NAME

Getopt::Long - Extended processing of command line options

SYNOPSIS

```
use Getopt::Long;
my $data   = "file.dat";
my $length = 24;
my $verbose;
GetOptions ("length=i" => \$length,    # numeric
            "file=s"   => \$data,      # string
            "verbose"  => \$verbose)   # flag
or die("Error in command line arguments\n");
```

DESCRIPTION

The Getopt::Long module implements an extended getopt function called GetOptions(). It parses the command line from @ARGV, recognizing and removing specified options and their possible values.

This function adheres to the POSIX syntax for command line options, with GNU extensions. In general, this means that options have long names instead of single letters, and are introduced with a double dash "--". Support for bundling of command line options, as was the case with the more traditional single-letter approach, is provided but not enabled by default.

Command Line Options, an Introduction

Command line operated programs traditionally take their arguments from the command line, for example filenames or other information that the program needs to know. Besides arguments, these programs often take command line *options* as well. Options are not necessary for the program to work, hence the name 'option', but are used to modify its default behaviour. For example, a program could do its job quietly, but with a suitable option it could provide verbose information about what it did.

Command line options come in several flavours. Historically, they are preceded by a single dash `-`, and consist of a single letter.

```
-l -a -c
```

Usually, these single-character options can be bundled:

```
-lac
```

Options can have values, the value is placed after the option character. Sometimes with whitespace in between, sometimes not:

```
-s 24 -s24
```

Due to the very cryptic nature of these options, another style was developed that used long names. So instead of a cryptic `-l` one could use the more descriptive `--long`. To distinguish between a bundle of single-character options and a long one, two dashes are used to precede the option name. Early implementations of long options used a plus + instead. Also, option values could be specified either like

```
--size=24
```

or

```
--size 24
```

The + form is now obsolete and strongly deprecated.

Getting Started with Getopt::Long

Getopt::Long is the Perl5 successor of `newgetopt.pl`. This was the first Perl module that provided support for handling the new style of command line options, in particular long option names, hence the Perl5 name Getopt::Long. This module also supports single-character options and bundling.

To use Getopt::Long from a Perl program, you must include the following line in your Perl program:

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

This will load the core of the Getopt::Long module and prepare your program for using it. Most of the actual Getopt::Long code is not loaded until you really call one of its functions.

In the default configuration, options names may be abbreviated to uniqueness, case does not matter, and a single dash is sufficient, even for long option names. Also, options may be placed between non-option arguments. See *Configuring Getopt::Long* for more details on how to configure Getopt::Long.

Simple options

The most simple options are the ones that take no values. Their mere presence on the command line enables the option. Popular examples are:

```
--all --verbose --quiet --debug
```

Handling simple options is straightforward:

```
my $verbose = ''; # option variable with default value (false)
my $all = ''; # option variable with default value (false)
GetOptions ('verbose' => \$verbose, 'all' => \$all);
```

The call to GetOptions() parses the command line arguments that are present in `@ARGV` and sets the option variable to the value `1` if the option did occur on the command line. Otherwise, the option variable is not touched. Setting the option value to true is often called *enabling* the option.

The option name as specified to the GetOptions() function is called the option *specification*. Later we'll see that this specification can contain more than just the option name. The reference to the variable is called the option *destination*.

GetOptions() will return a true value if the command line could be processed successfully. Otherwise, it will write error messages using die() and warn(), and return a false result.

A little bit less simple options

Getopt::Long supports two useful variants of simple options: *negatable* options and *incremental* options.

A negatable option is specified with an exclamation mark `!` after the option name:

```
my $verbose = ''; # option variable with default value (false)
GetOptions ('verbose!' => \$verbose);
```

Now, using `--verbose` on the command line will enable `$verbose`, as expected. But it is also allowed to use `--noverbose`, which will disable `$verbose` by setting its value to `0`. Using a suitable default value, the program can find out whether `$verbose` is false by default, or disabled by using `--noverbose`.

An incremental option is specified with a plus + after the option name:

```
my $verbose = ''; # option variable with default value (false)
GetOptions ('verbose+' => \$verbose);
```

Using `--verbose` on the command line will increment the value of `$verbose`. This way the program can keep track of how many times the option occurred on the command line. For example, each occurrence of `--verbose` could increase the verbosity level of the program.

Mixing command line option with other arguments

Usually programs take command line options as well as other arguments, for example, file names. It is good practice to always specify the options first, and the other arguments last. Getopt::Long will, however, allow the options and arguments to be mixed and 'filter out' all the options before passing the rest of the arguments to the program. To stop Getopt::Long from processing further arguments, insert a double dash `--` on the command line:

```
--size 24 -- --all
```

In this example, `--all` will *not* be treated as an option, but passed to the program unharmed, in `@ARGV`.

Options with values

For options that take values it must be specified whether the option value is required or not, and what kind of value the option expects.

Three kinds of values are supported: integer numbers, floating point numbers, and strings.

If the option value is required, Getopt::Long will take the command line argument that follows the option and assign this to the option variable. If, however, the option value is specified as optional, this will only be done if that value does not look like a valid command line option itself.

```
my $tag = ''; # option variable with default value
GetOptions ('tag=s' => \$tag);
```

In the option specification, the option name is followed by an equals sign `=` and the letter `s`. The equals sign indicates that this option requires a value. The letter `s` indicates that this value is an arbitrary string. Other possible value types are `i` for integer values, and `f` for floating point values. Using a colon `:` instead of the equals sign indicates that the option value is optional. In this case, if no suitable value is supplied, string valued options get an empty string `''` assigned, while numeric options are set to `0`.

Options with multiple values

Options sometimes take several values. For example, a program could use multiple directories to search for library files:

```
--library lib/stdlib --library lib/extlib
```

To accomplish this behaviour, simply specify an array reference as the destination for the option:

```
GetOptions ("library=s" => \@libfiles);
```

Alternatively, you can specify that the option can have multiple values by adding a "@", and pass a scalar reference as the destination:

```
GetOptions ("library=s@" => \$libfiles);
```

Used with the example above, `@libfiles` (or `@$libfiles`) would contain two strings upon completion: `"lib/stdlib"` and `"lib/extlib"`, in that order. It is also possible to specify that only

integer or floating point numbers are acceptable values.

Often it is useful to allow comma-separated lists of values as well as multiple occurrences of the options. This is easy using Perl's split() and join() operators:

```
GetOptions ("library=s" => \@libfiles);
@libfiles = split(/,/,join(',',@libfiles));
```

Of course, it is important to choose the right separator string for each purpose.

Warning: What follows is an experimental feature.

Options can take multiple values at once, for example

```
--coordinates 52.2 16.4 --rgbcolor 255 255 149
```

This can be accomplished by adding a repeat specifier to the option specification. Repeat specifiers are very similar to the `{...}` repeat specifiers that can be used with regular expression patterns. For example, the above command line would be handled as follows:

```
GetOptions('coordinates=f{2}' => \@coor, 'rgbcolor=i{3}' => \@color);
```

The destination for the option must be an array or array reference.

It is also possible to specify the minimal and maximal number of arguments an option takes. `foo=s{2,4}` indicates an option that takes at least two and at most 4 arguments. `foo=s{1,}` indicates one or more values; `foo:s{,}` indicates zero or more option values.

Options with hash values

If the option destination is a reference to a hash, the option will take, as value, strings of the form *key*`=`*value*. The value will be stored with the specified key in the hash.

```
GetOptions ("define=s" => \%defines);
```

Alternatively you can use:

```
GetOptions ("define=s%" => \$defines);
```

When used with command line options:

```
--define os=linux --define vendor=redhat
```

the hash `%defines` (or `%$defines`) will contain two keys, `"os"` with value `"linux"` and `"vendor"` with value `"redhat"`. It is also possible to specify that only integer or floating point numbers are acceptable values. The keys are always taken to be strings.

User-defined subroutines to handle options

Ultimate control over what should be done when (actually: each time) an option is encountered on the command line can be achieved by designating a reference to a subroutine (or an anonymous subroutine) as the option destination. When GetOptions() encounters the option, it will call the subroutine with two or three arguments. The first argument is the name of the option. (Actually, it is an object that stringifies to the name of the option.) For a scalar or array destination, the second argument is the value to be stored. For a hash destination, the second argument is the key to the hash, and the third argument the value to be stored. It is up to the subroutine to store the value, or do whatever it thinks is appropriate.

A trivial application of this mechanism is to implement options that are related to each other. For example:

```
my $verbose = ''; # option variable with default value (false)
GetOptions ('verbose' => \$verbose,
     'quiet'   => sub { $verbose = 0 });
```

Here `--verbose` and `--quiet` control the same variable `$verbose`, but with opposite values.

If the subroutine needs to signal an error, it should call die() with the desired error message as its argument. GetOptions() will catch the die(), issue the error message, and record that an error result must be returned upon completion.

If the text of the error message starts with an exclamation mark `!` it is interpreted specially by GetOptions(). There is currently one special command implemented: `die("!FINISH")` will cause GetOptions() to stop processing options, as if it encountered a double dash `--`.

In version 2.37 the first argument to the callback function was changed from string to object. This was done to make room for extensions and more detailed control. The object stringifies to the option name so this change should not introduce compatibility problems.

Here is an example of how to access the option name and value from within a subroutine:

```
GetOptions ('opt=i' => \&handler);
sub handler {
    my ($opt_name, $opt_value) = @_;
    print("Option name is $opt_name and value is $opt_value\n");
}
```

Options with multiple names

Often it is user friendly to supply alternate mnemonic names for options. For example `--height` could be an alternate name for `--length`. Alternate names can be included in the option specification, separated by vertical bar `|` characters. To implement the above example:

```
GetOptions ('length|height=f' => \$length);
```

The first name is called the *primary* name, the other names are called *aliases*. When using a hash to store options, the key will always be the primary name.

Multiple alternate names are possible.

Case and abbreviations

Without additional configuration, GetOptions() will ignore the case of option names, and allow the options to be abbreviated to uniqueness.

```
GetOptions ('length|height=f' => \$length, "head" => \$head);
```

This call will allow `--l` and `--L` for the length option, but requires a least `--hea` and `--hei` for the head and height options.

Summary of Option Specifications

Each option specifier consists of two parts: the name specification and the argument specification.

The name specification contains the name of the option, optionally followed by a list of alternative names separated by vertical bar characters.

```
length       option name is "length"
length|size|l     name is "length", aliases are "size" and "l"
```

The argument specification is optional. If omitted, the option is considered boolean, a value of 1 will

be assigned when the option is used on the command line.

The argument specification can be

!

The option does not take an argument and may be negated by prefixing it with "no" or "no-". E.g. `"foo!"` will allow `--foo` (a value of 1 will be assigned) as well as `--nofoo` and `--no-foo` (a value of 0 will be assigned). If the option has aliases, this applies to the aliases as well.

Using negation on a single letter option when bundling is in effect is pointless and will result in a warning.

+

The option does not take an argument and will be incremented by 1 every time it appears on the command line. E.g. `"more+"`, when used with `--more --more --more`, will increment the value three times, resulting in a value of 3 (provided it was 0 or undefined at first).

The + specifier is ignored if the option destination is not a scalar.

= *type* [*desttype*] [*repeat*]

The option requires an argument of the given type. Supported types are:

s

String. An arbitrary sequence of characters. It is valid for the argument to start with `-` or `--`.

i

Integer. An optional leading plus or minus sign, followed by a sequence of digits.

o

Extended integer, Perl style. This can be either an optional leading plus or minus sign, followed by a sequence of digits, or an octal string (a zero, optionally followed by '0', '1', .. '7'), or a hexadecimal string (`0x` followed by '0' .. '9', 'a' .. 'f', case insensitive), or a binary string (`0b` followed by a series of '0' and '1').

f

Real number. For example `3.14`, `-6.23E24` and so on.

The *desttype* can be `@` or `%` to specify that the option is list or a hash valued. This is only needed when the destination for the option value is not otherwise specified. It should be omitted when not needed.

The *repeat* specifies the number of values this option takes per occurrence on the command line. It has the format `{` [*min*] [`,` [*max*]] `}`.

min denotes the minimal number of arguments. It defaults to 1 for options with = and to 0 for options with `:`, see below. Note that *min* overrules the = / `:` semantics.

max denotes the maximum number of arguments. It must be at least *min*. If *max* is omitted, *but the comma is not*, there is no upper bound to the number of argument values taken.

: *type* [*desttype*]

Like =, but designates the argument as optional. If omitted, an empty string will be assigned to string values options, and the value zero to numeric options.

Note that if a string argument starts with `-` or `--`, it will be considered an option on itself.

: *number* [*desttype*]

Like `:i`, but if the value is omitted, the *number* will be assigned.

: + [*desttype*]

Like `:i`, but if the value is omitted, the current value for the option will be incremented.

Advanced Possibilities

Object oriented interface

Getopt::Long can be used in an object oriented way as well:

```
use Getopt::Long;
$p = Getopt::Long::Parser->new;
$p->configure(...configuration options...);
if ($p->getoptions(...options descriptions...)) ...
if ($p->getoptionsfromarray( \@array, ...options descriptions...)) ...
```

Configuration options can be passed to the constructor:

```
$p = new Getopt::Long::Parser
         config => [...configuration options...];
```

Thread Safety

Getopt::Long is thread safe when using ithreads as of Perl 5.8. It is *not* thread safe when using the older (experimental and now obsolete) threads implementation that was added to Perl 5.005.

Documentation and help texts

Getopt::Long encourages the use of Pod::Usage to produce help messages. For example:

```
use Getopt::Long;
use Pod::Usage;

my $man = 0;
my $help = 0;

GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-exitval => 0, -verbose => 2) if $man;

__END__

=head1 NAME

sample - Using Getopt::Long and Pod::Usage

=head1 SYNOPSIS

sample [options] [file ...]

 Options:
   -help            brief help message
   -man             full documentation

=head1 OPTIONS

=over 8
```

```
=item B<-help>

Print a brief help message and exits.

=item B<-man>

Prints the manual page and exits.

=back

=head1 DESCRIPTION

B<This program> will read the given input file(s) and do something
useful with the contents thereof.

=cut
```

See *Pod::Usage* for details.

Parsing options from an arbitrary array

By default, GetOptions parses the options that are present in the global array @ARGV. A special entry `GetOptionsFromArray` can be used to parse options from an arbitrary array.

```
use Getopt::Long qw(GetOptionsFromArray);
$ret = GetOptionsFromArray(\@myopts, ...);
```

When used like this, options and their possible values are removed from `@myopts`, the global @ARGV is not touched at all.

The following two calls behave identically:

```
$ret = GetOptions( ... );
$ret = GetOptionsFromArray(\@ARGV, ... );
```

This also means that a first argument hash reference now becomes the second argument:

```
$ret = GetOptions(\%opts, ... );
$ret = GetOptionsFromArray(\@ARGV, \%opts, ... );
```

Parsing options from an arbitrary string

A special entry `GetOptionsFromString` can be used to parse options from an arbitrary string.

```
use Getopt::Long qw(GetOptionsFromString);
$ret = GetOptionsFromString($string, ...);
```

The contents of the string are split into arguments using a call to `Text::ParseWords::shellwords`. As with `GetOptionsFromArray`, the global @ARGV is not touched.

It is possible that, upon completion, not all arguments in the string have been processed. `GetOptionsFromString` will, when called in list context, return both the return status and an array reference to any remaining arguments:

```
($ret, $args) = GetOptionsFromString($string, ... );
```

If any arguments remain, and `GetOptionsFromString` was not called in list context, a message will be given and `GetOptionsFromString` will return failure.

As with GetOptionsFromArray, a first argument hash reference now becomes the second argument.

Storing options values in a hash

Sometimes, for example when there are a lot of options, having a separate variable for each of them can be cumbersome. GetOptions() supports, as an alternative mechanism, storing options values in a hash.

To obtain this, a reference to a hash must be passed *as the first argument* to GetOptions(). For each option that is specified on the command line, the option value will be stored in the hash with the option name as key. Options that are not actually used on the command line will not be put in the hash, on other words, `exists($h{option})` (or defined()) can be used to test if an option was used. The drawback is that warnings will be issued if the program runs under `use strict` and uses `$h{option}` without testing with exists() or defined() first.

```
my %h = ();
GetOptions (\%h, 'length=i'); # will store in $h{length}
```

For options that take list or hash values, it is necessary to indicate this by appending an @ or % sign after the type:

```
GetOptions (\%h, 'colours=s@'); # will push to @{$h{colours}}
```

To make things more complicated, the hash may contain references to the actual destinations, for example:

```
my $len = 0;
my %h = ('length' => \$len);
GetOptions (\%h, 'length=i'); # will store in $len
```

This example is fully equivalent with:

```
my $len = 0;
GetOptions ('length=i' => \$len); # will store in $len
```

Any mixture is possible. For example, the most frequently used options could be stored in variables while all other options get stored in the hash:

```
my $verbose = 0;   # frequently referred
my $debug = 0;   # frequently referred
my %h = ('verbose' => \$verbose, 'debug' => \$debug);
GetOptions (\%h, 'verbose', 'debug', 'filter', 'size=i');
if ( $verbose ) { ... }
if ( exists $h{filter} ) { ... option 'filter' was specified ... }
```

Bundling

With bundling it is possible to set several single-character options at once. For example if a, v and x are all valid options,

```
-vax
```

will set all three.

Getopt::Long supports three styles of bundling. To enable bundling, a call to Getopt::Long::Configure is required.

The simplest style of bundling can be enabled with:

```
Getopt::Long::Configure ("bundling");
```

Configured this way, single-character options can be bundled but long options **must** always start with a double dash `--` to avoid ambiguity. For example, when `vax`, `a`, `v` and `x` are all valid options,

```
-vax
```

will set `a`, `v` and `x`, but

```
--vax
```

will set `vax`.

The second style of bundling lifts this restriction. It can be enabled with:

```
Getopt::Long::Configure ("bundling_override");
```

Now, `-vax` will set the option `vax`.

In all of the above cases, option values may be inserted in the bundle. For example:

```
-h24w80
```

is equivalent to

```
-h 24 -w 80
```

A third style of bundling allows only values to be bundled with options. It can be enabled with:

```
Getopt::Long::Configure ("bundling_values");
```

Now, `-h24` will set the option `h` to `24`, but option bundles like `-vxa` and `-h24w80` are flagged as errors.

Enabling `bundling_values` will disable the other two styles of bundling.

When configured for bundling, single-character options are matched case sensitive while long options are matched case insensitive. To have the single-character options matched case insensitive as well, use:

```
Getopt::Long::Configure ("bundling", "ignorecase_always");
```

It goes without saying that bundling can be quite confusing.

The lonesome dash

Normally, a lone dash `-` on the command line will not be considered an option. Option processing will terminate (unless "permute" is configured) and the dash will be left in `@ARGV`.

It is possible to get special treatment for a lone dash. This can be achieved by adding an option specification with an empty name, for example:

```
GetOptions ('' => \$stdio);
```

A lone dash on the command line will now be a legal option, and using it will set variable `$stdio`.

Argument callback

A special option 'name' <> can be used to designate a subroutine to handle non-option arguments. When GetOptions() encounters an argument that does not look like an option, it will immediately call this subroutine and passes it one parameter: the argument name. Well, actually it is an object that stringifies to the argument name.

For example:

```
my $width = 80;
sub process { ... }
GetOptions ('width=i' => \$width, '<>' => \&process);
```

When applied to the following command line:

```
arg1 --width=72 arg2 --width=60 arg3
```

This will call `process("arg1")` while `$width` is `80`, `process("arg2")` while `$width` is `72`, and `process("arg3")` while `$width` is `60`.

This feature requires configuration option **permute**, see section *Configuring Getopt::Long*.

Configuring Getopt::Long

Getopt::Long can be configured by calling subroutine Getopt::Long::Configure(). This subroutine takes a list of quoted strings, each specifying a configuration option to be enabled, e.g. `ignore_case`, or disabled, e.g. `no_ignore_case`. Case does not matter. Multiple calls to Configure() are possible.

Alternatively, as of version 2.24, the configuration options may be passed together with the `use` statement:

```
use Getopt::Long qw(:config no_ignore_case bundling);
```

The following options are available:

default

This option causes all configuration options to be reset to their default values.

posix_default

This option causes all configuration options to be reset to their default values as if the environment variable POSIXLY_CORRECT had been set.

auto_abbrev

Allow option names to be abbreviated to uniqueness. Default is enabled unless environment variable POSIXLY_CORRECT has been set, in which case `auto_abbrev` is disabled.

getopt_compat

Allow + to start options. Default is enabled unless environment variable POSIXLY_CORRECT has been set, in which case `getopt_compat` is disabled.

gnu_compat

`gnu_compat` controls whether `--opt=` is allowed, and what it should do. Without `gnu_compat`, `--opt=` gives an error. With `gnu_compat`, `--opt=` will give option `opt` and empty value. This is the way GNU getopt_long() does it.

gnu_getopt

This is a short way of setting `gnu_compat bundling permute`

no_getopt_compat. With gnu_getopt, command line handling should be fully compatible with GNU getopt_long().

require_order

Whether command line arguments are allowed to be mixed with options. Default is disabled unless environment variable POSIXLY_CORRECT has been set, in which case require_order is enabled.

See also permute, which is the opposite of require_order.

permute

Whether command line arguments are allowed to be mixed with options. Default is enabled unless environment variable POSIXLY_CORRECT has been set, in which case permute is disabled. Note that permute is the opposite of require_order.

If permute is enabled, this means that

```
--foo arg1 --bar arg2 arg3
```

is equivalent to

```
--foo --bar arg1 arg2 arg3
```

If an argument callback routine is specified, @ARGV will always be empty upon successful return of GetOptions() since all options have been processed. The only exception is when -- is used:

```
--foo arg1 --bar arg2 -- arg3
```

This will call the callback routine for arg1 and arg2, and then terminate GetOptions() leaving "arg3" in @ARGV.

If require_order is enabled, options processing terminates when the first non-option is encountered.

```
--foo arg1 --bar arg2 arg3
```

is equivalent to

```
--foo -- arg1 --bar arg2 arg3
```

If pass_through is also enabled, options processing will terminate at the first unrecognized option, or non-option, whichever comes first.

bundling (default: disabled)

Enabling this option will allow single-character options to be bundled. To distinguish bundles from long option names, long options *must* be introduced with -- and bundles with -.

Note that, if you have options a, l and all, and auto_abbrev enabled, possible arguments and option settings are:

```
using argument                sets option(s)
------------------------------------------
-a, --a                       a
-l, --l                       l
-al, -la, -ala, -all,...      a, l
--al, --all                   all
```

The surprising part is that --a sets option a (due to auto completion), not all.

Note: disabling bundling also disables bundling_override.

bundling_override (default: disabled)

If `bundling_override` is enabled, bundling is enabled as with `bundling` but now long option names override option bundles.

Note: disabling `bundling_override` also disables `bundling`.

Note: Using option bundling can easily lead to unexpected results, especially when mixing long options and bundles. Caveat emptor.

ignore_case (default: enabled)

If enabled, case is ignored when matching option names. If, however, bundling is enabled as well, single character options will be treated case-sensitive.

With `ignore_case`, option specifications for options that only differ in case, e.g., `"foo"` and `"Foo"`, will be flagged as duplicates.

Note: disabling `ignore_case` also disables `ignore_case_always`.

ignore_case_always (default: disabled)

When bundling is in effect, case is ignored on single-character options also.

Note: disabling `ignore_case_always` also disables `ignore_case`.

auto_version (default:disabled)

Automatically provide support for the **--version** option if the application did not specify a handler for this option itself.

Getopt::Long will provide a standard version message that includes the program name, its version (if $main::VERSION is defined), and the versions of Getopt::Long and Perl. The message will be written to standard output and processing will terminate.

`auto_version` will be enabled if the calling program explicitly specified a version number higher than 2.32 in the `use` or `require` statement.

auto_help (default:disabled)

Automatically provide support for the **--help** and **-?** options if the application did not specify a handler for this option itself.

Getopt::Long will provide a help message using module *Pod::Usage*. The message, derived from the SYNOPSIS POD section, will be written to standard output and processing will terminate.

`auto_help` will be enabled if the calling program explicitly specified a version number higher than 2.32 in the `use` or `require` statement.

pass_through (default: disabled)

With `pass_through` anything that is unknown, ambiguous or supplied with an invalid option will not be flagged as an error. Instead the unknown option(s) will be passed to the catchall <> if present, otherwise through to @ARGV. This makes it possible to write wrapper scripts that process only part of the user supplied command line arguments, and pass the remaining options to some other program.

If `require_order` is enabled, options processing will terminate at the first unrecognized option, or non-option, whichever comes first and all remaining arguments are passed to @ARGV instead of the catchall <> if present. However, if `permute` is enabled instead, results can become confusing.

Note that the options terminator (default `--`), if present, will also be passed through in @ARGV.

prefix

The string that starts options. If a constant string is not sufficient, see `prefix_pattern`.

prefix_pattern

A Perl pattern that identifies the strings that introduce options. Default is `--|-|\+` unless environment variable POSIXLY_CORRECT has been set, in which case it is `--|-`.

long_prefix_pattern

A Perl pattern that allows the disambiguation of long and short prefixes. Default is `--`.

Typically you only need to set this if you are using nonstandard prefixes and want some or all of them to have the same semantics as '--' does under normal circumstances.

For example, setting prefix_pattern to `--|-|\+|\/` and long_prefix_pattern to `--|\/` would add Win32 style argument handling.

debug (default: disabled)

Enable debugging output.

Exportable Methods

VersionMessage

This subroutine provides a standard version message. Its argument can be:

- A string containing the text of a message to print *before* printing the standard message.
- A numeric value corresponding to the desired exit status.
- A reference to a hash.

If more than one argument is given then the entire argument list is assumed to be a hash. If a hash is supplied (either as a reference or as a list) it should contain one or more elements with the following keys:

`-message`

`-msg`

The text of a message to print immediately prior to printing the program's usage message.

`-exitval`

The desired exit status to pass to the **exit()** function. This should be an integer, or else the string "NOEXIT" to indicate that control should simply be returned without terminating the invoking process.

`-output`

A reference to a filehandle, or the pathname of a file to which the usage message should be written. The default is `\*STDERR` unless the exit value is less than 2 (in which case the default is `\*STDOUT`).

You cannot tie this routine directly to an option, e.g.:

```
GetOptions("version" => \&VersionMessage);
```

Use this instead:

```
GetOptions("version" => sub { VersionMessage() });
```

HelpMessage

This subroutine produces a standard help message, derived from the program's POD section SYNOPSIS using *Pod::Usage*. It takes the same arguments as VersionMessage(). In particular, you cannot tie it directly to an option, e.g.:

```
GetOptions("help" => \&HelpMessage);
```

Use this instead:

```
GetOptions("help" => sub { HelpMessage() });
```

Return values and Errors

Configuration errors and errors in the option definitions are signalled using die() and will terminate the calling program unless the call to Getopt::Long::GetOptions() was embedded in `eval { ... }`, or die() was trapped using `$SIG{__DIE__}`.

GetOptions returns true to indicate success. It returns false when the function detected one or more errors during option parsing. These errors are signalled using warn() and can be trapped with `$SIG{__WARN__}`.

Legacy

The earliest development of `newgetopt.pl` started in 1990, with Perl version 4. As a result, its development, and the development of Getopt::Long, has gone through several stages. Since backward compatibility has always been extremely important, the current version of Getopt::Long still supports a lot of constructs that nowadays are no longer necessary or otherwise unwanted. This section describes briefly some of these 'features'.

Default destinations

When no destination is specified for an option, GetOptions will store the resultant value in a global variable named `opt_`*XXX*, where *XXX* is the primary name of this option. When a program executes under `use strict` (recommended), these variables must be pre-declared with our() or `use vars`.

```
our $opt_length = 0;
GetOptions ('length=i'); # will store in $opt_length
```

To yield a usable Perl variable, characters that are not part of the syntax for variables are translated to underscores. For example, `--fpp-struct-return` will set the variable `$opt_fpp_struct_return`. Note that this variable resides in the namespace of the calling program, not necessarily `main`. For example:

```
GetOptions ("size=i", "sizes=i@");
```

with command line "-size 10 -sizes 24 -sizes 48" will perform the equivalent of the assignments

```
$opt_size = 10;
@opt_sizes = (24, 48);
```

Alternative option starters

A string of alternative option starter characters may be passed as the first argument (or the first argument after a leading hash reference argument).

```
my $len = 0;
GetOptions ('/', 'length=i' => $len);
```

Now the command line may look like:

```
/length 24 -- arg
```

Note that to terminate options processing still requires a double dash `--`.

GetOptions() will not interpret a leading `"<>"` as option starters if the next argument is a reference. To force `"<"` and `">"` as option starters, use `"><"`. Confusing? Well, **using a starter argument is strongly deprecated** anyway.

Configuration variables

Previous versions of Getopt::Long used variables for the purpose of configuring. Although manipulating these variables still work, it is strongly encouraged to use the `Configure` routine that was introduced in version 2.17. Besides, it is much easier.

Tips and Techniques

Pushing multiple values in a hash option

Sometimes you want to combine the best of hashes and arrays. For example, the command line:

```
--list add=first --list add=second --list add=third
```

where each successive 'list add' option will push the value of add into array ref $list->{'add'}. The result would be like

```
$list->{add} = [qw(first second third)];
```

This can be accomplished with a destination routine:

```
GetOptions('list=s%' =>
            sub { push(@{$list{$_[1]}}, $_[2]) });
```

Troubleshooting

GetOptions does not return a false result when an option is not supplied

That's why they're called 'options'.

GetOptions does not split the command line correctly

The command line is not split by GetOptions, but by the command line interpreter (CLI). On Unix, this is the shell. On Windows, it is COMMAND.COM or CMD.EXE. Other operating systems have other CLIs.

It is important to know that these CLIs may behave different when the command line contains special characters, in particular quotes or backslashes. For example, with Unix shells you can use single quotes (`'`) and double quotes (`"`) to group words together. The following alternatives are equivalent on Unix:

```
"two words"
'two words'
two\ words
```

In case of doubt, insert the following statement in front of your Perl program:

```
print STDERR (join("|",@ARGV),"\n");
```

to verify how your CLI passes the arguments to the program.

Undefined subroutine &main::GetOptions called

Are you running Windows, and did you write

```
use GetOpt::Long;
```

(note the capital 'O')?

How do I put a "-?" option into a Getopt::Long?

You can only obtain this using an alias, and Getopt::Long of at least version 2.13.

```
use Getopt::Long;
GetOptions ("help|?");    # -help and -? will both set $opt_help
```

Other characters that can't appear in Perl identifiers are also supported as aliases with Getopt::Long of at least version 2.39.

As of version 2.32 Getopt::Long provides auto-help, a quick and easy way to add the options --help and -? to your program, and handle them.

See `auto_help` in section *Configuring Getopt::Long*.

AUTHOR

Johan Vromans <jvromans@squirrel.nl>

COPYRIGHT AND DISCLAIMER