

**NAME**

Variable::Magic – Associate user-defined magic to variables from Perl.

**VERSION**

Version 0.62

**SYNOPSIS**

```
use Variable::Magic qw<wizard cast VMG_OP_INFO_NAME>;

{ # A variable tracer
  my $wiz = wizard(
    set => sub { print "now set to ${$_[0]}!\n" },
    free => sub { print "destroyed!\n" },
  );

  my $a = 1;
  cast $a, $wiz;
  $a = 2;          # "now set to 2!"
}                  # "destroyed!"

{ # A hash with a default value
  my $wiz = wizard(
    data      => sub { $_[1] },
    fetch     => sub { $_[2] = $_[1] unless exists $_[0]->{$_[2]}; () },
    store     => sub { print "key $_[2] stored in $_[1]\n" },
    copy_key  => 1,
    op_info   => VMG_OP_INFO_NAME,
  );

  my %h = (_default => 0, apple => 2);
  cast %h, $wiz, '_default';
  print $h{banana}, "\n"; # "0" (there is no 'banana' key in %h)
  $h{pear} = 1;          # "key pear stored in helem"
}
```

**DESCRIPTION**

Magic is Perl's way of enhancing variables. This mechanism lets the user add extra data to any variable and hook syntactical operations (such as access, assignment or destruction) that can be applied to it. With this module, you can add your own magic to any variable without having to write a single line of XS.

You'll realize that these magic variables look a lot like tied variables. It is not surprising, as tied variables are implemented as a special kind of magic, just like any 'irregular' Perl variable : scalars like \$!, \$( or \$^W, the %ENV and %SIG hashes, the @ISA array, vec() and substr() lvalues, threads::shared variables... They all share the same underlying C API, and this module gives you direct access to it.

Still, the magic made available by this module differs from tying and overloading in several ways :

- Magic is not copied on assignment.

You attach it to variables, not values (as for blessed references).

- Magic does not replace the original semantics.

Magic callbacks usually get triggered before the original action takes place, and cannot prevent it from happening. This also makes catching individual events easier than with tie, where you have to provide fallbacks methods for all actions by usually inheriting from the correct Tie::Std\* class and overriding individual methods in your own class.

- Magic is multivalued.

You can safely apply different kinds of magics to the same variable, and each of them will be invoked

successively.

- Magic is type-agnostic.

The same magic can be applied on scalars, arrays, hashes, subs or globs. But the same hook (see below for a list) may trigger differently depending on the type of the variable.

- Magic is invisible at Perl level.

Magical and non-magical variables cannot be distinguished with `ref`, `tied` or another trick.

- Magic is notably faster.

Mainly because perl's way of handling magic is lighter by nature, and because there is no need for any method resolution. Also, since you don't have to reimplement all the variable semantics, you only pay for what you actually use.

The operations that can be overloaded are :

- *get*

This magic is invoked when the variable is evaluated. It is never called for arrays and hashes.

- *set*

This magic is called each time the value of the variable changes. It is called for array subscripts and slices, but never for hashes.

- *len*

This magic only applies to arrays (though it used to also apply to scalars), and is triggered when the 'size' or the 'length' of the variable has to be known by Perl. This is typically the magic involved when an array is evaluated in scalar context, but also on array assignment and loops (`for`, `map` or `grep`). The length is returned from the callback as an integer.

Starting from perl 5.12, this magic is no longer called by the `length` keyword, and starting from perl 5.17.4 it is also no longer called for scalars in any situation, making this magic only meaningful on arrays. You can use the constants `"VMG_COMPAT_SCALAR_LENGTH_NOLEN"` and `"VMG_COMPAT_SCALAR_NOLEN"` to see if this magic is available for scalars or not.

- *clear*

This magic is invoked when the variable is reset, such as when an array is emptied. Please note that this is different from undefining the variable, even though the magic is called when the clearing is a result of the `undef` (e.g. for an array, but actually a bug prevent it to work before perl 5.9.5 – see the history).

- *free*

This magic is called when a variable is destroyed as the result of going out of scope (but not when it is undefined). It behaves roughly like Perl object destructors (i.e. `DESTROY` methods), except that exceptions thrown from inside a *free* callback will always be propagated to the surrounding code.

- *copy*

When applied to tied arrays and hashes, this magic fires when you try to access or change their elements.

Starting from perl 5.17.0, it can also be applied to closure prototypes, in which case the magic will be called when the prototype is cloned. The `"VMG_COMPAT_CODE_COPY_CLONE"` constant is true when your perl support this feature.

- *dup*

This magic is invoked when the variable is cloned across threads. It is currently not available.

- *local*

When this magic is set on a variable, all subsequent localizations of the variable will trigger the callback. It is available on your perl if and only if `MGf_LOCAL` is true.

The following actions only apply to hashes and are available if and only if “`VMG_UVAR`” is true. They are referred to as *uvar* magics.

- *fetch*  
This magic is invoked each time an element is fetched from the hash.
- *store*  
This one is called when an element is stored into the hash.
- *exists*  
This magic fires when a key is tested for existence in the hash.
- *delete*  
This magic is triggered when a key is deleted in the hash, regardless of whether the key actually exists in it.

You can refer to the tests to have more insight of where the different magics are invoked.

## FUNCTIONS

wizard

```
wizard(
  data      => sub { ... },
  get       => sub { my ($ref, $data [, $op]) = @_; ... },
  set       => sub { my ($ref, $data [, $op]) = @_; ... },
  len       => sub {
    my ($ref, $data, $len [, $op]) = @_; ... ; return $newlen
  },
  clear     => sub { my ($ref, $data [, $op]) = @_; ... },
  free      => sub { my ($ref, $data [, $op]) = @_, ... },
  copy      => sub { my ($ref, $data, $key, $elt [, $op]) = @_; ... },
  local     => sub { my ($ref, $data [, $op]) = @_; ... },
  fetch     => sub { my ($ref, $data, $key [, $op]) = @_; ... },
  store     => sub { my ($ref, $data, $key [, $op]) = @_; ... },
  exists    => sub { my ($ref, $data, $key [, $op]) = @_; ... },
  delete    => sub { my ($ref, $data, $key [, $op]) = @_; ... },
  copy_key  => $bool,
  op_info   => [ 0 | VMG_OP_INFO_NAME | VMG_OP_INFO_OBJECT ],
)
```

This function creates a ‘wizard’, an opaque object that holds the magic information. It takes a list of keys / values as argument, whose keys can be :

- *data*  
A code (or string) reference to a private data constructor. It is called in scalar context each time the magic is cast onto a variable, with `$_[0]` being a reference to this variable and `@_[1 .. @_-1]` being all extra arguments that were passed to “cast”. The scalar returned from this call is then attached to the variable and can be retrieved later with “`getdata`”.
- *get, set, len, clear, free, copy, local, fetch, store, exists and delete*  
Code (or string) references to the respective magic callbacks. You don’t have to specify all of them : the magic corresponding to undefined entries will simply not be hooked.  
When those callbacks are executed, `$_[0]` is a reference to the magic variable and `$_[1]` is the associated private data (or `undef` when no private data constructor is supplied with the wizard). Other arguments depend on which kind of magic is involved :

- *len*  
`$_[2]` contains the natural, non-magical length of the variable (which can only be a scalar or an array as *len* magic is only relevant for these types). The callback is expected to return the new scalar or array length to use, or `undef` to default to the normal length.
- *copy*  
 When the variable for which the magic is invoked is an array or an hash, `$_[2]` is either an alias or a copy of the current key, and `$_[3]` is an alias to the current element (i.e. the value). Since `$_[2]` might be a copy, it is useless to try to change it or cast magic on it.  
 Starting from perl 5.17.0, this magic can also be called for code references. In this case, `$_[2]` is always `undef` and `$_[3]` is a reference to the cloned anonymous subroutine.
- *fetch, store, exists and delete*  
`$_[2]` is an alias to the current key. Note that `$_[2]` may rightfully be readonly if the key comes from a bareword, and as such it is unsafe to assign to it. You can ask for a copy instead by passing `copy_key => 1` to “wizard” which, at the price of a small performance hit, allows you to safely assign to `$_[2]` in order to e.g. redirect the action to another key.

Finally, if `op_info => $num` is also passed to `wizard`, then one extra element is appended to `@_`. Its nature depends on the value of `$num` :

- `VMG_OP_INFO_NAME`  
`$_[-1]` is the current op name.
- `VMG_OP_INFO_OBJECT`  
`$_[-1]` is the `B::OP` object for the current op.

Both result in a small performance hit, but just getting the name is lighter than getting the op object.

These callbacks are always executed in scalar context. The returned value is coerced into a signed integer, which is then passed straight to the perl magic API. However, note that perl currently only cares about the return value of the *len* magic callback and ignores all the others. Starting with Variable::Magic 0.58, a reference returned from a non-*len* magic callback will not be destroyed immediately but will be allowed to survive until the end of the statement that triggered the magic. This lets you use this return value as a token for triggering a destructor after the original magic action takes place. You can see an example of this technique in the cookbook.

Each callback can be specified as :

- a code reference, which will be called as a subroutine.
- a string reference, where the string denotes which subroutine is to be called when magic is triggered. If the subroutine name is not fully qualified, then the current package at the time the magic is invoked will be used instead.
- a reference to `undef`, in which case a no-op magic callback is installed instead of the default one. This may especially be helpful for *local* magic, where an empty callback prevents magic from being copied during localization.

Note that *free* magic is never called during global destruction, as there is no way to ensure that the wizard object and the callback were not destroyed before the variable.

Here is a simple usage example :

```
# A simple scalar tracer
my $wiz = wizard(
    get => sub { print STDERR "got ${$_[0]}\n" },
    set => sub { print STDERR "set to ${$_[0]}\n" },
    free => sub { print STDERR "${$_[0]} was deleted\n" },
);
```

cast

```
cast [$@%&*]var, $wiz, @args
```

This function associates \$wiz magic to the supplied variable, without overwriting any other kind of magic. It returns true on success or when \$wiz magic is already attached, and croaks on error. When \$wiz provides a data constructor, it is called just before magic is cast onto the variable, and it receives a reference to the target variable in \$\_[0] and the content of @args in @\_[1 .. @args]. Otherwise, @args is ignored.

```
# Casts $wiz onto $x, passing (\$x, '1') to the data constructor.
my $x;
cast $x, $wiz, 1;
```

The var argument can be an array or hash value. Magic for these scalars behaves like for any other, except that it is dispelled when the entry is deleted from the container. For example, if you want to call POSIX::tzset each time the 'TZ' environment variable is changed in %ENV, you can use :

```
use POSIX;
cast $ENV{TZ}, wizard set => sub { POSIX::tzset(); () };
```

If you want to handle the possible deletion of the 'TZ' entry, you must also specify *store* magic.

getdata

```
getdata [$@%&*]var, $wiz
```

This accessor fetches the private data associated with the magic \$wiz in the variable. It croaks when \$wiz does not represent a valid magic object, and returns an empty list if no such magic is attached to the variable or when the wizard has no data constructor.

```
# Get the data attached to $wiz in $x, or undef if $wiz
# did not attach any.
my $data = getdata $x, $wiz;
```

dispell

```
dispell [$@%&*]variable, $wiz
```

The exact opposite of “cast” : it dissociates \$wiz magic from the variable. This function returns true on success, 0 when no magic represented by \$wiz could be found in the variable, and croaks if the supplied wizard is invalid.

```
# Dispell now.
die 'no such magic in $x' unless dispell $x, $wiz;
```

## CONSTANTS

MGf\_COPY

Evaluates to true if and only if the *copy* magic is available. This is the case for perl 5.7.3 and greater, which is ensured by the requirements of this module.

MGf\_DUP

Evaluates to true if and only if the *dup* magic is available. This is the case for perl 5.7.3 and greater, which is ensured by the requirements of this module.

MGf\_LOCAL

Evaluates to true if and only if the *local* magic is available. This is the case for perl 5.9.3 and greater.

**VMG\_UVAR**

When this constant is true, you can use the *fetch*, *store*, *exists* and *delete* magics on hashes. Initial “VMG\_UVAR” capability was introduced in perl 5.9.5, with a fully functional implementation shipped with perl 5.10.0.

**VMG\_COMPAT\_SCALAR\_LENGTH\_NOLEN**

True for perls that don’t call *len* magic when taking the `length` of a magical scalar.

**VMG\_COMPAT\_SCALAR\_NOLEN**

True for perls that don’t call *len* magic on scalars. Implies “VMG\_COMPAT\_SCALAR\_LENGTH\_NOLEN”.

**VMG\_COMPAT\_ARRAY\_PUSH\_NOLEN**

True for perls that don’t call *len* magic when you push an element in a magical array. Starting from perl 5.11.0, this only refers to pushes in non-void context and hence is false.

**VMG\_COMPAT\_ARRAY\_PUSH\_NOLEN\_VOID**

True for perls that don’t call *len* magic when you push in void context an element in a magical array.

**VMG\_COMPAT\_ARRAY\_UNSHIFT\_NOLEN\_VOID**

True for perls that don’t call *len* magic when you unshift in void context an element in a magical array.

**VMG\_COMPAT\_ARRAY\_UNDEF\_CLEAR**

True for perls that call *clear* magic when undefining magical arrays.

**VMG\_COMPAT\_HASH\_DELETE\_NOUVAR\_VOID**

True for perls that don’t call *delete* magic when you delete an element from a hash in void context.

**VMG\_COMPAT\_CODE\_COPY\_CLONE**

True for perls that call *copy* magic when a magical closure prototype is cloned.

**VMG\_COMPAT\_GLOB\_GET**

True for perls that call *get* magic for operations on globs.

**VMG\_PERL\_PATCHLEVEL**

The perl patchlevel this module was built with, or 0 for non-debugging perls.

**VMG\_THREADSAFE**

True if and only if this module could have been built with thread-safety features enabled.

**VMG\_FORKSAFE**

True if and only if this module could have been built with fork-safety features enabled. This is always true except on Windows where it is false for perl 5.10.0 and below.

**VMG\_OP\_INFO\_NAME**

Value to pass with `op_info` to get the current op name in the magic callbacks.

**VMG\_OP\_INFO\_OBJECT**

Value to pass with `op_info` to get a `B::OP` object representing the current op in the magic callbacks.

**COOKBOOK****Associate an object to any perl variable**

This technique can be useful for passing user data through limited APIs. It is similar to using inside-out objects, but without the drawback of having to implement a complex destructor.

```
{
    package Magical::UserData;

    use Variable::Magic qw<wizard cast getdata>;

    my $wiz = wizard data => sub { $_[1] };

    sub ud (\[$@%*&]) : lvalue {
        my ($var) = @_;
        my $data = &getdata($var, $wiz);
```

```

    unless (defined $data) {
        $data = \(\my $slot);
        &cast($var, $wiz, $slot)
            or die "Couldn't cast UserData magic onto the variable";
    }
    $$data;
}

{
    BEGIN { *ud = \&Magical::UserData::ud }

    my $cb;
    $cb = sub { print 'Hello, ', ud(&$cb), "!\n" };

    ud(&$cb) = 'world';
    $cb->(); # Hello, world!
}

```

### Recursively cast magic on datastructures

cast can be called from any magical callback, and in particular from data. This allows you to recursively cast magic on datastructures :

```

my $wiz;
$wiz = wizard data => sub {
    my ($var, $depth) = @_;
    $depth ||= 0;
    my $r = ref $var;
    if ($r eq 'ARRAY') {
        &cast((ref() ? $_ : \$_), $wiz, $depth + 1) for @$var;
    } elsif ($r eq 'HASH') {
        &cast((ref() ? $_ : \$_), $wiz, $depth + 1) for values %$var;
    }
    return $depth;
},
free => sub {
    my ($var, $depth) = @_;
    my $r = ref $var;
    print "free $r at depth $depth\n";
    ();
};

{
    my %h = (
        a => [ 1, 2 ],
        b => { c => 3 }
    );
    cast %h, $wiz;
}

```

When %h goes out of scope, this prints something among the lines of :

```

free HASH at depth 0
free HASH at depth 1
free SCALAR at depth 2
free ARRAY at depth 1
free SCALAR at depth 3
free SCALAR at depth 3

```

Of course, this example does nothing with the values that are added after the cast.

### Delayed magic actions

Starting with Variable::Magic 0.58, the return value of the magic callbacks can be used to delay the action until after the original action takes place :

```

my $delayed;
my $delayed_aux = wizard(
    data => sub { $_[1] },
    free => sub {
        my ($target) = $_[1];
        my $target_data = &getdata($target, $delayed);
        local $target_data->{guard} = 1;
        if (ref $target eq 'SCALAR') {
            my $orig = $$target;
            $$target = $target_data->{mangler}->($orig);
        }
        return;
    },
);
$delayed = wizard(
    data => sub {
        return +{ guard => 0, mangler => $_[1] };
    },
    set => sub {
        return if $_[1]->{guard};
        my $token;
        cast $token, $delayed_aux, $_[0];
        return \$token;
    },
);
my $x = 1;
cast $x, $delayed => sub { $_[0] * 2 };
$x = 2;
# $x is now 4
# But note that the delayed action only takes place at the end of the
# current statement :
my @y = ($x = 5, $x);
# $x is now 10, but @y is (5, 5)

```

### PERL MAGIC HISTORY

The places where magic is invoked have changed a bit through perl history. Here is a little list of the most recent ones.

- **5.6.x**  
*p14416* : *copy* and *dup* magic.
- **5.8.9**  
*p28160* : Integration of *p25854* (see below).  
*p32542* : Integration of *p31473* (see below).



- **5.9.3**  
*p25854* : *len* magic is no longer called when pushing an element into a magic array.  
*p26569* : *local* magic.
- **5.9.5**  
*p31064* : Meaningful *uvar* magic.  
*p31473* : *clear* magic was not invoked when undefining an array. The bug is fixed as of this version.
- **5.10.0**  
 Since `PERL_MAGIC_uvar` is uppercased, `hv_magic_check()` triggers *copy* magic on hash stores for (non-tied) hashes that also have *uvar* magic.
- **5.11.x**  
*p32969* : *len* magic is no longer invoked when calling `length` with a magical scalar.  
*p34908* : *len* magic is no longer called when pushing / unshifting an element into a magical array in void context. The push part was already covered by *p25854*.  
*g9cdbc38b* : *len* magic is called again when pushing into a magical array in non-void context.

## EXPORT

The functions “wizard”, “cast”, “getdata” and “dispell” are only exported on request. All of them are exported by the tags `':funcs'` and `':all'`.

All the constants are also only exported on request, either individually or by the tags `':consts'` and `':all'`.

## CAVEATS

In order to hook hash operations with magic, you need at least perl 5.10.0 (see “VMG\_UVAR”).

If you want to store a magic object in the private data slot, you will not be able to recover the magic with “getdata”, since magic is not copied by assignment. You can work around this gotcha by storing a reference to the magic object instead.

If you define a wizard with *free* magic and cast it on itself, it results in a memory cycle, so this destructor will not be called when the wizard is freed.

## DEPENDENCIES

perl 5.8.

A C compiler. This module may happen to build with a C++ compiler as well, but don't rely on it, as no guarantee is made in this regard.

Carp (core since perl 5), XSLoader (since 5.6.0).

## SEE ALSO

`perlguts` and `perlapi` for internal information about magic.

`perltie` and `overload` for other ways of enhancing objects.

## AUTHOR

Vincent Pit, <perl at profvince.com>, <<http://www.profvince.com>>.

You can contact me by mail or on `irc.perl.org` (vincent).

## BUGS

Please report any bugs or feature requests to `bug-variable-magic at rt.cpan.org`, or through the web interface at <<http://rt.cpan.org/NoAuth/ReportBug.html?Queue=Variable-Magic>>. I will be notified, and then you'll automatically be notified of progress on your bug as I make changes.

## SUPPORT

You can find documentation for this module with the `perldoc` command.

perldoc Variable::Magic

**COPYRIGHT & LICENSE**

Copyright 2007,2008,2009,2010,2011,2012,2013,2014,2015,2016,2017 Vincent Pit, all rights reserved.

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