deeply personal expression by a single individual, and this famously carried through the first revisions of its documentation as well. Today, Perl's community understands that the language's continued development and support comes from many people working in concert, rather than any one person's vision or effort. Its documentation should not pretend otherwise.

The documentation should, however, carry forward the best tradition that Larry Wall set forth in the language's earliest days: Write both economically and with a humble, subtle wit, resulting in a technical manual that mixes concision with a friendly approachability. It avoids the dryness that one might expect from technical documentation, while not leaning so hard into overt comedy as to distract and confuse from the nonetheless-technical topics at hand.

Like the best written works, Perl's documentation has a soul. Get familiar with it as a reader to internalize its voice, and then find your own way to express it in your own contributions. Writing clearly, succinctly, and with knowledge of your audience's expectations will get you most of the way there, in the meantime.

Every line in the docs--whether English sentence or Perl statement--should serve the purpose of bringing understanding to the reader. Should a sentence exist mainly to make a wry joke that doesn't further the reader's knowledge of Perl, set it aside, and consider recasting it into a personal blog post or other article instead.

Write with a light heart, and a miserly hand.

INDEX OF PREFERRED TERMS

As noted above, this guide "inherits" all the preferred terms listed in the Chicago Manual of Style, 17th edition, and adds the following terms of particular interest to Perl documentation.

built-in function
: Not "builtin".

Darwin
: See macOS.

macOS
: Use this term for Apple’s operating system instead of "Mac OS X" or variants thereof.

  This term is also preferable to "Darwin", unless one needs to refer to macOS’s Unix layer specifically.

man page
: One unit of Unix-style documentation. Not "manpage". Preferable to "manual page".

Perl; perl
: The name of the programming language is Perl, with a leading capital "P", and the remainder in lowercase. (Never "PERL".)

  The interpreter program that reads and executes Perl code is named ""perl"", in lowercase and in monospace (as with any other command name).

  Generally, unless you are specifically writing about the command-line "perl" program (as, for example, "perlrun" does), use "Perl" instead.

Perl 5
: Documentation need not follow Perl’s name with a "5", or any other number, except during discussions of Perl’s history, future plans, or explicit comparisons between major Perl versions.

  Before 2019, specifying "Perl 5" was sometimes needed to distinguish the language from Perl 6. With the latter’s renaming to "Raku", this practice became unnecessary.

Perl 6
: See Raku.

Perl 5 Porters, the; porters, the; p5p
: The full name of the team responsible for Perl’s ongoing maintenance and development is "the Perl 5 Porters", and this sobriquet should be spelled out in the first mention within any one document. It may thereafter call the team "the porters" or "p5p".

Not "Perl5 Porters".

program
: The most general descriptor for a stand-alone work made out of executable Perl code. Synonymous with, and preferable to, "script".

Raku
: Perl’s "sister language", whose homepage is <https://raku.org>.

  Previously known as "Perl 6". In 2019, its design team renamed the language to better reflect its identity as a project independent from Perl. As such, Perl’s documentation should always refer to this language as "Raku" and not "Perl 6".

script
: See program.

semicolon
: Perl code’s frequently overlooked punctuation mark. Not "semi-colon".

Unix
: Not "UNIX", "*nix", or "Un*x". Applicable to both the original operating system from the 1970s as well as all its conceptual descendants. You may simply write "Unix" and not "a Unix-like operating system" when referring to a Unix-like operating system.

SEE ALSO

- perlpod
- perlpodstyle

AUTHOR

This guide was initially drafted by Jason McIntosh (jmac@jmac.org), under a grant from The Perl Foundation.