NAME

perlpod - the Plain Old Documentation format

DESCRIPTION

Pod is a simple-to-use markup language used for writing documentation for Perl, Perl programs, and Perl modules.

Translators are available for converting Pod to various formats like plain text, HTML, man pages, and more.

Pod markup consists of three basic kinds of paragraphs: *ordinary*, *verbatim*, and *command*.

Ordinary Paragraph

Most paragraphs in your documentation will be ordinary blocks of text, like this one. You can simply type in your text without any markup whatsoever, and with just a blank line before and after. When it gets formatted, it will undergo minimal formatting, like being rewrapped, probably put into a proportionally spaced font, and maybe even justified.

You can use formatting codes in ordinary paragraphs, for **bold**, *italic*, `code-style`, *hyperlinks*, and more. Such codes are explained in the "*Formatting Codes*" section, below.

Verbatim Paragraph

Verbatim paragraphs are usually used for presenting a codeblock or other text which does not require any special parsing or formatting, and which shouldn't be wrapped.

A verbatim paragraph is distinguished by having its first character be a space or a tab. (And commonly, all its lines begin with spaces and/or tabs.) It should be reproduced exactly, with tabs assumed to be on 8-column boundaries. There are no special formatting codes, so you can't italicize or anything like that. A \ means \, and nothing else.

Command Paragraph

A command paragraph is used for special treatment of whole chunks of text, usually as headings or parts of lists.

All command paragraphs (which are typically only one line long) start with "=", followed by an identifier, followed by arbitrary text that the command can use however it pleases. Currently recognized commands are

```
=pod
=head1 Heading Text
=head2 Heading Text
=head3 Heading Text
=head4 Heading Text
=over indentlevel
=item stuff
=back
=begin format
=end format
=for format text...
=encoding type
=cut
```

To explain them each in detail:

`=head1` *`Heading Text`*

`=head2` *`Heading Text`*

`=head3` *`Heading Text`*

`=head4` *`Heading Text`*

Head1 through head4 produce headings, head1 being the highest level. The text in the rest of this paragraph is the content of the heading. For example:

```
=head2 Object Attributes
```

The text "Object Attributes" comprises the heading there. (Note that head3 and head4 are recent additions, not supported in older Pod translators.) The text in these heading commands can use formatting codes, as seen here:

```
=head2 Possible Values for C<$/>
```

Such commands are explained in the "*Formatting Codes*" section, below.

`=over` *`indentlevel`*

`=item` *`stuff...`*

`=back`

Item, over, and back require a little more explanation: "=over" starts a region specifically for the generation of a list using "=item" commands, or for indenting (groups of) normal paragraphs. At the end of your list, use "=back" to end it. The *indentlevel* option to "=over" indicates how far over to indent, generally in ems (where one em is the width of an "M" in the document's base font) or roughly comparable units; if there is no *indentlevel* option, it defaults to four. (And some formatters may just ignore whatever *indentlevel* you provide.) In the *stuff* in `=item` *`stuff...`*, you may use formatting codes, as seen here:

```
=item Using C<$|> to Control Buffering
```

Such commands are explained in the "*Formatting Codes*" section, below.

Note also that there are some basic rules to using "=over" ... "=back" regions:

- Don't use "=item"s outside of an "=over" ... "=back" region.
- The first thing after the "=over" command should be an "=item", unless there aren't going to be any items at all in this "=over" ... "=back" region.
- Don't put "=head*n*" commands inside an "=over" ... "=back" region.
- And perhaps most importantly, keep the items consistent: either use "=item *" for all of them, to produce bullets; or use "=item 1.", "=item 2.", etc., to produce numbered lists; or use "=item foo", "=item bar", etc. -- namely, things that look nothing like bullets or numbers.

  If you start with bullets or numbers, stick with them, as formatters use the first "=item" type to decide how to format the list.

`=cut`

To end a Pod block, use a blank line, then a line beginning with "=cut", and a blank line after it. This lets Perl (and the Pod formatter) know that this is where Perl code is resuming. (The blank line before the "=cut" is not technically necessary, but many older Pod processors require it.)

`=pod`

The "=pod" command by itself doesn't do much of anything, but it signals to Perl (and Pod formatters) that a Pod block starts here. A Pod block starts with *any* command paragraph, so a "=pod" command is usually used just when you want to start a Pod block with an ordinary paragraph or a verbatim paragraph. For example:

```
=item stuff()
```

```
This function does stuff.

=cut

sub stuff {
  ...
}

=pod

Remember to check its return value, as in:

  stuff() || die "Couldn't do stuff!";

=cut
```

=begin *formatname*

=end *formatname*

=for *formatname text...*

For, begin, and end will let you have regions of text/code/data that are not generally interpreted as normal Pod text, but are passed directly to particular formatters, or are otherwise special. A formatter that can use that format will use the region, otherwise it will be completely ignored.

A command "=begin *formatname*", some paragraphs, and a command "=end *formatname*", mean that the text/data in between is meant for formatters that understand the special format called *formatname*. For example,

```
=begin html

<hr> <img src="thang.png">
<p> This is a raw HTML paragraph </p>

=end html
```

The command "=for *formatname text...*" specifies that the remainder of just this paragraph (starting right after *formatname*) is in that special format.

```
=for html <hr> <img src="thang.png">
<p> This is a raw HTML paragraph </p>
```

This means the same thing as the above "=begin html" ... "=end html" region.

That is, with "=for", you can have only one paragraph's worth of text (i.e., the text in "=foo targetname text..."), but with "=begin targetname" ... "=end targetname", you can have any amount of stuff inbetween. (Note that there still must be a blank line after the "=begin" command and a blank line before the "=end" command.

Here are some examples of how to use these:

```
=begin html

<br>Figure 1.<br><IMG SRC="figure1.png"><br>

=end html

=begin text

  ---------------
```

```
    |  foo        |
    |        bar  |
    ---------------

  ^^^^ Figure 1. ^^^^

  =end text
```

Some format names that formatters currently are known to accept include "roff", "man", "latex", "tex", "text", and "html". (Some formatters will treat some of these as synonyms.)

A format name of "comment" is common for just making notes (presumably to yourself) that won't appear in any formatted version of the Pod document:

```
  =for comment
  Make sure that all the available options are documented!
```

Some *formatnames* will require a leading colon (as in `"=for :formatname"`, or `"=begin :formatname" ... "=end :formatname"`), to signal that the text is not raw data, but instead *is* Pod text (i.e., possibly containing formatting codes) that's just not for normal formatting (e.g., may not be a normal-use paragraph, but might be for formatting as a footnote).

`=encoding` *`encodingname`*

This command is used for declaring the encoding of a document. Most users won't need this; but if your encoding isn't US-ASCII or Latin-1, then put a `=encoding` *`encodingname`* command early in the document so that pod formatters will know how to decode the document. For *encodingname*, use a name recognized by the *Encode::Supported* module. Examples:

```
  =encoding utf8

  =encoding koi8-r

  =encoding ShiftJIS

  =encoding big5
```

`=encoding` affects the whole document, and must occur only once.

And don't forget, when using any other command, that the command lasts up until the end of its *paragraph*, not its line. So in the examples below, you can see that every command needs the blank line after it, to end its paragraph.

Some examples of lists include:

```
  =over

  =item *

  First item

  =item *

  Second item

  =back
```

```
=over

=item Foo()

Description of Foo function

=item Bar()

Description of Bar function

=back
```

Formatting Codes

In ordinary paragraphs and in some command paragraphs, various formatting codes (a.k.a. "interior sequences") can be used:

`I<text>` -- italic text

Used for emphasis ("`be I<careful!>`") and parameters ("`redo I<LABEL>`")

`B<text>` -- bold text

Used for switches ("`perl's B<-n> switch`"), programs ("`some systems provide a B<chfn> for that`"), emphasis ("`be B<careful!>`"), and so on ("`and that feature is known as B<autovivification>`").

`C<code>` -- code text

Renders code in a typewriter font, or gives some other indication that this represents program text ("`C<gmtime($^T)>`") or some other form of computerese ("`C<drwxr-xr-x>`").

`L<name>` -- a hyperlink

There are various syntaxes, listed below. In the syntaxes given, `text`, `name`, and `section` cannot contain the characters '/' and '|'; and any '<' or '>' should be matched.

- `L<name>`

  Link to a Perl manual page (e.g., `L<Net::Ping>`). Note that `name` should not contain spaces. This syntax is also occasionally used for references to UNIX man pages, as in `L<crontab(5)>`.

- `L<name/"sec">` or `L<name/sec>`

  Link to a section in other manual page. E.g., `L<perlsyn/"For Loops">`

- `L</"sec">` or `L</sec>` or `L<"sec">`

  Link to a section in this manual page. E.g., `L</"Object Methods">`

A section is started by the named heading or item. For example, `L<perlvar/$.>` or `L<perlvar/"$.">` both link to the section started by "`=item $.`" in perlvar. And `L<perlsyn/For Loops>` or `L<perlsyn/"For Loops">` both link to the section started by "`=head2 For Loops`" in perlsyn.

To control what text is used for display, you use "`L<text|...>`", as in:

- `L<text|name>`

  Link this text to that manual page. E.g., `L<Perl Error Messages|perldiag>`

- `L<text|name/"sec">` or `L<text|name/sec>`

  Link this text to that section in that manual page. E.g., `L<postfix`

"if"|perlsyn/"Statement Modifiers">

- L<text|/"sec"> or L<text|/sec> or L<text|"sec">

  Link this text to that section in this manual page. E.g., L<the various attributes|/"Member Data">

Or you can link to a web page:

- L<scheme:...>

  Links to an absolute URL. For example, L<http://www.perl.org/>. But note that there is no corresponding L<text|scheme:...> syntax, for various reasons.

E<escape> -- a character escape

Very similar to HTML/XML &*foo*; "entity references":

- E<lt> -- a literal < (less than)
- E<gt> -- a literal > (greater than)
- E<verbar> -- a literal | (*ver*tical *bar*)
- E<sol> = a literal / (*sol*idus)

  The above four are optional except in other formatting codes, notably L<...>, and when preceded by a capital letter.

- E<htmlname>

  Some non-numeric HTML entity name, such as E<eacute>, meaning the same thing as é in HTML -- i.e., a lowercase e with an acute (/-shaped) accent.

- E<number>

  The ASCII/Latin-1/Unicode character with that number. A leading "0x" means that *number* is hex, as in E<0x201E>. A leading "0" means that *number* is octal, as in E<075>. Otherwise *number* is interpreted as being in decimal, as in E<181>.

  Note that older Pod formatters might not recognize octal or hex numeric escapes, and that many formatters cannot reliably render characters above 255. (Some formatters may even have to use compromised renderings of Latin-1 characters, like rendering E<eacute> as just a plain "e".)

F<filename> -- used for filenames

Typically displayed in italics. Example: "F<.cshrc>"

S<text> -- text contains non-breaking spaces

This means that the words in *text* should not be broken across lines. Example: S<$x ? $y : $z>.

X<topic name> -- an index entry

This is ignored by most formatters, but some may use it for building indexes. It always renders as empty-string. Example: X<absolutizing relative URLs>

Z<> -- a null (zero-effect) formatting code

This is rarely used. It's one way to get around using an E<...> code sometimes. For example, instead of "NE<lt>3" (for "N<3") you could write "NZ<><3" (the "Z<>" breaks up the "N" and the "<" so they can't be considered the part of a (fictitious) "N<...>" code.

Most of the time, you will need only a single set of angle brackets to delimit the beginning and end of formatting codes. However, sometimes you will want to put a real right angle bracket (a greater-than sign, '>') inside of a formatting code. This is particularly common when using a formatting code to

provide a different font-type for a snippet of code. As with all things in Perl, there is more than one way to do it. One way is to simply escape the closing bracket using an E code:

```
C<$a E<lt>=E<gt> $b>
```

This will produce: "$a <=> $b"

A more readable, and perhaps more "plain" way is to use an alternate set of delimiters that doesn't require a single ">" to be escaped. With the Pod formatters that are standard starting with perl5.5.660, doubled angle brackets ("<<" and ">>") may be used *if and only if there is whitespace right after the opening delimiter and whitespace right before the closing delimiter!* For example, the following will do the trick:

```
C<< $a <=> $b >>
```

In fact, you can use as many repeated angle-brackets as you like so long as you have the same number of them in the opening and closing delimiters, and make sure that whitespace immediately follows the last '<' of the opening delimiter, and immediately precedes the first '>' of the closing delimiter. (The whitespace is ignored.) So the following will also work:

```
C<<< $a <=> $b >>>
C<<<<  $a <=> $b     >>>>
```

And they all mean exactly the same as this:

```
C<$a E<lt>=E<gt> $b>
```

As a further example, this means that if you wanted to put these bits of code in C (code) style:

```
open(X, ">>thing.dat") || die $!
$foo->bar();
```

you could do it like so:

```
C<<< open(X, ">>thing.dat") || die $! >>>
C<< $foo->bar(); >>
```

which is presumably easier to read than the old way:

```
C<open(X, "E<gt>E<gt>thing.dat") || die $!>
C<$foo-E<gt>bar();>
```

This is currently supported by pod2text (Pod::Text), pod2man (Pod::Man), and any other pod2xxx or Pod::Xxxx translators that use Pod::Parser 1.093 or later, or Pod::Tree 1.02 or later.

The Intent

The intent is simplicity of use, not power of expression. Paragraphs look like paragraphs (block format), so that they stand out visually, and so that I could run them through `fmt` easily to reformat them (that's F7 in my version of **vi**, or Esc Q in my version of **emacs**). I wanted the translator to always leave the ' and ` and " quotes alone, in verbatim mode, so I could slurp in a working program, shift it over four spaces, and have it print out, er, verbatim. And presumably in a monospace font.

The Pod format is not necessarily sufficient for writing a book. Pod is just meant to be an idiot-proof common source for nroff, HTML, TeX, and other markup languages, as used for online documentation. Translators exist for **pod2text**, **pod2html**, **pod2man** (that's for nroff(1) and troff(1)), **pod2latex**, and **pod2fm**. Various others are available in CPAN.

Embedding Pods in Perl Modules

You can embed Pod documentation in your Perl modules and scripts. Start your documentation with an empty line, a "=head1" command at the beginning, and end it with a "=cut" command and an empty line. Perl will ignore the Pod text. See any of the supplied library modules for examples. If you're going to put your Pod at the end of the file, and you're using an __END__ or __DATA__ cut mark, make sure to put an empty line there before the first Pod command.

```
__END__

=head1 NAME

Time::Local - efficiently compute time from local and GMT time
```

Without that empty line before the "=head1", many translators wouldn't have recognized the "=head1" as starting a Pod block.

Hints for Writing Pod

- The **podchecker** command is provided for checking Pod syntax for errors and warnings. For example, it checks for completely blank lines in Pod blocks and for unknown commands and formatting codes. You should still also pass your document through one or more translators and proofread the result, or print out the result and proofread that. Some of the problems found may be bugs in the translators, which you may or may not wish to work around.
- If you're more familiar with writing in HTML than with writing in Pod, you can try your hand at writing documentation in simple HTML, and converting it to Pod with the experimental *Pod::HTML2Pod* module, (available in CPAN), and looking at the resulting code. The experimental *Pod::PXML* module in CPAN might also be useful.
- Many older Pod translators require the lines before every Pod command and after every Pod command (including "=cut"!) to be a blank line. Having something like this:

  ```
  # - - - - - - - - - - - -
  =item $firecracker->boom()

  This noisily detonates the firecracker object.
  =cut
  sub boom {
  ...
  ```

  ...will make such Pod translators completely fail to see the Pod block at all.

  Instead, have it like this:

  ```
  # - - - - - - - - - - - -

  =item $firecracker->boom()

  This noisily detonates the firecracker object.

  =cut

  sub boom {
  ...
  ```

- Some older Pod translators require paragraphs (including command paragraphs like "=head2 Functions") to be separated by *completely* empty lines. If you have an apparently empty line with some spaces on it, this might not count as a separator for those translators, and that

could cause odd formatting.

- Older translators might add wording around an L<> link, so that `L<Foo::Bar>` may become "the Foo::Bar manpage", for example. So you shouldn't write things like `the L<foo> documentation`, if you want the translated document to read sensibly -- instead write `the L<Foo::Bar|Foo::Bar> documentation` or `L<the Foo::Bar documentation|Foo::Bar>`, to control how the link comes out.
- Going past the 70th column in a verbatim block might be ungracefully wrapped by some formatters.

SEE ALSO

perlpodspec, *"PODs: Embedded Documentation" in perlsyn*, *perlnewmod*, *perldoc*, *pod2html*, *pod2man*, *podchecker*.

AUTHOR

Larry Wall, Sean M. Burke