NAME

perlpodspec – Plain Old Documentation: format specification and notes

DESCRIPTION

This document is detailed notes on the Pod markup language. Most people will only have to read perlpod to know how to write in Pod, but this document may answer some incidental questions to do with parsing and rendering Pod.

In this document, ‘‘must’’ / ‘‘must not’’, ‘‘should’’ / ‘‘should not’’, and ‘‘may’’ have their conventional (cf. RFC 2119) meanings: ‘‘X must do Y’’ means that if X doesn’t do Y, it’s against this specification, and should really be fixed. ‘‘X should do Y’’ means that it’s recommended, but X may fail to do Y, if there’s a good reason. ‘‘X may do Y’’ is merely a note that X can do Y at will (although it is up to the reader to detect any connotation of "and I think it would be *nice* if X did Y‘‘ versus ’’it wouldn’t really *bother* me if X did Y").

Notably, when I say ‘‘the parser should do Y’’, the parser may fail to do Y, if the calling application explicitly requests that the parser *not* do Y. I often phrase this as ‘‘the parser should, by default, do Y.’’ This doesn’t *require* the parser to provide an option for turning off whatever feature Y is (like expanding tabs in verbatim paragraphs), although it implicates that such an option *may* be provided.

Pod Definitions

Pod is embedded in files, typically Perl source files, although you can write a file that’s nothing but Pod.

A **line** in a file consists of zero or more non-newline characters, terminated by either a newline or the end of the file.

A **newline sequence** is usually a platform-dependent concept, but Pod parsers should understand it to mean any of CR (ASCII 13), LF (ASCII 10), or a CRLF (ASCII 13 followed immediately by ASCII 10), in addition to any other system-specific meaning. The first CR/CRLF/LF sequence in the file may be used as the basis for identifying the newline sequence for parsing the rest of the file.

A **blank line** is a line consisting entirely of zero or more spaces (ASCII 32) or tabs (ASCII 9), and terminated by a newline or end-of-file. A **non-blank line** is a line containing one or more characters other than space or tab (and terminated by a newline or end-of-file).

(*Note:* Many older Pod parsers did not accept a line consisting of spaces/tabs and then a newline as a blank line. The only lines they considered blank were lines consisting of *no characters at all*, terminated by a newline.)

Whitespace is used in this document as a blanket term for spaces, tabs, and newline sequences. (By itself, this term usually refers to literal whitespace. That is, sequences of whitespace characters in Pod source, as opposed to ‘‘E<32>’’, which is a formatting code that *denotes* a whitespace character.)

A **Pod parser** is a module meant for parsing Pod (regardless of whether this involves calling callbacks or building a parse tree or directly formatting it). A **Pod formatter** (or **Pod translator**) is a module or program that converts Pod to some other format (HTML, plaintext, TeX, PostScript, RTF). A **Pod processor** might be a formatter or translator, or might be a program that does something else with the Pod (like counting words, scanning for index points, etc.).

Pod content is contained in **Pod blocks**. A Pod block starts with a line that matches `m/\A=[a-zA-Z]/`, and continues up to the next line that matches `m/\A=cut/` or up to the end of the file if there is no `m/\A=cut/` line.

Within a Pod block, there are **Pod paragraphs**. A Pod paragraph consists of non-blank lines of text, separated by one or more blank lines.

For purposes of Pod processing, there are four types of paragraphs in a Pod block:

- A command paragraph (also called a ‘‘directive’’). The first line of this paragraph must match `m/\A=[a-zA-Z]/`. Command paragraphs are typically one line, as in:

  ```
  =head1 NOTES
  ```

```
=item *
```

But they may span several (non-blank) lines:

```
=for comment
Hm, I wonder what it would look like if
you tried to write a BNF for Pod from this.

=head3 Dr. Strangelove, or: How I Learned to
Stop Worrying and Love the Bomb
```

Some command paragraphs allow formatting codes in their content (i.e., after the part that matches `m/\A=[a-zA-Z]\S*\s*/`), as in:

```
=head1 Did You Remember to C<use strict;>?
```

In other words, the Pod processing handler for "head1" will apply the same processing to "Did You Remember to C<use strict;>?" that it would to an ordinary paragraph (i.e., formatting codes like "C<...>") are parsed and presumably formatted appropriately, and whitespace in the form of literal spaces and/or tabs is not significant.

- A **verbatim paragraph**. The first line of this paragraph must be a literal space or tab, and this paragraph must not be inside a "=begin *identifier*", ... "=end *identifier*" sequence unless "*identifier*" begins with a colon (":"). That is, if a paragraph starts with a literal space or tab, but *is* inside a "=begin *identifier*", ... "=end *identifier*" region, then it's a data paragraph, unless "*identifier*" begins with a colon.

  Whitespace *is* significant in verbatim paragraphs (although, in processing, tabs are probably expanded).

- An **ordinary paragraph**. A paragraph is an ordinary paragraph if its first line matches neither `m/\A=[a-zA-Z]/` nor `m/\A[ \t]/`, *and* if it's not inside a "=begin *identifier*", ... "=end *identifier*" sequence unless "*identifier*" begins with a colon (":").

- A **data paragraph**. This is a paragraph that *is* inside a "=begin *identifier*" ... "=end *identifier*" sequence where "*identifier*" does *not* begin with a literal colon (":"). In some sense, a data paragraph is not part of Pod at all (i.e., effectively it's "out-of-band"), since it's not subject to most kinds of Pod parsing; but it is specified here, since Pod parsers need to be able to call an event for it, or store it in some form in a parse tree, or at least just parse *around* it.

For example: consider the following paragraphs:

```
# <- that's the 0th column

=head1 Foo

Stuff

  $foo->bar

=cut
```

Here, "=head1 Foo" and "=cut" are command paragraphs because the first line of each matches `m/\A=[a-zA-Z]/`. "*[space][space]*`$foo->`bar" is a verbatim paragraph, because its first line starts with a literal whitespace character (and there's no "=begin"..."=end" region around).

The "=begin *identifier*" ... "=end *identifier*" commands stop paragraphs that they surround from being parsed as ordinary or verbatim paragraphs, if *identifier* doesn't begin with a colon. This is discussed in detail in the section "About Data Paragraphs and "=begin/=end" Regions".

Pod Commands

This section is intended to supplement and clarify the discussion in ‘‘Command Paragraph’’ in perlpod. These are the currently recognized Pod commands:

‘‘=head1’’, ‘‘=head2’’, ‘‘=head3’’, ‘‘=head4’’

This command indicates that the text in the remainder of the paragraph is a heading. That text may contain formatting codes. Examples:

```
=head1 Object Attributes

=head3 What B<Not> to Do!
```

‘‘=pod’’

This command indicates that this paragraph begins a Pod block. (If we are already in the middle of a Pod block, this command has no effect at all.) If there is any text in this command paragraph after ‘‘=pod’’, it must be ignored. Examples:

```
=pod

This is a plain Pod paragraph.

=pod This text is ignored.
```

‘‘=cut’’

This command indicates that this line is the end of this previously started Pod block. If there is any text after ‘‘=cut’’ on the line, it must be ignored. Examples:

```
=cut

=cut The documentation ends here.

=cut
# This is the first line of program text.
sub foo { # This is the second.
```

It is an error to try to *start* a Pod block with a ‘‘=cut’’ command. In that case, the Pod processor must halt parsing of the input file, and must by default emit a warning.

‘‘=over’’

This command indicates that this is the start of a list/indent region. If there is any text following the ‘‘=over’’, it must consist of only a nonzero positive numeral. The semantics of this numeral is explained in the ‘‘About =over...=back Regions’’ section, further below. Formatting codes are not expanded. Examples:

```
=over 3

=over 3.5

=over
```

‘‘=item’’

This command indicates that an item in a list begins here. Formatting codes are processed. The semantics of the (optional) text in the remainder of this paragraph are explained in the ‘‘About =over...=back Regions’’ section, further below. Examples:

```
=item

=item *

=item      *
```

```
=item 14

=item   3.

=item C<< $thing->stuff(I<dodad>) >>

=item For transporting us beyond seas to be tried for pretended
offenses

=item He is at this time transporting large armies of foreign
mercenaries to complete the works of death, desolation and
tyranny, already begun with circumstances of cruelty and perfidy
scarcely paralleled in the most barbarous ages, and totally
unworthy the head of a civilized nation.
```

“=back”

This command indicates that this is the end of the region begun by the most recent “=over” command. It permits no text after the “=back” command.

“=begin formatname”

“=begin formatname parameter”

This marks the following paragraphs (until the matching “=end formatname”) as being for some special kind of processing. Unless “formatname” begins with a colon, the contained non-command paragraphs are data paragraphs. But if “formatname” *does* begin with a colon, then non-command paragraphs are ordinary paragraphs or data paragraphs. This is discussed in detail in the section “About Data Paragraphs and ”=begin/=end“ Regions”.

It is advised that formatnames match the regexp `m/\A:?[-a-zA-Z0-9_]+\z/`. Everything following whitespace after the formatname is a parameter that may be used by the formatter when dealing with this region. This parameter must not be repeated in the “=end” paragraph. Implementors should anticipate future expansion in the semantics and syntax of the first parameter to “=begin”/“=end”/“=for”.

“=end formatname”

This marks the end of the region opened by the matching “=begin formatname” region. If “formatname” is not the formatname of the most recent open “=begin formatname” region, then this is an error, and must generate an error message. This is discussed in detail in the section “About Data Paragraphs and ”=begin/=end“ Regions”.

“=for formatname text...”

This is synonymous with:

```
=begin formatname

text...

=end formatname
```

That is, it creates a region consisting of a single paragraph; that paragraph is to be treated as a normal paragraph if “formatname” begins with a “:”; if “formatname” *doesn't* begin with a colon, then “text...” will constitute a data paragraph. There is no way to use “=for formatname text...” to express “text...” as a verbatim paragraph.

“=encoding encodingname”

This command, which should occur early in the document (at least before any non-US-ASCII data!), declares that this document is encoded in the encoding *encodingname*, which must be an encoding name that Encode recognizes. (Encode's list of supported encodings, in Encode::Supported, is useful here.) If the Pod parser cannot decode the declared encoding, it should emit a warning and may abort

parsing the document altogether.

A document having more than one ‘‘=encoding’’ line should be considered an error. Pod processors may silently tolerate this if the not-first ‘‘=encoding’’ lines are just duplicates of the first one (e.g., if there’s a ‘‘=encoding utf8’’ line, and later on another ‘‘=encoding utf8’’ line). But Pod processors should complain if there are contradictory ‘‘=encoding’’ lines in the same document (e.g., if there is a ‘‘=encoding utf8’’ early in the document and ‘‘=encoding big5’’ later). Pod processors that recognize BOMs may also complain if they see an ‘‘=encoding’’ line that contradicts the BOM (e.g., if a document with a UTF−16LE BOM has an ‘‘=encoding shiftjis’’ line).

If a Pod processor sees any command other than the ones listed above (like ‘‘=head’’, or ‘‘=haed1’’, or ‘‘=stuff’’, or ‘‘=cuttlefish’’, or ‘‘=w123’’), that processor must by default treat this as an error. It must not process the paragraph beginning with that command, must by default warn of this as an error, and may abort the parse. A Pod parser may allow a way for particular applications to add to the above list of known commands, and to stipulate, for each additional command, whether formatting codes should be processed.

Future versions of this specification may add additional commands.

Pod Formatting Codes

(Note that in previous drafts of this document and of perlpod(1), formatting codes were referred to as ‘‘interior sequences’’, and this term may still be found in the documentation for Pod parsers, and in error messages from Pod processors.)

There are two syntaxes for formatting codes:

- A formatting code starts with a capital letter (just US-ASCII [A−Z]) followed by a ‘‘<’’, any number of characters, and ending with the first matching ‘‘>’’. Examples:

```
That's what I<you> think!

What's C<dump()> for?

X<C<chmod> and C<unlink()> Under Different Operating Systems>
```

- A formatting code starts with a capital letter (just US-ASCII [A−Z]) followed by two or more ‘‘<’’’s, one or more whitespace characters, any number of characters, one or more whitespace characters, and ending with the first matching sequence of two or more ‘‘>’’’s, where the number of ‘‘>’’’s equals the number of ‘‘<’’’s in the opening of this formatting code. Examples:

```
That's what I<< you >> think!

C<<< open(X, ">>thing.dat") || die $! >>>

B<< $foo->bar(); >>
```

With this syntax, the whitespace character(s) after the ‘‘C<<<’’ and before the ‘‘>>>’’ (or whatever letter) are *not* renderable. They do not signify whitespace, are merely part of the formatting codes themselves. That is, these are all synonymous:

```
C<thing>
C<< thing >>
C<<           thing     >>
C<<<   thing >>>
C<<<<
thing
           >>>>
```

and so on.

Finally, the multiple-angle-bracket form does *not* alter the interpretation of nested formatting codes, meaning that the following four example lines are identical in meaning:

```
B<example: C<$a E<lt>=E<gt> $b>>

B<example: C<< $a <=> $b >>>

B<example: C<< $a E<lt>=E<gt> $b >>>

B<<< example: C<< $a E<lt>=E<gt> $b >> >>>
```

In parsing Pod, a notably tricky part is the correct parsing of (potentially nested!) formatting codes. Implementors should consult the code in the `parse_text` routine in Pod::Parser as an example of a correct implementation.

`I<text>` — italic text
: See the brief discussion in "Formatting Codes" in perlpod.

`B<text>` — bold text
: See the brief discussion in "Formatting Codes" in perlpod.

`C<code>` — code text
: See the brief discussion in "Formatting Codes" in perlpod.

`F<filename>` — style for filenames
: See the brief discussion in "Formatting Codes" in perlpod.

`X<topic name>` — an index entry
: See the brief discussion in "Formatting Codes" in perlpod.

  This code is unusual in that most formatters completely discard this code and its content. Other formatters will render it with invisible codes that can be used in building an index of the current document.

`Z<>` — a null (zero-effect) formatting code
: Discussed briefly in "Formatting Codes" in perlpod.

  This code is unusual in that it should have no content. That is, a processor may complain if it sees `Z<potatoes>`. Whether or not it complains, the *potatoes* text should ignored.

`L<name>` — a hyperlink
: The complicated syntaxes of this code are discussed at length in "Formatting Codes" in perlpod(1), and implementation details are discussed below, in "About L<...> Codes". Parsing the contents of L<content> is tricky. Notably, the content has to be checked for whether it looks like a URL, or whether it has to be split on literal "|" and/or "/" (in the right order!), and so on, *before* E<...> codes are resolved.

`E<escape>` — a character escape
: See "Formatting Codes" in perlpod(1), and several points in "Notes on Implementing Pod Processors".

`S<text>` — text contains non-breaking spaces
: This formatting code is syntactically simple, but semantically complex. What it means is that each space in the printable content of this code signifies a non-breaking space.

  Consider:

  ```
  C<$x ? $y    :  $z>

  S<C<$x ? $y     :  $z>>
  ```

  Both signify the monospace (c[ode] style) text consisting of "$x", one space, "?", one space, ":", one space, "$z". The difference is that in the latter, with the S code, those spaces are not "normal" spaces, but instead are non-breaking spaces.

If a Pod processor sees any formatting code other than the ones listed above (as in "N<...>", or "Q<...>", etc.), that processor must by default treat this as an error. A Pod parser may allow a way for particular

applications to add to the above list of known formatting codes; a Pod parser might even allow a way to stipulate, for each additional command, whether it requires some form of special processing, as L<...> does.

Future versions of this specification may add additional formatting codes.

Historical note: A few older Pod processors would not see a "**>**" as closing a "C<" code, if the "**>**" was immediately preceded by a "**-**". This was so that this:

```
C<$foo->bar>
```

would parse as equivalent to this:

```
C<$foo-E<gt>bar>
```

instead of as equivalent to a "C" formatting code containing only "$foo-", and then a "bar>" outside the "C" formatting code. This problem has since been solved by the addition of syntaxes like this:

```
C<< $foo->bar >>
```

Compliant parsers must not treat "**->**" as special.

Formatting codes absolutely cannot span paragraphs. If a code is opened in one paragraph, and no closing code is found by the end of that paragraph, the Pod parser must close that formatting code, and should complain (as in "Unterminated I code in the paragraph starting at line 123: 'Time objects are not...'"). So these two paragraphs:

```
I<I told you not to do this!

Don't make me say it again!>
```

...must *not* be parsed as two paragraphs in italics (with the I code starting in one paragraph and starting in another.) Instead, the first paragraph should generate a warning, but that aside, the above code must parse as if it were:

```
I<I told you not to do this!>

Don't make me say it again!E<gt>
```

(In SGMLish jargon, all Pod commands are like block-level elements, whereas all Pod formatting codes are like inline-level elements.)

Notes on Implementing Pod Processors

The following is a long section of miscellaneous requirements and suggestions to do with Pod processing.

- Pod formatters should tolerate lines in verbatim blocks that are of any length, even if that means having to break them (possibly several times, for very long lines) to avoid text running off the side of the page. Pod formatters may warn of such line-breaking. Such warnings are particularly appropriate for lines are over 100 characters long, which are usually not intentional.
- Pod parsers must recognize *all* of the three well-known newline formats: CR, LF, and CRLF. See perlport.
- Pod parsers should accept input lines that are of any length.
- Since Perl recognizes a Unicode Byte Order Mark at the start of files as signaling that the file is Unicode encoded as in UTF-16 (whether big-endian or little-endian) or UTF-8, Pod parsers should do the same. Otherwise, the character encoding should be understood as being UTF-8 if the first highbit byte sequence in the file seems valid as a UTF-8 sequence, or otherwise as CP-1252 (earlier versions of this specification used Latin-1 instead of CP-1252).

  Future versions of this specification may specify how Pod can accept other encodings. Presumably treatment of other encodings in Pod parsing would be as in XML parsing: whatever the encoding declared by a particular Pod file, content is to be stored in memory as Unicode characters.

- The well known Unicode Byte Order Marks are as follows: if the file begins with the two literal byte values 0xFE 0xFF, this is the BOM for big-endian UTF-16. If the file begins with the two literal byte value 0xFF 0xFE, this is the BOM for little-endian UTF-16. On an ASCII platform, if the file begins with the three literal byte values 0xEF 0xBB 0xBF, this is the BOM for UTF-8. A mechanism portable to EBCDIC platforms is to:

  ```
  my $utf8_bom = "\x{FEFF}";
  utf8::encode($utf8_bom);
  ```

- A naive, but often sufficient heuristic on ASCII platforms, for testing the first highbit byte-sequence in a BOM-less file (whether in code or in Pod!), to see whether that sequence is valid as UTF-8 (RFC 2279) is to check whether that the first byte in the sequence is in the range 0xC2 – 0xFD *and* whether the next byte is in the range 0x80 – 0xBF. If so, the parser may conclude that this file is in UTF-8, and all highbit sequences in the file should be assumed to be UTF-8. Otherwise the parser should treat the file as being in CP-1252. (A better check, and which works on EBCDIC platforms as well, is to pass a copy of the sequence to **utf8::decode()** which performs a full validity check on the sequence and returns TRUE if it is valid UTF-8, FALSE otherwise. This function is always pre-loaded, is fast because it is written in C, and will only get called at most once, so you don't need to avoid it out of performance concerns.) In the unlikely circumstance that the first highbit sequence in a truly non-UTF-8 file happens to appear to be UTF-8, one can cater to our heuristic (as well as any more intelligent heuristic) by prefacing that line with a comment line containing a highbit sequence that is clearly *not* valid as UTF-8. A line consisting of simply "#", an e-acute, and any non-highbit byte, is sufficient to establish this file's encoding.

- Pod processors must treat a "=for [label] [content...]" paragraph as meaning the same thing as a "=begin [label]" paragraph, content, and an "=end [label]" paragraph. (The parser may conflate these two constructs, or may leave them distinct, in the expectation that the formatter will nevertheless treat them the same.)

- When rendering Pod to a format that allows comments (i.e., to nearly any format other than plaintext), a Pod formatter must insert comment text identifying its name and version number, and the name and version numbers of any modules it might be using to process the Pod. Minimal examples:

  ```
  %% POD::Pod2PS v3.14159, using POD::Parser v1.92

  <!-- Pod::HTML v3.14159, using POD::Parser v1.92 -->

  {\doccomm generated by Pod::Tree::RTF 3.14159 using Pod::Tree 1.08}

  .\" Pod::Man version 3.14159, using POD::Parser version 1.92
  ```

  Formatters may also insert additional comments, including: the release date of the Pod formatter program, the contact address for the author(s) of the formatter, the current time, the name of input file, the formatting options in effect, version of Perl used, etc.

  Formatters may also choose to note errors/warnings as comments, besides or instead of emitting them otherwise (as in messages to STDERR, or `die`ing).

- Pod parsers *may* emit warnings or error messages ("Unknown E code E<zslig>!") to STDERR (whether through printing to STDERR, or `warn`ing/`carp`ing, or `die`ing/`croak`ing), but *must* allow suppressing all such STDERR output, and instead allow an option for reporting errors/warnings in some other way, whether by triggering a callback, or noting errors in some attribute of the document object, or some similarly unobtrusive mechanism — or even by appending a "Pod Errors" section to the end of the parsed form of the document.

- In cases of exceptionally aberrant documents, Pod parsers may abort the parse. Even then, using `die`ing/`croak`ing is to be avoided; where possible, the parser library may simply close the input file and add text like "*** Formatting Aborted ***" to the end of the (partial) in-memory document.

- In paragraphs where formatting codes (like E<...>, B<...>) are understood (i.e., *not* verbatim paragraphs, but *including* ordinary paragraphs, and command paragraphs that produce renderable text, like ‘‘=head1’’), literal whitespace should generally be considered ‘‘insignificant’’, in that one literal space has the same meaning as any (nonzero) number of literal spaces, literal newlines, and literal tabs (as long as this produces no blank lines, since those would terminate the paragraph). Pod parsers should compact literal whitespace in each processed paragraph, but may provide an option for overriding this (since some processing tasks do not require it), or may follow additional special rules (for example, specially treating period-space-space or period-newline sequences).
- Pod parsers should not, by default, try to coerce apostrophe (’) and quote (‘‘) into smart quotes (little 9’s, 66’s, 99’s, etc), nor try to turn backtick (‘) into anything else but a single backtick character (distinct from an open quote character!), nor ’’−−" into anything but two minus signs. They *must never* do any of those things to text in C<...> formatting codes, and never *ever* to text in verbatim paragraphs.
- When rendering Pod to a format that has two kinds of hyphens (−), one that’s a non-breaking hyphen, and another that’s a breakable hyphen (as in ‘‘object-oriented’’, which can be split across lines as ‘‘object−’’, newline, ‘‘oriented’’), formatters are encouraged to generally translate ‘‘−’’ to non-breaking hyphen, but may apply heuristics to convert some of these to breaking hyphens.
- Pod formatters should make reasonable efforts to keep words of Perl code from being broken across lines. For example, ‘‘Foo::Bar’’ in some formatting systems is seen as eligible for being broken across lines as ‘‘Foo::’’ newline ‘‘Bar’’ or even ‘‘Foo::−’’ newline ‘‘Bar’’. This should be avoided where possible, either by disabling all line-breaking in mid-word, or by wrapping particular words with internal punctuation in ‘‘don’t break this across lines’’ codes (which in some formats may not be a single code, but might be a matter of inserting non-breaking zero-width spaces between every pair of characters in a word.)
- Pod parsers should, by default, expand tabs in verbatim paragraphs as they are processed, before passing them to the formatter or other processor. Parsers may also allow an option for overriding this.
- Pod parsers should, by default, remove newlines from the end of ordinary and verbatim paragraphs before passing them to the formatter. For example, while the paragraph you’re reading now could be considered, in Pod source, to end with (and contain) the newline(s) that end it, it should be processed as ending with (and containing) the period character that ends this sentence.
- Pod parsers, when reporting errors, should make some effort to report an approximate line number (‘‘Nested E<>’s in Paragraph #52, near line 633 of Thing/Foo.pm!’’), instead of merely noting the paragraph number (‘‘Nested E<>’s in Paragraph #52 of Thing/Foo.pm!’’). Where this is problematic, the paragraph number should at least be accompanied by an excerpt from the paragraph (‘‘Nested E<>’s in Paragraph #52 of Thing/Foo.pm, which begins ’Read/write accessor for the C<interest rate> attribute...’’’).
- Pod parsers, when processing a series of verbatim paragraphs one after another, should consider them to be one large verbatim paragraph that happens to contain blank lines. I.e., these two lines, which have a blank line between them:

  ```
  use Foo;

  print Foo->VERSION
  ```

  should be unified into one paragraph (‘‘\tuse Foo;\n\n\tprint Foo−>VERSION’’) before being passed to the formatter or other processor. Parsers may also allow an option for overriding this.

  While this might be too cumbersome to implement in event-based Pod parsers, it is straightforward for parsers that return parse trees.
- Pod formatters, where feasible, are advised to avoid splitting short verbatim paragraphs (under twelve lines, say) across pages.

- Pod parsers must treat a line with only spaces and/or tabs on it as a "blank line" such as separates paragraphs. (Some older parsers recognized only two adjacent newlines as a "blank line" but would not recognize a newline, a space, and a newline, as a blank line. This is noncompliant behavior.)
- Authors of Pod formatters/processors should make every effort to avoid writing their own Pod parser. There are already several in CPAN, with a wide range of interface styles — and one of them, Pod::Simple, comes with modern versions of Perl.
- Characters in Pod documents may be conveyed either as literals, or by number in E<n> codes, or by an equivalent mnemonic, as in E<eacute> which is exactly equivalent to E<233>. The numbers are the Latin1/Unicode values, even on EBCDIC platforms.

  When referring to characters by using a E<n> numeric code, numbers in the range 32–126 refer to those well known US-ASCII characters (also defined there by Unicode, with the same meaning), which all Pod formatters must render faithfully. Characters whose E<> numbers are in the ranges 0–31 and 127–159 should not be used (neither as literals, nor as E<number> codes), except for the literal byte-sequences for newline (ASCII 13, ASCII 13 10, or ASCII 10), and tab (ASCII 9).

  Numbers in the range 160–255 refer to Latin–1 characters (also defined there by Unicode, with the same meaning). Numbers above 255 should be understood to refer to Unicode characters.
- Be warned that some formatters cannot reliably render characters outside 32–126; and many are able to handle 32–126 and 160–255, but nothing above 255.
- Besides the well-known "E<lt>" and "E<gt>" codes for less-than and greater-than, Pod parsers must understand "E<sol>" for "/" (solidus, slash), and "E<verbar>" for "|" (vertical bar, pipe). Pod parsers should also understand "E<lchevron>" and "E<rchevron>" as legacy codes for characters 171 and 187, i.e., "left-pointing double angle quotation mark" = "left pointing guillemet" and "right-pointing double angle quotation mark" = "right pointing guillemet". (These look like little "<<" and ">>", and they are now preferably expressed with the HTML/XHTML codes "E<laquo>" and "E<raquo>".)
- Pod parsers should understand all "E<html>" codes as defined in the entity declarations in the most recent XHTML specification at `www.W3.org`. Pod parsers must understand at least the entities that define characters in the range 160–255 (Latin–1). Pod parsers, when faced with some unknown "E<*identifier*>" code, shouldn't simply replace it with nullstring (by default, at least), but may pass it through as a string consisting of the literal characters E, less-than, *identifier*, greater-than. Or Pod parsers may offer the alternative option of processing such unknown "E<*identifier*>" codes by firing an event especially for such codes, or by adding a special node-type to the in-memory document tree. Such "E<*identifier*>" may have special meaning to some processors, or some processors may choose to add them to a special error report.
- Pod parsers must also support the XHTML codes "E<quot>" for character 34 (doublequote, "), "E<amp>" for character 38 (ampersand, &), and "E<apos>" for character 39 (apostrophe, ').
- Note that in all cases of "E<whatever>", *whatever* (whether an htmlname, or a number in any base) must consist only of alphanumeric characters — that is, *whatever* must match `m/\A\w+\z/`. So "E<Â 0Â 1Â 2Â 3Â >" is invalid, because it contains spaces, which aren't alphanumeric characters. This presumably does not *need* special treatment by a Pod processor; "Â 0Â 1Â 2Â 3Â " doesn't look like a number in any base, so it would presumably be looked up in the table of HTML-like names. Since there isn't (and cannot be) an HTML-like entity called "Â 0Â 1Â 2Â 3Â ", this will be treated as an error. However, Pod processors may treat "E<Â 0Â 1Â 2Â 3Â >" or "E<e–acute>" as *syntactically* invalid, potentially earning a different error message than the error message (or warning, or event) generated by a merely unknown (but theoretically valid) htmlname, as in "E<qacute>" [sic]. However, Pod parsers are not required to make this distinction.
- Note that E<number> *must not* be interpreted as simply "codepoint *number* in the current/native character set". It always means only "the character represented by codepoint *number* in Unicode." (This is identical to the semantics of &#*number*; in XML.)

  This will likely require many formatters to have tables mapping from treatable Unicode codepoints

(such as the "\xE9" for the e−acute character) to the escape sequences or codes necessary for conveying such sequences in the target output format. A converter to *roff would, for example know that "\xE9" (whether conveyed literally, or via a E<...> sequence) is to be conveyed as "e*"'. Similarly, a program rendering Pod in a Mac OS application window, would presumably need to know that "\xE9" maps to codepoint 142 in MacRoman encoding that (at time of writing) is native for Mac OS. Such Unicode2whatever mappings are presumably already widely available for common output formats. (Such mappings may be incomplete! Implementers are not expected to bend over backwards in an attempt to render Cherokee syllabics, Etruscan runes, Byzantine musical symbols, or any of the other weird things that Unicode can encode.) And if a Pod document uses a character not found in such a mapping, the formatter should consider it an unrenderable character.

- If, surprisingly, the implementor of a Pod formatter can't find a satisfactory pre-existing table mapping from Unicode characters to escapes in the target format (e.g., a decent table of Unicode characters to *roff escapes), it will be necessary to build such a table. If you are in this circumstance, you should begin with the characters in the range 0x00A0 − 0x00FF, which is mostly the heavily used accented characters. Then proceed (as patience permits and fastidiousness compels) through the characters that the (X)HTML standards groups judged important enough to merit mnemonics for. These are declared in the (X)HTML specifications at the www.W3.org site. At time of writing (September 2001), the most recent entity declaration files are:

  ```
  http://www.w3.org/TR/xhtml1/DTD/xhtml-lat1.ent
  http://www.w3.org/TR/xhtml1/DTD/xhtml-special.ent
  http://www.w3.org/TR/xhtml1/DTD/xhtml-symbol.ent
  ```

  Then you can progress through any remaining notable Unicode characters in the range 0x2000−0x204D (consult the character tables at www.unicode.org), and whatever else strikes your fancy. For example, in *xhtml−symbol.ent*, there is the entry:

  ```
  <!ENTITY infin    "&#8734;"> <!-- infinity, U+221E ISOtech -->
  ```

  While the mapping "infin" to the character "\x{221E}" will (hopefully) have been already handled by the Pod parser, the presence of the character in this file means that it's reasonably important enough to include in a formatter's table that maps from notable Unicode characters to the codes necessary for rendering them. So for a Unicode−to−*roff mapping, for example, this would merit the entry:

  ```
  "\x{221E}" => '\(in',
  ```

  It is eagerly hoped that in the future, increasing numbers of formats (and formatters) will support Unicode characters directly (as (X)HTML does with `&infin;`, `&#8734;`, or `&#x221E;`), reducing the need for idiosyncratic mappings of Unicode−to−*my_escapes*.

- It is up to individual Pod formatter to display good judgement when confronted with an unrenderable character (which is distinct from an unknown E<thing> sequence that the parser couldn't resolve to anything, renderable or not). It is good practice to map Latin letters with diacritics (like "E<eacute>"/"E<233>") to the corresponding unaccented US-ASCII letters (like a simple character 101, "e"), but clearly this is often not feasible, and an unrenderable character may be represented as "?", or the like. In attempting a sane fallback (as from E<233> to "e"), Pod formatters may use the `%Latin1Code_to_fallback` table in Pod::Escapes, or Text::Unidecode, if available.

  For example, this Pod text:

  ```
  magic is enabled if you set C<$Currency> to 'E<euro>'.
  ```

  may be rendered as: "magic is enabled if you set `$Currency` to '*?*'" or as "magic is enabled if you set `$Currency` to '**[euro]**'", or as "magic is enabled if you set `$Currency` to '[x20AC]', etc.

  A Pod formatter may also note, in a comment or warning, a list of what unrenderable characters were encountered.

- E<...> may freely appear in any formatting code (other than in another E<...> or in an Z<>). That is, ‘‘X<The E<euro>1,000,000 Solution>’’ is valid, as is ‘‘L<The E<euro>1,000,000 Solution|Million::Euros>’’.

- Some Pod formatters output to formats that implement non-breaking spaces as an individual character (which I’ll call ‘‘NBSP’’), and others output to formats that implement non-breaking spaces just as spaces wrapped in a ‘‘don’t break this across lines’’ code. Note that at the level of Pod, both sorts of codes can occur: Pod can contain a NBSP character (whether as a literal, or as a ‘‘E<160>’’ or ‘‘E<nbsp>’’ code); and Pod can contain ‘‘S<foo I<bar> baz>’’ codes, where ‘‘mere spaces’’ (character 32) in such codes are taken to represent non-breaking spaces. Pod parsers should consider supporting the optional parsing of ‘‘S<foo I<bar> baz>’’ as if it were "foo*NBSP*I<bar>*NBSP*baz", and, going the other way, the optional parsing of groups of words joined by NBSP’s as if each group were in a S<...> code, so that formatters may use the representation that maps best to what the output format demands.

- Some processors may find that the `S<...>` code is easiest to implement by replacing each space in the parse tree under the content of the S, with an NBSP. But note: the replacement should apply *not* to spaces in *all* text, but *only* to spaces in *printable* text. (This distinction may or may not be evident in the particular tree/event model implemented by the Pod parser.) For example, consider this unusual case:

  ```
  S<L</Autoloaded Functions>>
  ```

  This means that the space in the middle of the visible link text must not be broken across lines. In other words, it’s the same as this:

  ```
  L<"AutoloadedE<160>Functions"/Autoloaded Functions>
  ```

  However, a misapplied space-to-NBSP replacement could (wrongly) produce something equivalent to this:

  ```
  L<"AutoloadedE<160>Functions"/AutoloadedE<160>Functions>
  ```

  ...which is almost definitely not going to work as a hyperlink (assuming this formatter outputs a format supporting hypertext).

  Formatters may choose to just not support the S format code, especially in cases where the output format simply has no NBSP character/code and no code for ‘‘don’t break this stuff across lines’’.

- Besides the NBSP character discussed above, implementors are reminded of the existence of the other ‘‘special’’ character in Latin−1, the ‘‘soft hyphen’’ character, also known as ‘‘discretionary hyphen’’, i.e. `E<173>` = `E<0xAD>` = `E<shy>`). This character expresses an optional hyphenation point. That is, it normally renders as nothing, but may render as a ‘‘−’’ if a formatter breaks the word at that point. Pod formatters should, as appropriate, do one of the following: 1) render this with a code with the same meaning (e.g., ‘‘\−’’ in RTF), 2) pass it through in the expectation that the formatter understands this character as such, or 3) delete it.

  For example:

  ```
  sigE<shy>action
  manuE<shy>script
  JarkE<shy>ko HieE<shy>taE<shy>nieE<shy>mi
  ```

  These signal to a formatter that if it is to hyphenate ‘‘sigaction’’ or ‘‘manuscript’’, then it should be done as "sig−*[linebreak]*action‘‘ or ’’manu−*[linebreak]*script" (and if it doesn’t hyphenate it, then the `E<shy>` doesn’t show up at all). And if it is to hyphenate ‘‘Jarkko’’ and/or ‘‘Hietaniemi’’, it can do so only at the points where there is a `E<shy>` code.

  In practice, it is anticipated that this character will not be used often, but formatters should either support it, or delete it.

- If you think that you want to add a new command to Pod (like, say, a "=biblio" command), consider whether you could get the same effect with a for or begin/end sequence: "=for biblio ..." or "=begin biblio" ... "=end biblio". Pod processors that don't understand "=for biblio", etc, will simply ignore it, whereas they may complain loudly if they see "=biblio".
- Throughout this document, "Pod" has been the preferred spelling for the name of the documentation format. One may also use "POD" or "pod". For the documentation that is (typically) in the Pod format, you may use "pod", or "Pod", or "POD". Understanding these distinctions is useful; but obsessing over how to spell them, usually is not.

About L<...> Codes

As you can tell from a glance at perlpod(1), the L<...> code is the most complex of the Pod formatting codes. The points below will hopefully clarify what it means and how processors should deal with it.

- In parsing an L<...> code, Pod parsers must distinguish at least four attributes:

  First:
  : The link-text. If there is none, this must be `undef`. (E.g., in "L<Perl Functions|perlfunc>", the link-text is "Perl Functions". In "L<Time::HiRes>" and even "L<|Time::HiRes>", there is no link text. Note that link text may contain formatting.)

  Second:
  : The possibly inferred link-text; i.e., if there was no real link text, then this is the text that we'll infer in its place. (E.g., for "L<Getopt::Std>", the inferred link text is "Getopt::Std"

  Third:
  : The name or URL, or `undef` if none. (E.g., in "L<Perl Functions|perlfunc>", the name (also sometimes called the page) is "perlfunc(1)" In "L</CAVEATS>", the name is `undef`.)

  Fourth:
  : The section (AKA "item" in older perlpods), or `undef` if none. E.g., in "L<Getopt::Std/DESCRIPTION>", "DESCRIPTION" is the section. (Note that this is not the same as a manpage section like the "5" in "man 5 crontab". "Section Foo" in the Pod sense means the part of the text that's introduced by the heading or item whose text is "Foo".)

  Pod parsers may also note additional attributes including:

  Fifth:
  : A flag for whether item 3 (if present) is a URL (like [dq]http://lists.perl.org[dq] is), in which case there should be no section attribute; a Pod name (like "perldoc(1)" and "Getopt::Std" are); or possibly a man page name (like "**crontab(5)** " is).

  Sixth:
  : The raw original L<...> content, before text is split on "|", "/", etc, and before E<...> codes are expanded.

  (The above were numbered only for concise reference below. It is not a requirement that these be passed as an actual list or array.)

  For example:

```
L<Foo::Bar>
  =>  undef,                          # link text
      "Foo::Bar",                     # possibly inferred link text
      "Foo::Bar",                     # name
      undef,                          # section
      'pod',                          # what sort of link
      "Foo::Bar"                      # original content

L<Perlport's section on NL's|perlport/Newlines>
  =>  "Perlport's section on NL's",   # link text
      "Perlport's section on NL's",   # possibly inferred link text
```

```
          "perlport",                    # name
          "Newlines",                    # section
          'pod',                         # what sort of link
          "Perlport's section on NL's|perlport/Newlines"
                                         # original content

  L<perlport/Newlines>
    =>  undef,                           # link text
          '"Newlines" in perlport',      # possibly inferred link text
          "perlport",                    # name
          "Newlines",                    # section
          'pod',                         # what sort of link
          "perlport/Newlines"            # original content

  L<crontab(5)/"DESCRIPTION">
    =>  undef,                           # link text
          '"DESCRIPTION" in crontab(5)',
# possibly inferred link text
          "crontab(5)",                  # name
          "DESCRIPTION",                 # section
          'man',                         # what sort of link
          'DESCRIPTION"" -- crontab(5)'/
# original content

  L</Object Attributes>
    =>  undef,                           # link text
          '"Object Attributes"',         # possibly inferred link text
          undef,                         # name
          "Object Attributes",           # section
          'pod',                         # what sort of link
          "/Object Attributes"           # original content

  L<http://www.perl.org/>
    =>  undef,                           # link text
          "http://www.perl.org/",
         # possibly inferred link text
          "http://www.perl.org/",
         # name
          undef,                         # section
          'url',                         # what sort of link
          "http://www.perl.org/"
          # original content

  L<Perl.org|http://www.perl.org/>
    =>  "Perl.org",                      # link text
          "http://www.perl.org/",
         # possibly inferred link text
          "http://www.perl.org/",
         # name
          undef,                         # section
          'url',                         # what sort of link
          "Perl.org|http://www.perl.org/"
# original content
```

Note that you can distinguish URL-links from anything else by the fact that they match

`m/\A\w+:[^:\s]\S*\z/`. So `L<http://www.perl.com>` is a URL, but `L<HTTP::Response>` isn't.

- In case of L<...> codes with no "text|" part in them, older formatters have exhibited great variation in actually displaying the link or cross reference. For example, L<**crontab**(5)> would render as "the `crontab(5)` manpage", or "in the `crontab(5)` manpage" or just "`." -- crontab(5)`

  Pod processors must now treat "text|"–less links as follows:

  ```
  L<name>          =>  L<name|name>
  L</section>      =>  L<"section"|/section>
  L<name/section> =>  L<"section" in name|name/section>
  ```

- Note that section names might contain markup. I.e., if a section starts with:

  ```
  =head2 About the C<-M> Operator
  ```

  or with:

  ```
  =item About the C<-M> Operator
  ```

  then a link to it would look like this:

  ```
  L<somedoc/About the C<-M> Operator>
  ```

  Formatters may choose to ignore the markup for purposes of resolving the link and use only the renderable characters in the section name, as in:

  ```
  <h1><a name="About_the_-M_Operator">About the <code>-M</code>
  Operator</h1>

  ...

  <a href="somedoc#About_the_-M_Operator">About the <code>-M</code>
  Operator" in somedoc</a>
  ```

- Previous versions of perlpod(1) distinguished `L<name/"section">` links from `L<name/item>` links (and their targets). These have been merged syntactically and semantically in the current specification, and *section* can refer either to a "=head*n* Heading Content" command or to a "=item Item Content" command. This specification does not specify what behavior should be in the case of a given document having several things all seeming to produce the same *section* identifier (e.g., in HTML, several things all producing the same *anchorname* in ... elements). Where Pod processors can control this behavior, they should use the first such anchor. That is, `L<Foo/Bar>` refers to the *first* "Bar" section in Foo.

  But for some processors/formats this cannot be easily controlled; as with the HTML example, the behavior of multiple ambiguous ... is most easily just left up to browsers to decide.

- In a `L<text|...>` code, text may contain formatting codes for formatting or for E<...> escapes, as in:

  ```
  L<B<ummE<234>stuff>|...>
  ```

  For `L<...>` codes without a "name|" part, only `E<...>` and `Z<>` codes may occur. That is, authors should not use "`L<B<Foo::Bar>>`".

  Note, however, that formatting codes and Z<>'s can occur in any and all parts of an L<...> (i.e., in *name*, *section*, *text*, and *url*).

  Authors must not nest L<...> codes. For example, "L<The L<Foo::Bar> man page>" should be treated as an error.

- Note that Pod authors may use formatting codes inside the "text" part of "L<text|name>" (and so on for L<text|/"sec">).

  In other words, this is valid:

  ```
  Go read L<the docs on C<$.>|perlvar/"$.">
  ```

  Some output formats that do allow rendering "L<...>" codes as hypertext, might not allow the link-text to be formatted; in that case, formatters will have to just ignore that formatting.

- At time of writing, `L<name>` values are of two types: either the name of a Pod page like `L<Foo::Bar>` (which might be a real Perl module or program in an `@INC` / PATH directory, or a .pod file in those places); or the name of a Unix man page, like `L<crontab(5)>`. In theory, `L<chmod>` is ambiguous between a Pod page called "chmod", or the Unix man page "chmod" (in whatever man-section). However, the presence of a string in parens, as in "**crontab(5)**", is sufficient to signal that what is being discussed is not a Pod page, and so is presumably a Unix man page. The distinction is of no importance to many Pod processors, but some processors that render to hypertext formats may need to distinguish them in order to know how to render a given `L<foo>` code.

- Previous versions of perlpod(1) allowed for a `L<section>` syntax (as in `L<Object Attributes>`), which was not easily distinguishable from `L<name>` syntax and for `L<"section">` which was only slightly less ambiguous. This syntax is no longer in the specification, and has been replaced by the `L</section>` syntax (where the slash was formerly optional). Pod parsers should tolerate the `L<"section">` syntax, for a while at least. The suggested heuristic for distinguishing `L<section>` from `L<name>` is that if it contains any whitespace, it's a *section*. Pod processors should warn about this being deprecated syntax.

About =over...=back Regions

"=over"..."=back" regions are used for various kinds of list-like structures. (I use the term "region" here simply as a collective term for everything from the "=over" to the matching "=back".)

- The non-zero numeric *indentlevel* in "=over *indentlevel*" ... "=back" is used for giving the formatter a clue as to how many "spaces" (ems, or roughly equivalent units) it should tab over, although many formatters will have to convert this to an absolute measurement that may not exactly match with the size of spaces (or M's) in the document's base font. Other formatters may have to completely ignore the number. The lack of any explicit *indentlevel* parameter is equivalent to an *indentlevel* value of 4. Pod processors may complain if *indentlevel* is present but is not a positive number matching `m/\A(\d*\.)?\d+\z/`.

- Authors of Pod formatters are reminded that "=over" ... "=back" may map to several different constructs in your output format. For example, in converting Pod to (X)HTML, it can map to any of <ul>..., <ol>..., <dl>...</dl>, or <blockquote>...</blockquote>. Similarly, "=item" can map to <li> or <dt>.

- Each "=over" ... "=back" region should be one of the following:

  - An "=over" ... "=back" region containing only "=item *" commands, each followed by some number of ordinary/verbatim paragraphs, other nested "=over" ... "=back" regions, "=for..." paragraphs, and "=begin"..."=end" regions.

    (Pod processors must tolerate a bare "=item" as if it were "=item *".) Whether "*" is rendered as a literal asterisk, an "o", or as some kind of real bullet character, is left up to the Pod formatter, and may depend on the level of nesting.

  - An "=over" ... "=back" region containing only `m/\A=item\s+\d+\.?\s*\z/` paragraphs, each one (or each group of them) followed by some number of ordinary/verbatim paragraphs, other nested "=over" ... "=back" regions, "=for..." paragraphs, and/or "=begin"..."=end" codes. Note that the numbers must start at 1 in each section, and must proceed in order and without skipping numbers.

    (Pod processors must tolerate lines like "=item 1" as if they were "=item 1.", with the period.)

- An ‘‘=over’’ ... ‘‘=back’’ region containing only ‘‘=item [text]’’ commands, each one (or each group of them) followed by some number of ordinary/verbatim paragraphs, other nested ‘‘=over’’ ... ‘‘=back’’ regions, or ‘‘=for...’’ paragraphs, and ‘‘=begin’’...‘‘=end’’ regions.

  The ‘‘=item [text]’’ paragraph should not match `m/\A=item\s+\d+\.?\s*\z/` or `m/\A=item\s+\*\s*\z/`, nor should it match just `m/\A=item\s*\z/`.

- An ‘‘=over’’ ... ‘‘=back’’ region containing no ‘‘=item’’ paragraphs at all, and containing only some number of ordinary/verbatim paragraphs, and possibly also some nested ‘‘=over’’ ... ‘‘=back’’ regions, ‘‘=for...’’ paragraphs, and ‘‘=begin’’...‘‘=end’’ regions. Such an itemless ‘‘=over’’ ... ‘‘=back’’ region in Pod is equivalent in meaning to a ‘‘<blockquote>...</blockquote>’’ element in HTML.

Note that with all the above cases, you can determine which type of ‘‘=over’’ ... ‘‘=back’’ you have, by examining the first (non−‘‘=cut’’, non−‘‘=pod’’) Pod paragraph after the ‘‘=over’’ command.

- Pod formatters *must* tolerate arbitrarily large amounts of text in the "=item *text...*" paragraph. In practice, most such paragraphs are short, as in:

```
=item For cutting off our trade with all parts of the world
```

  But they may be arbitrarily long:

```
=item For transporting us beyond seas to be tried for pretended
offenses

=item He is at this time transporting large armies of foreign
mercenaries to complete the works of death, desolation and
tyranny, already begun with circumstances of cruelty and perfidy
scarcely paralleled in the most barbarous ages, and totally
unworthy the head of a civilized nation.
```

- Pod processors should tolerate ‘‘=item *’’ / "=item *number*" commands with no accompanying paragraph. The middle item is an example:

```
=over

=item 1

Pick up dry cleaning.

=item 2

=item 3

Stop by the store.  Get Abba Zabas, Stoli, and cheap lawn chairs.

=back
```

- No ‘‘=over’’ ... ‘‘=back’’ region can contain headings. Processors may treat such a heading as an error.
- Note that an ‘‘=over’’ ... ‘‘=back’’ region should have some content. That is, authors should not have an empty region like this:

```
=over

=back
```

  Pod processors seeing such a contentless ‘‘=over’’ ... ‘‘=back’’ region, may ignore it, or may report it as an error.

- Processors must tolerate an ‘‘=over’’ list that goes off the end of the document (i.e., which has no matching ‘‘=back’’), but they may warn about such a list.
- Authors of Pod formatters should note that this construct:

```
=item Neque

=item Porro

=item Quisquam Est

Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
velit, sed quia non numquam eius modi tempora incidunt ut
labore et dolore magnam aliquam quaerat voluptatem.

=item Ut Enim
```

  is semantically ambiguous, in a way that makes formatting decisions a bit difficult. On the one hand, it could be mention of an item ‘‘Neque’’, mention of another item ‘‘Porro’’, and mention of another item ‘‘Quisquam Est’’, with just the last one requiring the explanatory paragraph ‘‘Qui dolorem ipsum quia dolor...’’; and then an item ‘‘Ut Enim’’. In that case, you’d want to format it like so:

```
Neque

Porro

Quisquam Est
  Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
  velit, sed quia non numquam eius modi tempora incidunt ut
  labore et dolore magnam aliquam quaerat voluptatem.

Ut Enim
```

  But it could equally well be a discussion of three (related or equivalent) items, ‘‘Neque’’, ‘‘Porro’’, and ‘‘Quisquam Est’’, followed by a paragraph explaining them all, and then a new item ‘‘Ut Enim’’. In that case, you’d probably want to format it like so:

```
Neque
Porro
Quisquam Est
  Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
  velit, sed quia non numquam eius modi tempora incidunt ut
  labore et dolore magnam aliquam quaerat voluptatem.

Ut Enim
```

  But (for the foreseeable future), Pod does not provide any way for Pod authors to distinguish which grouping is meant by the above ‘‘=item’’–cluster structure. So formatters should format it like so:

```
Neque

Porro

Quisquam Est

  Qui dolorem ipsum quia dolor sit amet, consectetur, adipisci
  velit, sed quia non numquam eius modi tempora incidunt ut
  labore et dolore magnam aliquam quaerat voluptatem.
```

```
Ut Enim
```

That is, there should be (at least roughly) equal spacing between items as between paragraphs (although that spacing may well be less than the full height of a line of text). This leaves it to the reader to use (con)textual cues to figure out whether the ‘‘Qui dolorem ipsum...’’ paragraph applies to the ‘‘Quisquam Est’’ item or to all three items ‘‘Neque’’, ‘‘Porro’’, and ‘‘Quisquam Est’’. While not an ideal situation, this is preferable to providing formatting cues that may be actually contrary to the author’s intent.

About Data Paragraphs and “=begin/=end” Regions

Data paragraphs are typically used for inlining non-Pod data that is to be used (typically passed through) when rendering the document to a specific format:

```
=begin rtf

\par{\pard\qr\sa4500{\i Printed\~\chdate\~\chtime}\par}

=end rtf
```

The exact same effect could, incidentally, be achieved with a single ‘‘=for’’ paragraph:

```
=for rtf \par{\pard\qr\sa4500{\i Printed\~\chdate\~\chtime}\par}
```

(Although that is not formally a data paragraph, it has the same meaning as one, and Pod parsers may parse it as one.)

Another example of a data paragraph:

```
=begin html

I like <em>PIE</em>!

<hr>Especially pecan pie!

=end html
```

If these were ordinary paragraphs, the Pod parser would try to expand the ‘‘E’’ (in the first paragraph) as a formatting code, just like ‘‘E<lt>’’ or ‘‘E<eacute>’’. But since this is in a "=begin *identifier*‘‘...’’=end *identifier*" region *and* the identifier ‘‘html’’ doesn’t begin have a ‘‘:’’ prefix, the contents of this region are stored as data paragraphs, instead of being processed as ordinary paragraphs (or if they began with a spaces and/or tabs, as verbatim paragraphs).

As a further example: At time of writing, no ‘‘biblio’’ identifier is supported, but suppose some processor were written to recognize it as a way of (say) denoting a bibliographic reference (necessarily containing formatting codes in ordinary paragraphs). The fact that ‘‘biblio’’ paragraphs were meant for ordinary processing would be indicated by prefacing each ‘‘biblio’’ identifier with a colon:

```
=begin :biblio

Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.

=end :biblio
```

This would signal to the parser that paragraphs in this begin...end region are subject to normal handling as ordinary/verbatim paragraphs (while still tagged as meant only for processors that understand the ‘‘biblio’’ identifier). The same effect could be had with:

```
=for :biblio
Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.
```

The ‘‘:’’ on these identifiers means simply ‘‘process this stuff normally, even though the result will be for

some special target''. I suggest that parser APIs report ``biblio'' as the target identifier, but also report that it had a ``:'' prefix. (And similarly, with the above ``html'', report ``html'' as the target identifier, and note the *lack* of a ``:'' prefix.)

Note that a "=begin *identifier*``...''=end *identifier*" region where *identifier* begins with a colon, *can* contain commands. For example:

```
=begin :biblio

Wirth's classic is available in several editions, including:

=for comment
 hm, check abebooks.com for how much used copies cost.

=over

=item

Wirth, Niklaus.  1975.  I<Algorithmen und Datenstrukturen.>
Teubner, Stuttgart.  [Yes, it's in German.]

=item

Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.

=back

=end :biblio
```

Note, however, a "=begin *identifier*``...''=end *identifier*" region where *identifier* does *not* begin with a colon, should not directly contain ``=head1'' ... ``=head4'' commands, nor ``=over'', nor ``=back'', nor ``=item''. For example, this may be considered invalid:

```
=begin somedata

This is a data paragraph.

=head1 Don't do this!

This is a data paragraph too.

=end somedata
```

A Pod processor may signal that the above (specifically the ``=head1'' paragraph) is an error. Note, however, that the following should *not* be treated as an error:

```
=begin somedata

This is a data paragraph.

=cut

# Yup, this isn't Pod anymore.
sub excl { (rand() > .5) ? "hoo!" : "hah!" }

=pod
```

```
  This is a data paragraph too.

  =end somedata
```

And this too is valid:

```
  =begin someformat

  This is a data paragraph.

    And this is a data paragraph.

  =begin someotherformat

  This is a data paragraph too.

    And this is a data paragraph too.

  =begin :yetanotherformat

  =head2 This is a command paragraph!

  This is an ordinary paragraph!

    And this is a verbatim paragraph!

  =end :yetanotherformat

  =end someotherformat

  Another data paragraph!

  =end someformat
```

The contents of the above ‘‘=begin :yetanotherformat’’ ... ‘‘=end :yetanotherformat’’ region *aren't* data paragraphs, because the immediately containing region's identifier (‘‘:yetanotherformat’’) begins with a colon. In practice, most regions that contain data paragraphs will contain *only* data paragraphs; however, the above nesting is syntactically valid as Pod, even if it is rare. However, the handlers for some formats, like ‘‘html’’, will accept only data paragraphs, not nested regions; and they may complain if they see (targeted for them) nested regions, or commands, other than ‘‘=end’’, ‘‘=pod’’, and ‘‘=cut’’.

Also consider this valid structure:

```
  =begin :biblio

  Wirth's classic is available in several editions, including:

  =over

  =item

  Wirth, Niklaus.  1975.  I<Algorithmen und Datenstrukturen.>
  Teubner, Stuttgart.  [Yes, it's in German.]

  =item

  Wirth, Niklaus.  1976.  I<Algorithms + Data Structures =
```

```
Programs.>  Prentice-Hall, Englewood Cliffs, NJ.

=back

Buy buy buy!

=begin html

<img src='wirth_spokesmodeling_book.png'>

<hr>

=end html

Now now now!

=end :biblio
```

There, the ‘‘=begin html’’...‘‘=end html’’ region is nested inside the larger ‘‘=begin :biblio’’...‘‘=end :biblio’’ region. Note that the content of the ‘‘=begin html’’...‘‘=end html’’ region is data paragraph(s), because the immediately containing region’s identifier (‘‘html’’) *doesn’t* begin with a colon.

Pod parsers, when processing a series of data paragraphs one after another (within a single region), should consider them to be one large data paragraph that happens to contain blank lines. So the content of the above ‘‘=begin html’’...‘‘=end html’’ *may* be stored as two data paragraphs (one consisting of ‘‘<img src=’wirth_spokesmodeling_book.png’>\n’’ and another consisting of ‘‘<hr>\n’’), but *should* be stored as a single data paragraph (consisting of ‘‘<img src=’wirth_spokesmodeling_book.png’>\n\n<hr>\n’’).

Pod processors should tolerate empty "=begin *something*‘‘...’’=end *something*‘‘ regions, empty ’’=begin :*something*‘‘...’’=end :*something*‘‘ regions, and contentless ’’=for *something*‘‘ and ’’=for :*something*" paragraphs. I.e., these should be tolerated:

```
=for html

=begin html

=end html

=begin :biblio

=end :biblio
```

Incidentally, note that there’s no easy way to express a data paragraph starting with something that looks like a command. Consider:

```
=begin stuff

=shazbot

=end stuff
```

There, ‘‘=shazbot’’ will be parsed as a Pod command ‘‘shazbot’’, not as a data paragraph ‘‘=shazbot\n’’. However, you can express a data paragraph consisting of ‘‘=shazbot\n’’ using this code:

```
=for stuff =shazbot
```

The situation where this is necessary, is presumably quite rare.

Note that =end commands must match the currently open =begin command. That is, they must properly nest. For example, this is valid:

```
=begin outer

X

=begin inner

Y

=end inner

Z

=end outer
```

while this is invalid:

```
=begin outer

X

=begin inner

Y

=end outer

Z

=end inner
```

This latter is improper because when the ‘‘=end outer’’ command is seen, the currently open region has the formatname ‘‘inner’’, not ‘‘outer’’. (It just happens that ‘‘outer’’ is the format name of a higher-up region.) This is an error. Processors must by default report this as an error, and may halt processing the document containing that error. A corollary of this is that regions cannot ‘‘overlap’’. That is, the latter block above does not represent a region called ‘‘outer’’ which contains X and Y, overlapping a region called ‘‘inner’’ which contains Y and Z. But because it is invalid (as all apparently overlapping regions would be), it doesn’t represent that, or anything at all.

Similarly, this is invalid:

```
=begin thing

=end hting
```

This is an error because the region is opened by ‘‘thing’’, and the ‘‘=end’’ tries to close ‘‘hting’’ [sic].

This is also invalid:

```
=begin thing

=end
```

This is invalid because every ‘‘=end’’ command must have a formatname parameter.

SEE ALSO

perlpod(1), ‘‘PODs: Embedded Documentation’’ in perlsyn(1), podchecker

AUTHOR

Sean M. Burke