NAME

Params::Check - A generic input parsing/checking mechanism.

SYNOPSIS

```
use Params::Check qw[check allow last_error];

sub fill_personal_info {
    my %hash = @_;
    my $x;

    my $tmpl = {
        firstname   => { required   => 1, defined => 1 },
        lastname    => { required   => 1, store => \$x },
        gender      => { required   => 1,
                         allow      => [qr/M/i, qr/F/i],
                       },
        married     => { allow      => [0,1] },
        age         => { default    => 21,
                         allow      => qr/^\d+$/,
                       },

        phone       => { allow => [ sub { return 1 if /$valid_re/ },
                                    '1-800-PERL' ]
                       },
        id_list     => { default        => [],
                         strict_type    => 1
                       },
        employer    => { default => 'NSA', no_override => 1 },
    };

    ### check() returns a hashref of parsed args on success ###
    my $parsed_args = check( $tmpl, \%hash, $VERBOSE )
                        or die qw[Could not parse arguments!];

    ... other code here ...
}

my $ok = allow( $colour, [qw|blue green yellow|] );

my $error = Params::Check::last_error();
```

DESCRIPTION

Params::Check is a generic input parsing/checking mechanism.

It allows you to validate input via a template. The only requirement is that the arguments must be named.

Params::Check can do the following things for you:

- Convert all keys to lowercase
- Check if all required arguments have been provided
- Set arguments that have not been provided to the default

- Weed out arguments that are not supported and warn about them to the user
- Validate the arguments given by the user based on strings, regexes, lists or even subroutines
- Enforce type integrity if required

Most of Params::Check's power comes from its template, which we'll discuss below:

Template

As you can see in the synopsis, based on your template, the arguments provided will be validated.

The template can take a different set of rules per key that is used.

The following rules are available:

default

This is the default value if none was provided by the user. This is also the type `strict_type` will look at when checking type integrity (see below).

required

A boolean flag that indicates if this argument was a required argument. If marked as required and not provided, check() will fail.

strict_type

This does a `ref()` check on the argument provided. The `ref` of the argument must be the same as the `ref` of the default value for this check to pass.

This is very useful if you insist on taking an array reference as argument for example.

defined

If this template key is true, enforces that if this key is provided by user input, its value is `defined`. This just means that the user is not allowed to pass `undef` as a value for this key and is equivalent to: allow => sub { defined $_[0] && OTHER TESTS }

no_override

This allows you to specify `constants` in your template. ie, they keys that are not allowed to be altered by the user. It pretty much allows you to keep all your `configurable` data in one place; the `Params::Check` template.

store

This allows you to pass a reference to a scalar, in which the data will be stored:

```
my $x;
my $args = check(foo => { default => 1, store => \$x }, $input);
```

This is basically shorthand for saying:

```
my $args = check( { foo => { default => 1 }, $input );
my $x    = $args->{foo};
```

You can alter the global variable $Params::Check::NO_DUPLICATES to control whether the `store`'d key will still be present in your result set. See the *Global Variables* section below.

allow

A set of criteria used to validate a particular piece of data if it has to adhere to particular rules.

See the `allow()` function for details.

Functions

check(\%tmpl, \%args, [$verbose]);

This function is not exported by default, so you'll have to ask for it via:

```
use Params::Check qw[check];
```

or use its fully qualified name instead.

`check` takes a list of arguments, as follows:

Template

This is a hashreference which contains a template as explained in the `SYNOPSIS` and `Template` section.

Arguments

This is a reference to a hash of named arguments which need checking.

Verbose

A boolean to indicate whether `check` should be verbose and warn about what went wrong in a check or not.

You can enable this program wide by setting the package variable `$Params::Check::VERBOSE` to a true value. For details, see the section on `Global Variables` below.

`check` will return when it fails, or a hashref with lowercase keys of parsed arguments when it succeeds.

So a typical call to check would look like this:

```
my $parsed = check( \%template, \%arguments, $VERBOSE )
                or warn q[Arguments could not be parsed!];
```

A lot of the behaviour of `check()` can be altered by setting package variables. See the section on `Global Variables` for details on this.

allow($test_me, \@criteria);

The function that handles the `allow` key in the template is also available for independent use.

The function takes as first argument a key to test against, and as second argument any form of criteria that are also allowed by the `allow` key in the template.

You can use the following types of values for allow:

string

The provided argument MUST be equal to the string for the validation to pass.

regexp

The provided argument MUST match the regular expression for the validation to pass.

subroutine

The provided subroutine MUST return true in order for the validation to pass and the argument accepted.

(This is particularly useful for more complicated data).

array ref

The provided argument MUST equal one of the elements of the array ref for the validation to

pass. An array ref can hold all the above values.

It returns true if the key matched the criteria, or false otherwise.

last_error()

Returns a string containing all warnings and errors reported during the last time `check` was called.

This is useful if you want to report then some other way than `carp`'ing when the verbose flag is on.

It is exported upon request.

Global Variables

The behaviour of Params::Check can be altered by changing the following global variables:

$Params::Check::VERBOSE

This controls whether Params::Check will issue warnings and explanations as to why certain things may have failed. If you set it to 0, Params::Check will not output any warnings.

The default is 1 when *warnings* are enabled, 0 otherwise;

$Params::Check::STRICT_TYPE

This works like the `strict_type` option you can pass to `check`, which will turn on `strict_type` globally for all calls to `check`.

The default is 0;

$Params::Check::ALLOW_UNKNOWN

If you set this flag, unknown options will still be present in the return value, rather than filtered out. This is useful if your subroutine is only interested in a few arguments, and wants to pass the rest on blindly to perhaps another subroutine.

The default is 0;

$Params::Check::STRIP_LEADING_DASHES

If you set this flag, all keys passed in the following manner:

```
function( -key => 'val' );
```

will have their leading dashes stripped.

$Params::Check::NO_DUPLICATES

If set to true, all keys in the template that are marked as to be stored in a scalar, will also be removed from the result set.

Default is false, meaning that when you use `store` as a template key, `check` will put it both in the scalar you supplied, as well as in the hashref it returns.

$Params::Check::PRESERVE_CASE

If set to true, *Params::Check* will no longer convert all keys from the user input to lowercase, but instead expect them to be in the case the template provided. This is useful when you want to use similar keys with different casing in your templates.

Understand that this removes the case-insensitivy feature of this module.

Default is 0;

$Params::Check::ONLY_ALLOW_DEFINED

If set to true, *Params::Check* will require all values passed to be `defined`. If you wish to enable this on a 'per key' basis, use the template option `defined` instead.

Default is 0;

$Params::Check::SANITY_CHECK_TEMPLATE

If set to true, *Params::Check* will sanity check templates, validating for errors and unknown keys. Although very useful for debugging, this can be somewhat slow in hot-code and large loops.

To disable this check, set this variable to `false`.

Default is 1;

$Params::Check::WARNINGS_FATAL

If set to true, *Params::Check* will `croak` when an error during template validation occurs, rather than return `false`.

Default is 0;

$Params::Check::CALLER_DEPTH

This global modifies the argument given to `caller()` by `Params::Check::check()` and is useful if you have a custom wrapper function around `Params::Check::check()`. The value must be an integer, indicating the number of wrapper functions inserted between the real function call and `Params::Check::check()`.

Example wrapper function, using a custom stacktrace:

```
    sub check {
        my ($template, $args_in) = @_;

        local $Params::Check::WARNINGS_FATAL = 1;
        local $Params::Check::CALLER_DEPTH = $Params::Check::CALLER_DEPTH +
 1;
        my $args_out = Params::Check::check($template, $args_in);

        my_stacktrace(Params::Check::last_error) unless $args_out;

        return $args_out;
    }
```

Default is 0;

AUTHOR

This module by Jos Boumans <kane@cpan.org>.

Acknowledgements

Thanks to Richard Soderberg for his performance improvements.

COPYRIGHT