

NAME

Getopt::Long::Descriptive – Getopt::Long, but simpler and more powerful

VERSION

version 0.104

SYNOPSIS

```
use Getopt::Long::Descriptive;

my ($opt, $usage) = describe_options(
    'my-program %o <some-arg>',
    [ 'server|s=s', "the server to connect to", { required => 1 } ],
    [ 'port|p=i',   "the port to connect to",   { default   => 79 } ],
    [],
    [ 'verbose|v',  "print extra stuff"          ],
    [ 'help',       "print usage message and exit", { shortcircuit => 1 } ],
);

print($usage->text), exit if $opt->help;

Client->connect( $opt->server, $opt->port );

print "Connected!\n" if $opt->verbose;
...and running my-program --help will produce:

my-program [-psv] [long options...] <some-arg>
-s --server      the server to connect to
-p --port        the port to connect to

-v --verbose     print extra stuff
--help           print usage message and exit
```

DESCRIPTION

Getopt::Long::Descriptive is yet another Getopt library. It's built atop Getopt::Long, and gets a lot of its features, but tries to avoid making you think about its huge array of options.

It also provides usage (help) messages, data validation, and a few other useful features.

FUNCTIONS

Getopt::Long::Descriptive only exports one routine by default: `describe_options`. All GLD's exports are exported by `Sub::Exporter`.

describe_options

```
my ($opt, $usage) = describe_options($usage_desc, @opt_spec, \%arg);
```

This routine inspects `@ARGV` for options that match the supplied spec. If all the options are valid then it returns the options given and an object for generating usage messages; if not then it dies with an explanation of what was wrong and a usage message.

The `$opt` object will be a dynamically-generated subclass of `Getopt::Long::Descriptive::Opts`. In brief, each of the options in `@opt_spec` becomes an accessor method on the object, using the first-given name, with dashes converted to underscores. For more information, see the documentation for the `Opts` class.

The `$usage` object will be a `Getopt::Long::Descriptive::Usage` object, which provides a `text` method to get the text of the usage message and `die` to die with it. For more methods and options, consults the documentation for the `Usage` class.

\$usage_desc

The `$usage_desc` parameter to `describe_options` is a `sprintf`-like string that is used in generating the first line of the usage message. It's a one-line summary of how the command is to be

invoked. A typical usage description might be:

```
$usage_desc = "%c %o <source> <desc>";
```

`%c` will be replaced with what `Getopt::Long::Descriptive` thinks is the program name (it's computed from `$0`, see “`prog_name`”).

`%o` will be replaced with a list of the short options, as well as the text “[long options...]” if any have been defined.

The rest of the usage description can be used to summarize what arguments are expected to follow the program's options, and is entirely free-form.

Literal `%` characters will need to be written as `%%`, just like with `sprintf`.

@opt_spec

The `@opt_spec` part of the args to `describe_options` is used to configure option parsing and to produce the usage message. Each entry in the list is an arrayref describing one option, like this:

```
@opt_spec = (
  [ "verbose|V" => "be noisy"          ],
  [ "logfile=s" => "file to log to" ],
);
```

The first value in the arrayref is a `Getopt::Long`-style option specification. In brief, they work like this: each one is a pipe-delimited list of names, optionally followed by a type declaration. Type declarations are `'=x'` or `':x'`, where `=` means a value is required and `:` means it is optional. `x` may be `'s'` to indicate a string is required, `'i'` for an integer, or `'f'` for a number with a fractional part. The type spec may end in `@` to indicate that the option may appear multiple times.

For more information on how these work, see the `Getopt::Long` documentation.

The first name given should be the canonical name, as it will be used as the accessor method on the `$opt` object. Dashes in the name will be converted to underscores, and all letters will be lowercased. For this reason, all options should generally have a long-form name.

The second value in the arrayref is a description of the option, for use in the usage message.

Special Option Specifications

If the option specification (arrayref) is empty, it will have no effect other than causing a blank line to appear in the usage message.

If the option specification contains only one element, it will be printed in the usage message with no other effect. If the element is a reference, its referent will be printed as-is. Otherwise, it will be reformatted like other text in the usage message.

If the option specification contains a third element, it adds extra constraints or modifiers to the interpretation and validation of the value. These are the keys that may be present in that hashref, and how they behave:

implies

```
implies => 'bar'
implies => [qw(foo bar)]
implies => { foo => 1, bar => 2 }
```

If option *A* has an “implies” entry, then if *A* is given, other options will be enabled. The value may be a single option to set, an arrayref of options to set, or a hashref of options to set to specific values.

required

```
required => 1
```

If an option is required, failure to provide the option will result in `describe_options` printing the usage message and exiting.

hidden

```
hidden => 1
```

This option will not show up in the usage text.

You can achieve the same behavior by using the string “hidden” for the option’s description.

one_of

```
one_of => \@subopt_specs
```

This is useful for a group of options that are related. Each option spec is added to the list for normal parsing and validation.

Your option name will end up with a value of the name of the option that was chosen. For example, given the following spec:

```
[ "mode" => hidden => { one_of => [
  [ "get|g" => "get the value" ],
  [ "set|s" => "set the value" ],
  [ "delete" => "delete it" ],
] } ],
```

No usage text for ‘mode’ will be displayed, but text for get, set, and delete will be displayed.

If more than one of get, set, or delete is given, an error will be thrown.

So, given the @opt_spec above, and an @ARGV of ('--get '), the following would be true:

```
$opt->get == 1;

$opt->mode eq 'get';
```

Note: get would not be set if mode defaulted to ‘get’ and no arguments were passed in.

Even though the option sub-specs for one_of are meant to be ‘first class’ specs, some options don’t make sense with them, e.g. required.

As a further shorthand, you may specify one_of options using this form:

```
[ mode => \@option_specs, \%constraints ]
```

shortcircuit

```
shortcircuit => 1
```

If this option is present no other options will be returned. Other options present will be checked for proper types, but *not* for constraints. This provides a way of specifying --help style options.

Params::Validate

In addition, any constraint understood by Params::Validate may be used.

For example, to accept positive integers:

```
[ 'max-iterations=i', "maximum number of iterations",
  { callbacks => { positive => sub { shift() > 0 } } } ],
```

(Internally, all constraints are translated into Params::Validate options or callbacks.)

%arg

The %arg to describe_options is optional. If the last parameter is a hashref, it contains extra arguments to modify the way describe_options works. Valid arguments are:

```
getopt_conf    - an arrayref of strings, passed to Getopt::Long::Configure
show_defaults - a boolean which controls whether an option's default
                value (if applicable) is shown as part of the usage message
                (for backward compatibility this defaults to false)
```

prog_name

This routine, exported on demand, returns the basename of \$0, grabbed at compile-time. You can override this guess by calling `prog_name($string)` yourself.

OTHER EXPORTS**-types**

Any of the `Params::Validate` type constants (`SCALAR`, etc.) can be imported as well. You can get all of them at once by importing `-types`.

-all

This import group will import `-type`, `describe_options`, and `prog_name`.

CUSTOMIZING

`Getopt::Long::Descriptive` uses `Sub::Exporter` to build and export the `describe_options` routine. By writing a new class that extends `Getopt::Long::Descriptive`, the behavior of the constructed `describe_options` routine can be changed.

The following methods can be overridden:

usage_class

```
my $class = Getopt::Long::Descriptive->usage_class;
```

This returns the class to be used for constructing a `Usage` object, and defaults to `Getopt::Long::Descriptive::Usage`.

SEE ALSO

- `Getopt::Long`
- `Params::Validate`

AUTHORS

- Hans Dieter Pearcey <hdp@cpan.org>
- Ricardo Signes <rjbs@cpan.org>

CONTRIBUTORS

- Arthur Axel 'fREW' Schmidt <frioux@gmail.com>
- Dave Rolsky <autarch@urth.org>
- Diab Jerius <djerius@cfa.harvard.edu>
- Hans Dieter Pearcey <hdp@pobox.com>
- Hans Dieter Pearcey <hdp@weftsoar.net>
- Harley Pig <harleypig@gmail.com>
- hdp@cpan.org <hdp@cpan.org@fc0e91e4-031c-0410-8307-be39b06d7656>
- Karen Etheridge <ether@cpan.org>
- Niels Thykier <niels@thykier.net>
- Olaf Alders <olaf@wundersolutions.com>
- Roman Hubacek <roman.hubacek@centrum.cz>
- Smylers <SMYLERS@cpan.fsck.com>
- Thomas Neumann <blacky+perl@fluffbunny.de>
- zhouzhen1 <zhouzhen1@gmail.com>

COPYRIGHT AND LICENSE

This software is copyright (c) 2005 by Hans Dieter Pearcey.

This is free software; you can redistribute it and/or modify it under the same terms as the Perl 5 programming language system itself.