

NAME

Pod::Simple::Subclassing -- write a formatter as a Pod::Simple subclass

SYNOPSIS

```
package Pod::SomeFormatter;
use Pod::Simple;
@ISA = qw(Pod::Simple);
$VERSION = '1.01';
use strict;

sub _handle_element_start {
my($parser, $element_name, $attr_hash_r) = @_;
...
}

sub _handle_element_end {
my($parser, $element_name, $attr_hash_r) = @_;
# NOTE: $attr_hash_r is only present when $element_name is "over" or "begin"
# The remaining code excerpts will mostly ignore this $attr_hash_r, as it is
# mostly useless. It is documented where "over-*" and "begin" events are
# documented.
...
}

sub _handle_text {
my($parser, $text) = @_;
...
}
1;
```

DESCRIPTION

This document is about using [Pod::Simple](#) to write a Pod processor, generally a Pod formatter. If you just want to know about using an existing Pod formatter, instead see its documentation and see also the docs in Pod::Simple.

The zeroeth step in writing a Pod formatter is to make sure that there isn't already a decent one in CPAN. See <http://search.cpan.org/>, and run a search on the name of the format you want to render to. Also consider joining the Pod People list <http://lists.perl.org/showlist.cgi?name=pod-people> and asking whether anyone has a formatter for that format — maybe someone cobbled one together but just hasn't released it.

The first step in writing a Pod processor is to read [perlpodspec\(1\)](#), which contains information on writing a Pod parser (which has been largely taken care of by Pod::Simple), but also a lot of requirements and recommendations for writing a formatter.

The second step is to actually learn the format you're planning to format to — or at least as much as you need to know to represent Pod, which probably isn't much.

The third step is to pick which of Pod::Simple's interfaces you want to use:

Pod::Simple

The basic [Pod::Simple](#) interface that uses `_handle_element_start()`, `_handle_element_end()` and `_handle_text()`.

Pod::Simple::Methody

The [Pod::Simple::Methody](#) interface is event-based, similar to that of [HTML::Parser](#) or [XML::Parser](#)'s "Handlers".

Pod::Simple::PullParser

[Pod::Simple::PullParser](#) provides a token-stream interface, sort of like `HTML::TokeParser`'s interface.

Pod::Simple::SimpleTree

[Pod::Simple::SimpleTree](#) provides a simple tree interface, rather like `XML::Parser`'s "Tree" interface. Users familiar with XML handling will be comfortable with this interface. Users interested in outputting XML, should look into the modules that produce an XML representation of the Pod stream, notably `Pod::Simple::XMLOutputStream`; you can feed the output of such a class to whatever XML parsing system you are most at home with.

The last step is to write your code based on how the events (or tokens, or tree-nodes, or the XML, or however you're parsing) will map to constructs in the output format. Also be sure to consider how to escape text nodes containing arbitrary text, and what to do with text nodes that represent preformatted text (from verbatim sections).

Events

TODO intro... mention that events are supplied for implicits, like for missing >'s

In the following section, we use XML to represent the event structure associated with a particular construct. That is, TODO

```
$parser->_handle_element_start( element_name, attr_hashref )
$parser->_handle_element_end( element_name )
$parser->_handle_text( text_string )
```

TODO describe

events with an `element_name` of Document

Parsing a document produces this event structure:

```
<Document start_line="543">
...all events...
</Document>
```

The value of the `start_line` attribute will be the line number of the first Pod directive in the document.

If there is no Pod in the given document, then the event structure will be this:

```
<Document contentless="1" start_line="543">
</Document>
```

In that case, the value of the `start_line` attribute will not be meaningful; under current implementations, it will probably be the line number of the last line in the file.

events with an `element_name` of Para

Parsing a plain (non-verbatim, non-directive, non-data) paragraph in a Pod document produces this event structure:

```
<Para start_line="543">
...all events in this paragraph...
</Para>
```

The value of the `start_line` attribute will be the line number of the start of the paragraph.

For example, parsing this paragraph of Pod:

```
The value of the I<start_line> attribute will be the
line number of the start of the paragraph.
```

produces this event structure:

```
<Para start_line="129">
The value of the
<I>
start_line
</I>
attribute will be the line number of the first Pod directive
in the document.
</Para>
```

events with an `element_name` of B, C, F, or I.

Parsing a `B<...>` formatting code (or of course any of its semantically identical syntactic variants `B<<...>`, or `B<<<<...>>>>`, etc.) produces this event structure:

```
<B>
...stuff...
</B>
```

Currently, there are no attributes conveyed.

Parsing C, F, or I codes produce the same structure, with only a different element name.

If your parser object has been set to accept other formatting codes, then they will be presented like these B/C/F/I codes — i.e., without any attributes.

events with an `element_name` of S

Normally, parsing an `S<...>` sequence produces this event structure, just as if it were a B/C/F/I code:

```
<S>
...stuff...
</S>
```

However, [Pod::Simple](#) (and presumably all derived parsers) offers the `nbsp_for_S` option which, if enabled, will suppress all S events, and instead change all spaces in the content to non-breaking spaces. This is intended for formatters that output to a format that has no code that means the same as `S<...>`, but which has a code/character that means non-breaking space.

events with an `element_name` of X

Normally, parsing an `X<...>` sequence produces this event structure, just as if it were a B/C/F/I code:

```
<X>
...stuff...
</X>
```

However, [Pod::Simple](#) (and presumably all derived parsers) offers the `nix_X_codes` option which, if enabled, will suppress all X events and ignore their content. For formatters/processors that don't use X events, this is presumably quite useful.

events with an `element_name` of L

Because the `L<...>` is the most complex construct in the language, it should not surprise you that the events it generates are the most complex in the language. Most of complexity is hidden away in the attribute values, so for those of you writing a Pod formatter that produces a non-hypertextual format, you can just ignore the attributes and treat an L event structure like a formatting element that (presumably) doesn't actually produce a change in formatting. That is, the content of the L event structure (as opposed to its attributes) is always what text should be displayed.

There are, at first glance, three kinds of L links: URL, man, and pod.

When a `L<some_url>` code is parsed, it produces this event structure:

```
<L content-implicit="yes" raw="that_url" to="that_url" type="url">
that_url
</L>
```

The `type="url"` attribute is always specified for this type of L code.

For example, this Pod source:

```
L<http://www.perl.com/CPAN/authors/>
```

produces this event structure:

```
<L content-implicit="yes" raw="" -- http://www.perl.com/CPAN/authors/ -P
to="" -- http://www.perl.com/CPAN/authors/ -P
type="url">
  http://www.perl.com/CPAN/authors/
</L>
```

When a `L<manpage(section)>` code is parsed (and these are fairly rare and not terribly useful), it produces this event structure:

```
<L content-implicit="yes" raw="manpage(section)" to="manpage(section)" type="man">
manpage(section)
</L>
```

The `type="man"` attribute is always specified for this type of L code.

For example, this Pod source:

```
L<crontab(5)>
```

produces this event structure:

```
<L content-implicit="yes" raw="crontab(5)" to="crontab(5)" type="man">
crontab(5)
</L>
```

In the rare cases where a man page link has a section specified, that text appears in a `section` attribute.

For example, this Pod source:

```
L<crontab(5) / "ENVIRONMENT">
```

will produce this event structure:

```
<L content-implicit="yes" raw="crontab(5) / "ENVIRONMENT" section="ENVIRONMENT" in crontab(5)
"ENVIRONMENT" in crontab(5)
</L>
```

In the rare case where the Pod document has code like `L<some text|manpage(section)>`, then the `some text` will appear as the content of the element, the `manpage(section)` text will appear only as the value of the `to` attribute, and there will be no `content-implicit="yes"` attribute (whose presence means that the Pod parser had to infer what text should appear as the link text — as opposed to cases where that attribute is absent, which means that the Pod parser did *not* have to infer the link text, because that L code explicitly specified some link text.)

For example, this Pod source:

```
L<hell itself!|crontab(5)>
```

will produce this event structure:

```
<L raw="hell itself!|crontab(5)" to="crontab(5)" type="man">
hell itself!
</L>
```

The last type of L structure is for links to/within Pod documents. It is the most complex because it can have a *to* attribute, or a *section* attribute, or both. The `type="pod"` attribute is always specified for this type of L code.

In the most common case, the simple case of a L<podpage> code produces this event structure:

```
<L content-implicit="yes" raw="podpage" to="podpage" type="pod">
podpage
</L>
```

For example, this Pod source:

```
L<Net::Ping>
```

produces this event structure:

```
<L content-implicit="yes" raw="Net::Ping" to="Net::Ping" type="pod">
Net::Ping
</L>
```

In cases where there is link-text explicitly specified, it is to be found in the content of the element (and not the attributes), just as with the L<*sometext*|*manpage(section)*> case discussed above. For example, this Pod source:

```
L<Perl Error Messages|perldiag>
```

produces this event structure:

```
<L raw="Perl Error Messages|perldiag" to="perldiag" type="pod">
Perl Error Messages
</L>
```

In cases of links to a section in the current Pod document, there is a *section* attribute instead of a *to* attribute. For example, this Pod source:

```
L</"Member Data">
```

produces this event structure:

```
<L content-implicit="yes" raw="/&quot;Member Data&quot;" section="Member Data"
"Member Data"
</L>
```

As another example, this Pod source:

```
L<the various attributes|/"Member Data">
```

produces this event structure:

```
<L raw="the various attributes|/"Member Data&quot;" section="Member Data"
the various attributes
</L>
```

In cases of links to a section in a different Pod document, there are both a *section* attribute and a *to* attribute. For example, this Pod source:

```
L<perlsyn/"Basic BLOCKs and Switch Statements">
```

produces this event structure:

```
<L content-implicit="yes" raw="perlsyn/&quot;Basic BLOCKs and Switch Statemen
"Basic BLOCKs and Switch Statements" in perlsyn
</L>
```

As another example, this Pod source:

```
L<SWITCH statements|perlsyn/"Basic BLOCKs and Switch Statements">
```

produces this event structure:

```
<L raw="SWITCH statements|perlsyn/&quot;Basic BLOCKs and Switch Statements&quot;
SWITCH statements
</L>
```

Incidentally, note that we do not distinguish between these syntaxes:

```
L/"Member Data">
L<"Member Data">
L</Member Data>
L<Member Data> [deprecated syntax]
```

That is, they all produce the same event structure (for the most part), namely:

```
<L content-implicit="yes" raw="$depends_on_syntax" section="Member Data" type
&#34;Member Data&#34;
</L>
```

The *raw* attribute depends on what the raw content of the L<> is, so that is why the event structure is the same “for the most part”.

If you have not guessed it yet, the *raw* attribute contains the raw, original, unescaped content of the L<> formatting code. In addition to the examples above, take notice of the following event structure produced by the following L<> formatting code.

```
L<click B<here>|page/About the C<-M> switch>

<L raw="click B<here>|page/About the C<-M> switch" section="About the -M swit
click B<here>
</L>
```

Specifically, notice that the formatting codes are present and unescaped in *raw*.

There is a known bug in the *raw* attribute where any surrounding whitespace is condensed into a single ' '. For example, given L< link>, *raw* will be “ link”.

events with an *element_name* of E or Z

While there are Pod codes E<...> and Z<>, these *do not* produce any E or Z events — that is, there are no such events as E or Z.

events with an *element_name* of Verbatim

When a Pod verbatim paragraph (AKA “codeblock”) is parsed, it produces this event structure:

```
<Verbatim start_line="543" xml:space="preserve">
...text...
</Verbatim>
```

The value of the *start_line* attribute will be the line number of the first line of this verbatim block. The *xml:space* attribute is always present, and always has the value “preserve”.

The text content will have tabs already expanded.

events with an *element_name* of head1 .. head4

When a “=head1 ...” directive is parsed, it produces this event structure:

```
<head1>
...stuff...
</head1>
```

For example, a directive consisting of this:

```
=head1 Options to C<new> et al.
```

will produce this event structure:

```
<head1 start_line="543">
Options to
<C>
new
</C>
et al.
</head1>
```

“=head2” through “=head4” directives are the same, except for the element names in the event structure.

events with an `element_name` of encoding

In the default case, the events corresponding to `=encoding` directives are not emitted. They are emitted if `keep_encoding_directive` is true. In that case they produce event structures like “events with an `element_name` of head1 .. head4” above.

events with an `element_name` of over-bullet

When an “=over ... =back” block is parsed where the items are a bulleted list, it will produce this event structure:

```
<over-bullet indent="4" start_line="543">
<item-bullet start_line="545">
...Stuff...
</item-bullet>
...more item-bullets...
</over-bullet fake-closer="1">
```

The attribute *fake-closer* is only present if it is a true value; it is not present if it is a false value. It is shown in the above example to illustrate where the attribute is (in the **closing** tag). It signifies that the `=over` did not have a matching `=back`, and thus [Pod::Simple](#) had to create a fake closer.

For example, this Pod source:

```
=over

=item *

Something

=back
```

Would produce an event structure that does **not** have the *fake-closer* attribute, whereas this Pod source:

```
=over

=item *

Gasp! An unclosed =over block!
```

would. The rest of the over-* examples will not demonstrate this attribute, but they all can have it. See [Pod::Checker](#)’s source for an example of this attribute being used.

The value of the *indent* attribute is whatever value is after the “=over” directive, as in “=over 8”. If no such value is specified in the directive, then the *indent* attribute has the value “4”.

For example, this Pod source:

```
=over
=item *
Stuff
=item *
Bar I<baz>!
=back
```

produces this event structure:

```
<over-bullet indent="4" start_line="10">
<item-bullet start_line="12">
Stuff
</item-bullet>
<item-bullet start_line="14">
Bar <I>baz</I>!
</item-bullet>
</over-bullet>
```

events with an `element_name` of `over-number`

When an “=over ... =back” block is parsed where the items are a numbered list, it will produce this event structure:

```
<over-number indent="4" start_line="543">
<item-number number="1" start_line="545">
...Stuff...
</item-number>
...more item-number...
</over-bullet>
```

This is like the “over-bullet” event structure; but note that the contents are “item-number” instead of “item-bullet”, and note that they will have a “number” attribute, which some formatters/processors may ignore (since, for example, there’s no need for it in HTML when producing an “......” structure), but which any processor may use.

Note that the values for the *number* attributes of “item-number” elements in a given “over-number” area *will* start at 1 and go up by one each time. If the Pod source doesn’t follow that order (even though it really should!), whatever numbers it has will be ignored (with the correct values being put in the *number* attributes), and an error message might be issued to the user.

events with an `element_name` of `over-text`

These events are somewhat unlike the other over-* structures, as far as what their contents are. When an “=over ... =back” block is parsed where the items are a list of text “subheadings”, it will produce this event structure:

```
<over-text indent="4" start_line="543">
<item-text>
...stuff...
</item-text>
...stuff (generally Para or Verbatim elements)...
<item-text>
...more item-text and/or stuff...
</over-text>
```

The *indent* and *fake-closer* attributes are as with the other over-* events.

For example, this Pod source:

```
=over

=item Foo

Stuff

=item Bar I<baz>!

Quux

=back
```

produces this event structure:

```
<over-text indent="4" start_line="20">
<item-text start_line="22">
Foo
</item-text>
<Para start_line="24">
Stuff
</Para>
<item-text start_line="26">
Bar
<I>
baz
</I>
!
</item-text>
<Para start_line="28">
Quux
</Para>
</over-text>
```

events with an `element_name` of `over-block`

These events are somewhat unlike the other `over-*` structures, as far as what their contents are. When an “`=over ... =back`” block is parsed where there are no items, it will produce this event structure:

```
<over-block indent="4" start_line="543">
...stuff (generally Para or Verbatim elements)...
</over-block>
```

The *indent* and *fake-closer* attributes are as with the other `over-*` events.

For example, this Pod source:

```
=over

For cutting off our trade with all parts of the world

For transporting us beyond seas to be tried for pretended offenses

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
```

will produce this event structure:

```
<over-block indent="4" start_line="2">
<Para start_line="4">
For cutting off our trade with all parts of the world
</Para>
<Para start_line="6">
For transporting us beyond seas to be tried for pretended offenses
</Para>
<Para start_line="8">
He is at this time transporting large armies of [...more text...]
</Para>
</over-block>
```

events with an `element_name` of `over-empty`

Note: These events are only triggered if `parse_empty_lists()` is set to a true value.

These events are somewhat unlike the other `over-*` structures, as far as what their contents are. When an “`=over ... =back`” block is parsed where there is no content, it will produce this event structure:

```
<over-empty indent="4" start_line="543">
</over-empty>
```

The `indent` and `fake-closer` attributes are as with the other `over-*` events.

For example, this Pod source:

```
=over

=over

=back

=back
```

will produce this event structure:

```
<over-block indent="4" start_line="1">
<over-empty indent="4" start_line="3">
</over-empty>
</over-block>
```

Note that the outer `=over` is a block because it has no `=items` but still has content: the inner `=over`. The inner `=over`, in turn, is completely empty, and is treated as such.

events with an `element_name` of `item-bullet`

See “events with an `element_name` of `over-bullet`”, above.

events with an `element_name` of `item-number`

See “events with an `element_name` of `over-number`”, above.

events with an `element_name` of `item-text`

See “events with an `element_name` of `over-text`”, above.

events with an `element_name` of `for`

TODO...

events with an `element_name` of `Data`

TODO...

More Pod::Simple Methods

[Pod::Simple Methods](#) "Pod::Simple" provides a lot of methods that aren't generally interesting to the end user of an existing Pod formatter, but some of which you might find useful in writing a Pod formatter. They


```
I like S<Dutch apple pie>, don't you?
```

and treat it as if it were:

```
I like DutchE<nbsp>appleE<nbsp>pie, don't you?
```

This is handy for output formats that don't have anything quite like an “S<...>” code, but which do have a code for non-breaking space.

There is currently no method for going the other way; but I can probably provide one upon request.

```
$parser->version_report()
```

This returns a string reporting the \$VERSION value from your module (and its classname) as well as the \$VERSION value of Pod::Simple. Note that [perlpodspec\(1\)](#) requires output formats (wherever possible) to note this detail in a comment in the output format. For example, for some kind of SGML output format:

```
print OUT "<!-- \n", $parser->version_report, "\n -->";
```

```
$parser->pod_para_count()
```

This returns the count of Pod paragraphs seen so far.

```
$parser->line_count()
```

This is the current line number being parsed. But you might find the “line_number” event attribute more accurate, when it is present.

```
$parser->nix_X_codes( SOMEVALUE )
```

This attribute, when set to a true value (and it is false by default) ignores any “X<...>” sequences in the document being parsed. Many formats don't actually use the content of these codes, so have no reason to process them.

```
$parser->keep_encoding_directive( SOMEVALUE )
```

This attribute, when set to a true value (it is false by default) will keep =encoding and its content in the event structure. Most formats don't actually need to process the content of an =encoding directive, even when this directive sets the encoding and the processor makes use of the encoding information. Indeed, it is possible to know the encoding without processing the directive content.

```
$parser->merge_text( SOMEVALUE )
```

This attribute, when set to a true value (and it is false by default) makes sure that only one event (or token, or node) will be created for any single contiguous sequence of text. For example, consider this somewhat contrived example:

```
I just LOVE Z<>hotE<32>apple pie!
```

When that is parsed and events are about to be called on it, it may actually seem to be four different text events, one right after another: one event for “I just LOVE ”, one for “hot”, one for “ ”, and one for “apple pie!”. But if you have merge_text on, then you're guaranteed that it will be fired as one text event: “I just LOVE hot apple pie!”.

```
$parser->code_handler( CODE_REF )
```

This specifies code that should be called when a code line is seen (i.e., a line outside of the Pod). Normally this is undef, meaning that no code should be called. If you provide a routine, it should start out like this:

```
sub get_code_line { # or whatever you'll call it
my($line, $line_number, $parser) = @_;
...
}
```

Note, however, that sometimes the Pod events aren't processed in exactly the same order as the code lines are — i.e., if you have a file with Pod, then code, then more Pod, sometimes the code will be processed (via whatever you have code_handler call) before the all of the preceding Pod has been processed.

`$parser->cut_handler(CODE_REF)`

This is just like the `code_handler` attribute, except that it's for “=cut” lines, not code lines. The same caveats apply. “=cut” lines are unlikely to be interesting, but this is included for completeness.

`$parser->pod_handler(CODE_REF)`

This is just like the `code_handler` attribute, except that it's for “=pod” lines, not code lines. The same caveats apply. “=pod” lines are unlikely to be interesting, but this is included for completeness.

`$parser->whiteline_handler(CODE_REF)`

This is just like the `code_handler` attribute, except that it's for lines that are seemingly blank but have whitespace (“ ” and/or “\t”) on them, not code lines. The same caveats apply. These lines are unlikely to be interesting, but this is included for completeness.

`$parser->whine(linenumber, complaint string)`

This notes a problem in the Pod, which will be reported in the “Pod Errors” section of the document and/or sent to `STDERR`, depending on the values of the attributes `no_whining`, `no_errata_section`, and `complain_stderr`.

`$parser->scream(linenumber, complaint string)`

This notes an error like `whine` does, except that it is not suppressible with `no_whining`. This should be used only for very serious errors.

`$parser->source_dead(1)`

This aborts parsing of the current document, by switching on the flag that indicates that EOF has been seen. In particularly drastic cases, you might want to do this. It's rather nicer than just calling `die!`

`$parser->hide_line_numbers(SOMEVALUE)`

Some subclasses that indiscriminately dump event attributes (well, except for ones beginning with “”) can use this object attribute for refraining to dump the “start_line” attribute.

`$parser->no_whining(SOMEVALUE)`

This attribute, if set to true, will suppress reports of non-fatal error messages. The default value is false, meaning that complaints *are* reported. How they get reported depends on the values of the attributes `no_errata_section` and `complain_stderr`.

`$parser->no_errata_section(SOMEVALUE)`

This attribute, if set to true, will suppress generation of an errata section. The default value is false — i.e., an errata section will be generated.

`$parser->complain_stderr(SOMEVALUE)`

This attribute, if set to true will send complaints to `STDERR`. The default value is false — i.e., complaints do not go to `STDERR`.

`$parser->bare_output(SOMEVALUE)`

Some formatter subclasses use this as a flag for whether output should have prologue and epilogue code omitted. For example, setting this to true for an HTML formatter class should omit the “<html><head><title>...</title><body>...” prologue and the “</body></html>” epilogue.

If you want to set this to true, you should probably also set `no_whining` or at least `no_errata_section` to true.

`$parser->preserve_whitespace(SOMEVALUE)`

If you set this attribute to a true value, the parser will try to preserve whitespace in the output. This means that such formatting conventions as two spaces after periods will be preserved by the parser. This is primarily useful for output formats that treat whitespace as significant (such as text or `*roff`, but not HTML).

`$parser->parse_empty_lists(SOMEVALUE)`

If this attribute is set to true, the parser will not ignore empty `=over/=back` blocks. The type of `=over` will be *empty*, documented above, “events with an `element_name` of `over-empty`”.

SEE ALSO

[Pod::Simple](#) — event-based Pod-parsing framework

[Pod::Simple::Methody](#) — like [Pod::Simple](#), but each sort of event calls its own method (like `start_head3`)

[Pod::Simple::PullParser](#) — a Pod-parsing framework like [Pod::Simple](#), but with a token-stream interface

[Pod::Simple::SimpleTree](#) — a Pod-parsing framework like [Pod::Simple](#), but with a tree interface

[Pod::Simple::Checker](#) — a simple [Pod::Simple](#) subclass that reads documents, and then makes a plaintext report of any errors found in the document

[Pod::Simple::DumpAsXML](#) — for dumping Pod documents as tidily indented XML, showing each event on its own line

[Pod::Simple::XMLOutputStream](#) — dumps a Pod document as XML (without introducing extra whitespace as [Pod::Simple::DumpAsXML](#) does).

[Pod::Simple::DumpAsText](#) — for dumping Pod documents as tidily indented text, showing each event on its own line

[Pod::Simple::LinkSection](#) — class for objects representing the values of the `TODO` and `TODO` attributes of `L<...>` elements

[Pod::Escapes](#) — the module that [Pod::Simple](#) uses for evaluating `E<...>` content

[Pod::Simple::Text](#) — a simple plaintext formatter for Pod

[Pod::Simple::TextContent](#) — like [Pod::Simple::Text](#), but makes no effort for indent or wrap the text being formatted

[Pod::Simple::HTML](#) — a simple HTML formatter for Pod

[perlpod\(1\)](#)

[perlpodspec\(1\)](#)

[perldoc\(1\)](#)

SUPPORT

Questions or discussion about POD and [Pod::Simple](#) should be sent to the `pod-people@perl.org` mail list. Send an empty email to `pod-people-subscribe@perl.org` to subscribe.

This module is managed in an open GitHub repository, <<https://github.com/perl-pod/pod-simple/>>. Feel free to fork and contribute, or to clone <[git://github.com/perl-pod/pod-simple.git](https://github.com/perl-pod/pod-simple.git)> and send patches!

Patches against [Pod::Simple](#) are welcome. Please send bug reports to <bug-pod-simple@rt.cpan.org>.

COPYRIGHT AND DISCLAIMERS

Copyright (c) 2002 Sean M. Burke.

This library is free software; you can redistribute it and/or modify it under the same terms as Perl itself.

This program is distributed in the hope that it will be useful, but without any warranty; without even the implied warranty of merchantability or fitness for a particular purpose.

AUTHOR

[Pod::Simple](#) was created by Sean M. Burke <sburke@cpan.org>. But don't bother him, he's retired.

[Pod::Simple](#) is maintained by:

- Allison Randal allison@perl.org
- Hans Dieter Pearcey hdp@cpan.org
- David E. Wheeler dwheeler@cpan.org