perltie

how to hide an object class in a simple variable

Table of Contents

SYNOPSIS

```
tie VARIABLE, CLASSNAME, LIST
untie VARIABLE
```

DESCRIPTION

Prior to release 5.0 of Perl, a programmer could use *dbmopen()* to magically connect an on-disk database in the standard Unix dbm(3x) format to a **%HASH** in their program. However, their Perl was either built with one particular dbm library or another, but not both, and you couldn't extend this mechanism to other packages or types of variables.

Now you can.

The *tie()* function binds a variable to a class (package) that will provide the implementation for access methods for that variable. Once this magic has been performed, accessing a tied variable automatically triggers method calls in the proper class. All of the complexity of the class is hidden behind magic methods calls. The method names are in ALL CAPS, which is a convention that Perl uses to indicate that they're called implicitly rather than explicitly--just like the *BEGIN()* and *END()* functions.

In the *tie()* call, `VARIABLE` is the name of the variable to be enchanted. `CLASSNAME` is the name of a class implementing objects of the correct type. Any additional arguments in the LIST are passed to the appropriate constructor method for that class--meaning TIESCALAR() , TIEARRAY() , or TIEHASH() . (Typically these are arguments such as might be passed to the *dbminit()* function of C.) The object returned by the ''new'' method is also returned by the *tie()* function, which would be useful if you wanted to access other methods in `CLASSNAME`. (You don't actually have to return a reference to a right ''type'' (e.g. HASH or `CLASSNAME`) so long as it's a properly blessed object.)

Unlike *dbmopen()*, the *tie()* function will not `use` or `require` a module for you--you need to do that explicitly yourself.

Tying Scalars

A class implementing a tied scalar should define the following methods: TIESCALAR, FETCH, STORE, and possible DESTROY.

Let's look at each in turn, using as an example a tie class for scalars that allows the user to do something like:

```
tie $his_speed, 'Nice', getppid();
tie $my_speed,  'Nice', $$;
```

And now whenever either of those variables is accessed, its current system priority is retrieved and returned. If those variables are set, then the process's priority is changed!

We'll use Jarkko Hietaniemi *<Jarkko.Hietaniemi@hut.fi>*'s BSD::Resource class (not included) to access the PRIO_PROCESS, PRIO_MIN, and PRIO_MAX constants from your system, as well as the *getpriority()* and *setpriority()* system calls. Here's the preamble of the class.

```
package Nice;
use Carp;
use BSD::Resource;
```

```
use strict;
$Nice::DEBUG = 0 unless defined $Nice::DEBUG;
```

TIESCALAR classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference to a new scalar (probably anonymous) that it's creating. For example:

```
sub TIESCALAR {
    my $class = shift;
    my $pid = shift || $$; # 0 means me
    if ($pid !~ /^\d+$/) {
        carp "Nice::TieScalar got non-numeric pid $pid" if $^W;
        return undef;
    }
    unless (kill 0 => $pid) { # EPERM or ERSCH, no doubt
        carp "Nice::TieScalar got bad pid $pid: $!" if $^W;
        return undef;
    }
    return bless \$pid => $class;
}
```

This tie class has chosen to return an error rather than raising an exception if its constructor should fail. While this is how *dbmopen()* works, other classes may well not wish to be so forgiving. It checks the global variable **`$^W`** to see whether to emit a bit of noise anyway.

FETCH this

This method will be triggered every time the tied variable is accessed (read). It takes no arguments beyond its self reference, which is the object representing the scalar we're dealing with. Since in this case we're just using a SCALAR ref for the tied scalar object, a simple **$$**self allows the method to get at the real value stored there. In our example below, that real value is the process ID to which we've tied our variable.

```
sub FETCH {
    my $self = shift;
    confess "wrong type" unless ref $self;
    croak "usage error" if @_;
    my $nicety;
    local($!) = 0;
    $nicety = getpriority(PRIO_PROCESS, $$self);
    if ($!) { croak "getpriority failed: $!" }
    return $nicety;
}
```

This time we've decided to blow up (raise an exception) if the renice fails--there's no place for us to return an error otherwise, and it's probably the right thing to do.

STORE this, value

This method will be triggered every time the tied variable is set (assigned). Beyond its self reference, it also expects one (and only one) argument--the new value the user is trying to assign.

```
sub STORE {
    my $self = shift;
```

```
        confess "wrong type" unless ref $self;
        my $new_nicety = shift;
        croak "usage error" if @_;
        if ($new_nicety < PRIO_MIN) {
            carp sprintf
              "WARNING: priority %d less than minimum system priority %d",
                  $new_nicety, PRIO_MIN if $^W;
            $new_nicety = PRIO_MIN;
        }
        if ($new_nicety > PRIO_MAX) {
            carp sprintf
              "WARNING: priority %d greater than maximum system priority %d",
                  $new_nicety, PRIO_MAX if $^W;
            $new_nicety = PRIO_MAX;
        }
        unless (defined setpriority(PRIO_PROCESS, $$self, $new_nicety)) {
            confess "setpriority failed: $!";
        }
        return $new_nicety;
    }
```

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with other object classes, such a method is seldom ncessary, since Perl deallocates its moribund object's memory for you automatically--this isn't C++, you know. We'll use a DESTROY method here for debugging purposes only.

```
    sub DESTROY {
        my $self = shift;
        confess "wrong type" unless ref $self;
        carp "[ Nice::DESTROY pid $$self ]" if $Nice::DEBUG;
    }
```

That's about all there is to it. Actually, it's more than all there.is to it, since we've done a few nice things here for the sake of completeness, robustness, and general aesthetics. Simpler TIESCALAR classes are certainly possible.

Tying Arrays

A class implementing a tied ordinary array should define the following methods: TIEARRAY, FETCH, STORE, and perhaps DESTROY.

WARNING: Tied arrays are *incomplete*. It is also distinctly lacking something for the `$#ARRAY` access (which is hard, as it's an lvalue), as well as the other obvious array functions, like *push()*, *pop()*, *shift()*, *unshift()*, and *splice()*.

For this discussion, we'll implement an array whose indices are fixed at its creation. If you try to access anything beyond those bounds, you'll take an exception. (Well, if you access an individual element; an aggregate assignment would be missed.) For example:

```
require Bounded_Array;
tie @ary, Bounded_Array, 2;
$| = 1;
for $i (0 .. 10) {
    print "setting index $i: ";
    $ary[$i] = 10 * $i;
    $ary[$i] = 10 * $i;
    print "value of elt $i now $ary[$i]\n";
}
```

The preamble code for the class is as follows:

```
package Bounded_Array;
use Carp;
use strict;
```

TIEARRAY classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference through which the new array (probably an anonymous ARRAY ref) will be accessed.

In our example, just to show you that you don't *really* have to return an ARRAY reference, we'll choose a HASH reference to represent our object. A HASH works out well as a generic record type: the `{BOUND}` field will store the maximum bound allowed, and the C<{ARRAY} field will hold the true ARRAY ref. If someone outside the class tries to dereference the object returned (doubtless thinking it an ARRAY ref), they'll blow up. This just goes to show you that you should respect an object's privacy.

```
sub TIEARRAY {
    my $class = shift;
    my $bound = shift;
    confess "usage: tie(\@ary, 'Bounded_Array', max_subscript)"
        if @_ || $bound =~ /\D/;
    return bless {
        BOUND => $bound,
        ARRAY => [],
    } => $class;
}
```

FETCH this, index

This method will be triggered every time an individual element the tied array is accessed (read). It takes one argument beyond its self reference: the index whose value we're trying to fetch.

```
sub FETCH {
  my($self,$idx) = @_;
  if ($idx > $self->{BOUND}) {
    confess "Array OOB: $idx > $self->{BOUND}";
  }
  return $self->{ARRAY}[$idx];
}
```

As you may have noticed, the name of the FETCH method (et al.) is the same for all accesses, even though the constructors differ in names (TIESCALAR vs TIEARRAY). While in theory you could have the same class servicing several tied types, in practice this becomes cumbersome, and it's easiest to simply keep them at one tie type per class.

STORE this, index, value

This method will be triggered every time an element in the tied array is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something and the value we're trying to put there. For example:

```
sub STORE {
  my($self, $idx, $value) = @_;
  print "[STORE $value at $idx]\n" if _debug;
  if ($idx > $self->{BOUND} ) {
    confess "Array OOB: $idx > $self->{BOUND}";
  }
  return $self->{ARRAY}[$idx] = $value;
}
```

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with the sclar tie class, this is almost never needed in a language that does its own garbage collection, so this time we'll just leave it out.

The code we presented at the top of the tied array class accesses many.elements of the array, far more than we've set the bounds to. Therefore, it will blow up once they try to access beyond the 2nd element of **@ary**, as the following output demonstrates:

```
setting index 0: value of elt 0 now 0
setting index 1: value of elt 1 now 10
setting index 2: value of elt 2 now 20
setting index 3: Array OOB: 3 > 2 at Bounded_Array.pm line 39
        Bounded_Array::FETCH called at testba line 12
```

Tying Hashes

As the first Perl data type to be tied (see *dbmopen()*), associative arrays have the most complete and useful *tie()* implementation. A class implementing a tied associative array should define the following methods: TIEHASH is the constructor. FETCH and STORE access the key and value pairs. EXISTS reports whether a key is present in the hash, and DELETE deletes one. CLEAR empties the hash by deleting all the key and value pairs. FIRSTKEY and NEXTKEY implement the *keys()* and *each()* functions to iterate over all the keys. And DESTROY is called when the tied variable is garbage collected.

If this seems like a lot, then feel free to merely inherit from the standard TieHash module for most of your methods, redefining only the interesting ones. See *TieHash* for details.

Remember that Perl distinguishes between a key not existing in the hash, and the key existing in the hash but having a corresponding value of `undef`. The two possibilities can be tested with the *`exists()`*

and *defined()* functions.

Here's an example of a somewhat interesting tied hash class: it gives you a hash representing a particular user's dotfiles. You index into the hash with the name of the file (minus the dot) and you get back that dotfile's contents. For example:

```
use DotFiles;
tie %mydots, DotFiles;
if ( $mydots{profile} =~ /MANPATH/ ||
     $mydots{login}   =~ /MANPATH/ ||
     $mydots{cshrc}   =~ /MANPATH/ )
 {
    print "you seem to set your manpath\n";
}
```

Or here's another sample of using our tied class:

```
tie %him, DotFiles, 'daemon';
foreach $f ( keys %him ) {
    printf "daemon dot file %s is size %d\n",
        $f, length $him{$f};
}
```

In our tied hash DotFiles example, we use a regular hash for the object containing several important fields, of which only the `{LIST}` field will be what the user thinks of as the real hash.

USER
whose dot files this object represents

HOME
where those dotfiles live

CLOBBER
whether we should try to change or remove those dot files

LIST the hash of dotfile names and content mappings

Here's the start of *Dotfiles.pm*:.

```
package DotFiles;
use Carp;
sub whowasi { (caller(1))[3] . '()' }
my $DEBUG = 0;
sub debug { $DEBUG = @_ ? shift : 1 }
```

For our example, we want to able to emit debugging info to help in tracing during development. We keep also one convenience function around internally to help print out warnings; *whowasi()* returns the function name that calls it.

Here are the methods for the DotFiles tied hash.

TIEHASH classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference through which the new object (probably but not necessarily an anonymous hash) will be accessed.

Here's the constructor:

```
sub TIEHASH {
    my $self = shift;
    my $user = shift || $>;
    my $dotdir = shift || '';
    croak "usage: @{[&whowasi]} [USER [DOTDIR]]" if @_;
    $user = getpwuid($user) if $user =~ /^\d+$/;
    my $dir = (getpwnam($user))[7]
            || croak "@{[&whowasi]}: no user $user";
    $dir .= "/$dotdir" if $dotdir;
    my $node = {
        USER    => $user,
        HOME    => $dir,
        LIST    => {},
        CLOBBER => 0,
    };
    opendir(DIR, $dir)
            || croak "@{[&whowasi]}: can't opendir $dir: $!";
    foreach $dot ( grep /^\./ && -f "$dir/$_", readdir(DIR)) {
        $dot =~ s/^\.//;
        $node->{LIST}{$dot} = undef;
    }
    closedir DIR;
    return bless $node => $self;
}
```

It's probably worth mentioning that if you're going to filetest the return values out of a readdir, you'd better prepend the directory in question. Otherwise, since we didn't *chdir()* there, it would have been testing the wrong file.

FETCH this, key

This method will be triggered every time an element in the tied hash is accessed (read). It takes one argument beyond its self reference: the key whose value we're trying to fetch.

Here's the fetch for our DotFiles example.

```
sub FETCH {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $dir = $self->{HOME};
    my $file = "$dir/.$dot";
    unless (exists $self->{LIST}->{$dot} || -f $file) {
        carp "@{[&whowasi]}: no $dot file" if $DEBUG;
        return undef;
    }
    if (defined $self->{LIST}->{$dot}) {
        return $self->{LIST}->{$dot};
    } else {
        return $self->{LIST}->{$dot} = `cat $dir/.$dot`;
    }
```

```
}
```

It was easy to write by having it call the Unix cat(1) command, but it would probably be more portable to open the file manually (and somewhat more efficient). Of course, since dot files are a Unixy concept, we're not that concerned.

STORE this, key, value

This method will be triggered every time an element in the tied hash is set (written). It takes two arguments beyond its self reference: the index at which we're trying to store something, and the value we're trying to put there.

Here in our DotFiles example, we'll be careful not to let them try to overwrite the file unless they've called the *clobber()* method on the original object reference returned by *tie()*.

```
sub STORE {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $value = shift;
    my $file = $self->{HOME} . "/.$dot";
    my $user = $self->{USER};
    croak "@{[&whowasi]}: $file not clobberable"
        unless $self->{CLOBBER};
    open(F, "> $file") || croak "can't open $file: $!";
    print F $value;
    close(F);
}
```

If they wanted to clobber something, they might say:

```
$ob = tie %daemon_dots, 'daemon';
$ob->clobber(1);
$daemon_dots{signature} = "A true daemon\n";
```

Where the clobber method is simply:

```
sub clobber {
    my $self = shift;
    $self->{CLOBBER} = @_ ? shift : 1;
}
```

DELETE this, key

This method is triggered when we remove an element from the hash, typically by using the *delete()* function. Again, we'll be careful to check whether they really want to clobber files.

```
sub DELETE   {
    carp &whowasi if $DEBUG;
    my $self = shift;
    my $dot = shift;
    my $file = $self->{HOME} . "/.$dot";
    croak "@{[&whowasi]}: won't remove file $file"
        unless $self->{CLOBBER};
```

```
        delete $self->{LIST}->{$dot};
        unlink($file) || carp "@{[&whowasi]}: can't unlink $file: $!";
    }
```

CLEAR this

This method is triggered when the whole hash is to be cleared, usually by assigning the empty list to it.

In our example, that would remove all the user's dotfiles! It's such a dangerous thing that they'll have to set CLOBBER to something higher than 1 to make it happen.

```
    sub CLEAR    {
        carp &whowasi if $DEBUG;
        my $self = shift;
        croak "@{[&whowasi]}: won't remove all dotfiles for $self->{USER}"
            unless $self->{CLOBBER} > 1;
        my $dot;
        foreach $dot ( keys %{$self->{LIST}}) {
            $self->DELETE($dot);
        }
    }
```

EXISTS this, key

This method is triggered when the user uses the *exists()* function on a particular hash. In our example, we'll look at the `{LIST}` hash element for this:

```
    sub EXISTS   {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $dot = shift;
        return exists $self->{LIST}->{$dot};
    }
```

FIRSTKEY this

This method will be triggered when the user is going to iterate through the hash, such as via a *keys()* or *each()* call.

```
    sub FIRSTKEY {
        carp &whowasi if $DEBUG;
        my $self = shift;
        my $a = keys %{$self->{LIST}};
        each %{$self->{LIST}}
    }
```

NEXTKEY this, lastkey

This method gets triggered during a *keys()* or *each()* iteration. It has a second argument which is the last key that had been accessed. This is useful if you're carrying about ordering or calling the iterator from more than one sequence, or not really storing things in a hash anywhere.

For our example, we our using a real hash so we'll just do the simple thing, but we'll have to indirect through the LIST field.

```
sub NEXTKEY  {
    carp &whowasi if $DEBUG;
    my $self = shift;
    return each %{ $self->{LIST} }
}
```

DESTROY this

This method is triggered when a tied hash is about to go out of scope. You don't really need it unless you're trying to add debugging or have auxiliary state to clean up. Here's a very simple function:

```
sub DESTROY  {
    carp &whowasi if $DEBUG;
}
```

Note that functions such as *keys()* and *values()* may return huge array.values when used on large objects, like DBM files. You may prefer to use the *each()* function to iterate over such. Example:

```
# print out history file offsets
use NDBM_File;
tie(%HIST, NDBM_File, '/usr/lib/news/history', 1, 0);
while (($key,$val) = each %HIST) {
    print $key, ' = ', unpack('L',$val), "\n";
}
untie(%HIST);
```

Tying FileHandles

This isn't implemented yet. Sorry; maybe someday.

SEE ALSO

See *DB_File* or *Config* for some interesting *tie()* implementations.

BUGS

Tied arrays are *incomplete*. It is also distinctly lacking something for the **$#**ARRAY access (which is hard, as it's an lvalue), as well as the other obvious array functions, like *push()*, *pop()*, *shift()*, *unshift()*, and *splice()*.

AUTHOR

Tom Christiansen