NAME

Pod::Constants – Include constants from POD

SYNOPSIS

```
our ($myvar, $VERSION, @myarray, $html, %myhash);

use Pod::Constants -trim => 1,
    'Pod Section Name' => \$myvar,
    'Version' => sub { eval },
    'Some list' => \@myarray,
    html => \$html,
    'Some hash' => \%myhash;

=head2 Pod Section Name

This string will be loaded into $myvar

=head2 Version

# This is an example of using a closure.  $_ is set to the
# contents of the paragraph.  In this example, "eval" is
# used to execute this code at run time.
$VERSION = 0.19;

=head2 Some list

Each line from this section of the file
will be placed into a separate array element.
For example, this is $myarray[2].

=head2 Some hash

This text will not go into the hash, because
it doesn't look like a definition list.
    key1 => Some value (this will go into the hash)
    var2 => Some Other value (so will this)
    wtf = This won't make it in.

=head2 %myhash's value after the above:

   ( key1 => "Some value (this will go into the hash)",
     var2 => "Some Other value (so will this)"          )

=begin html <p>This text will be in $html</p>

=cut
```

DESCRIPTION

This module allows you to specify those constants that should be documented in your POD, and pull them out a run time in a fairly arbitrary fashion.

Pod::Constants uses Pod::Parser to do the parsing of the source file. It has to open the source file it is called from, and does so directly either by lookup in `%INC` or by assuming it is `$0` if the caller is "main" (or it can't find `%INC`{*caller()*})

ARBITARY DECISIONS

I have made this code only allow the "Pod Section Name" to match 'headN', 'item', 'for' and 'begin' POD sections. If you have a good reason why you think it should match other POD sections, drop me a line and if I'm convinced I'll put it in the standard version.

For 'for' and 'begin' sections, only the first word is counted as being a part of the specifier, as opposed to 'headN' and 'item', where the entire rest of the line counts.

FUNCTIONS

import(@args)

This function is called when we are "use'"d. It determines the source file by inspecting the value of *caller()* or `$0`.

The form of `@args` is HOOK => `$where`.

`$where` may be a scalar reference, in which case the contents of the POD section called "HOOK" will be loaded into `$where`.

`$where` may be an array reference, in which case the contents of the array will be the contents of the POD section called "HOOK", split into lines.

`$where` may be a hash reference, in which case any lines with a "=>" symbol present will have everything on the left have side of the => operator as keys and everything on the right as values. You do not need to quote either, nor have trailing commas at the end of the lines.

`$where` may be a code reference (sub { }), in which case the sub is called when the hook is encountered. `$_` is set to the value of the POD paragraph.

You may also specify the behaviour of whitespace trimming; by default, no trimming is done except on the HOOK names. Setting "−trim => 1" turns on a package "global" (until the next time import is called) that will trim the `$_` sent for processing by the hook processing function (be it a given function, or the built-in array/hash splitters) for leading and trailing whitespace.

The name of HOOK is matched against any "=head1", "=head2", "=item", "=for", "=begin" value. If you specify the special hooknames "*item", "*head1", etc, then you will get a function that is run for every

Note that the supplied functions for array and hash splitting are exactly equivalent to fairly simple Perl blocks:

Array:

```
HOOK => sub { @array = split /\n/, $_ }
```

Hash:

```
HOOK => sub {
%hash =
     (map { map { s/^\s+|\s+$//g; $_ } split /=>/, $_ }
       (grep m/^
           ( (?:[^=]|=[^>])+ )   # scan up to "=>"
           =>
           ( (?:[^=]|=[^>])+ =? )# don't allow more "=>"'s
           $/x, split /\n/, $_));
}
```

Well, they're simple if you can grok map, a regular expression like that and a functional programming style. If you can't I'm sure it is probably voodoo to you.

Here's the procedural equivalent:

```
 HOOK => sub {
    for my $line (split /\n/, $_) {
        my ($key, $value, $junk) = split /=>/, $line;
        next if $junk;
        $key =~ s/^\s+|\s+$//g
        $value =~ s/^\s+|\s+$//g
        $hash{$key} = $value;
    }
 },
```

import_from_file($filename, `@args`**)**

Very similar to straight "import", but you specify the source filename explicitly.

add_hook(NAME => value)

This function adds another hook, it is useful for dynamic updating of parsing through the document.

For an example, please see t/01−constants.t in the source distribution. More detailed examples will be added in a later release.

delete_hook(@list)

Deletes the named hooks. Companion function to add_hook

CLOSURES AS DESTINATIONS

If the given value is a ref CODE, then that function is called, with `$_` set to the value of the paragraph. This can be very useful for applying your own custom mutations to the POD to change it from human readable text into something your program can use.

After I added this function, I just kept on thinking of cool uses for it. The nice, succinct code you can make with it is one of Pod::Constant's strongest features.

Below are some examples.

EXAMPLES

Module Makefile.PL maintenance

Tired of keeping those module Makefile.PL's up to date? Note: This method seems to break dh-make-perl.

Example Makefile.PL

```
eval "use Pod::Constants";
($Pod::Constants::VERSION >= 0.11)
    or die <<EOF
####
####  ERROR: This module requires Pod::Constants 0.11 or
####  higher to be installed.
####
EOF

my ($VERSION, $NAME, $PREREQ_PM, $ABSTRACT, $AUTHOR);
Pod::Constants::import_from_file
     (
      'MyTestModule.pm',
      'MODULE RELEASE' => sub { ($VERSION) = m/(\d+\.\d+)/ },
      'DEPENDENCIES' => ($PREREQ_PM = { }),
       -trim => 1,
      'NAME' => sub { $ABSTRACT=$_; ($NAME) = m/(\S+)/ },
      'AUTHOR' => \$AUTHOR,
     );

WriteMakefile
     (
      'NAME'         => $NAME,
      'PREREQ_PM'        => $PREREQ_PM,
      'VERSION'          => $VERSION,
      ($] >= 5.005 ?    ## Add these new keywords supported since 5.005
       (ABSTRACT         => $ABSTRACT,
        AUTHOR           => $AUTHOR) : ()),
     );
```

Corresponding Module

```
=head1 NAME

MyTestModule - Demonstrate Pod::Constant's Makefile.PL usefulness

=head2 MODULE RELEASE

This is release 1.05 of this module.

=head2 DEPENDENCIES

The following modules are required to make this module:

   Some::Module => 0.02

=head2 AUTHOR

Ima Twat <ima@twat.name>

=cut

our $VERSION;
use Pod::Constants -trim => 1,
    'MODULE RELEASE' => sub { ($VERSION) = m/(\d+\.\d+) or die };
```

AUTHOR

Sam Vilain, <samv@cpan.org>

Maintained by Marius Gavrilescu, <marius@ieval.ro> since July 2015

COPYRIGHT AND LICENSE

BUGS/TODO

I keep thinking it would be nice to be able to import an =item list into an array or something, eg for a program argument list. But I'm not too sure how it would be all that useful in practice; you'd end up putting the function names for callbacks in the pod or something (perhaps not all that bad).

Would this be useful?

```
Pod::Constants::import(Foo::SECTION => \$myvar);
```

Debug output is not very readable