NAME
perltie - how to hide an object class in a simple variable

SYNOPSIS
```
 tie VARIABLE, CLASSNAME, LIST

 $object = tied VARIABLE

 untie VARIABLE
```

DESCRIPTION
Prior to release 5.0 of Perl, a programmer could use *dbmopen()* to connect an on-disk database in the standard Unix *dbm(3x)* format magically 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. 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()*, *TIEHASH()*, or *TIEHANDLE()*. (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.) You can also retrieve a reference to the underlying object using the *tied()* function.

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 possibly UNTIE and/or 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 <*jhi@iki.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::Tie::Scalar got non-numeric pid $pid" if $^W;
return undef;
}

unless (kill 0, $pid) { # EPERM or ERSCH, no doubt
carp "Nice::Tie::Scalar 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. Because in this case we're using just 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. Don't worry about returning a value from STORE; the semantic of assignment returning the assigned value is implemented with FETCH.

```
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: $!";
        }
        }
```

UNTIE this

This method will be triggered when the `untie` occurs. This can be useful if the class needs to know when no further calls will be made. (Except DESTROY of course.) See "The `untie` Gotcha" below for more details.

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 necessary, because 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, because 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, FETCHSIZE, STORESIZE, CLEAR and perhaps UNTIE and/or DESTROY.

FETCHSIZE and STORESIZE are used to provide `$#array` and equivalent `scalar(@array)` access.

The methods POP, PUSH, SHIFT, UNSHIFT, SPLICE, DELETE, and EXISTS are required if the perl operator with the corresponding (but lowercase) name is to operate on the tied array. The **Tie::Array** class can be used as a base class to implement the first five of these in terms of the basic methods above. The default implementations of DELETE and EXISTS in **Tie::Array** simply `croak`.

In addition EXTEND will be called when perl would have pre-extended allocation in a real array.

For this discussion, we'll implement an array whose elements are a fixed size at creation. If you try to create an element larger than the fixed size, you'll take an exception. For example:

```
    use FixedElem_Array;
    tie @array, 'FixedElem_Array', 3;
    $array[0] = 'cat'; # ok.
    $array[1] = 'dogs'; # exception, length('dogs') > 3.
```

The preamble code for the class is as follows:

```
package FixedElem_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 `{ELEMSIZE}` field will store the maximum element size allowed, and the `{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 $elemsize = shift;
if ( @_ || $elemsize =˜ /\D/ ) {
croak "usage: tie ARRAY, '" . __PACKAGE__ . "', elem_size";
}
return bless {
ELEMSIZE => $elemsize,
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 = shift;
my $index = shift;
return $self->{ARRAY}->[$index];
}
```

If a negative array index is used to read from an array, the index will be translated to a positive one internally by calling FETCHSIZE before being passed to FETCH. You may disable this feature by assigning a true value to the variable `$NEGATIVE_INDICES` in the tied array class.

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 keep them at simply 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.

In our example, `undef` is really `$self->{ELEMSIZE}` number of spaces so we have a little more work to do here:

```
sub STORE {
my $self = shift;
my( $index, $value ) = @_;
if ( length $value > $self->{ELEMSIZE} ) {
croak "length of $value is greater than $self->{ELEMSIZE}";
}
# fill in the blanks
$self->EXTEND( $index ) if $index > $self->FETCHSIZE();
# right justify to keep element size for smaller elements
$self->{ARRAY}->[$index] = sprintf "%$self->{ELEMSIZE}s", $value;
}
```

Negative indexes are treated the same as with FETCH.

FETCHSIZE this

Returns the total number of items in the tied array associated with object *this*. (Equivalent to `scalar(@array)`). For example:

```
sub FETCHSIZE {
my $self = shift;
return scalar @{$self->{ARRAY}};
}
```

STORESIZE this, count

Sets the total number of items in the tied array associated with object *this* to be *count*. If this makes the array larger then class's mapping of `undef` should be returned for new positions. If the array becomes smaller then entries beyond count should be deleted.

In our example, 'undef' is really an element containing `$self->{ELEMSIZE}` number of spaces. Observe:

```
sub STORESIZE {
my $self = shift;
my $count = shift;
if ( $count > $self->FETCHSIZE() ) {
foreach ( $count - $self->FETCHSIZE() .. $count ) {
$self->STORE( $_, '' );
}
} elsif ( $count < $self->FETCHSIZE() ) {
foreach ( 0 .. $self->FETCHSIZE() - $count - 2 ) {
$self->POP();
}
}
}
```

EXTEND this, count

Informative call that array is likely to grow to have *count* entries. Can be used to optimize allocation. This method need do nothing.

In our example, we want to make sure there are no blank (`undef`) entries, so `EXTEND` will make use of `STORESIZE` to fill elements as needed:

```
sub EXTEND {
my $self = shift;
my $count = shift;
$self->STORESIZE( $count );
}
```

EXISTS this, key

Verify that the element at index *key* exists in the tied array *this*.

In our example, we will determine that if an element consists of `$self->{ELEMSIZE}` spaces only, it does not exist:

```
sub EXISTS {
my $self = shift;
my $index = shift;
return 0 if ! defined $self->{ARRAY}->[$index] ||
$self->{ARRAY}->[$index] eq ' ' x $self->{ELEMSIZE};
return 1;
}
```

DELETE this, key

Delete the element at index *key* from the tied array *this*.

In our example, a deleted item is `$self->{ELEMSIZE}` spaces:

```
sub DELETE {
my $self = shift;
my $index = shift;
return $self->STORE( $index, '' );
}
```

CLEAR this

Clear (remove, delete, ...) all values from the tied array associated with object *this*. For example:

```
sub CLEAR {
my $self = shift;
return $self->{ARRAY} = [];
}
```

PUSH this, LIST

Append elements of *LIST* to the array. For example:

```
sub PUSH {
my $self = shift;
my @list = @_;
my $last = $self->FETCHSIZE();
$self->STORE( $last + $_, $list[$_] ) foreach 0 .. $#list;
return $self->FETCHSIZE();
}
```

POP this

Remove last element of the array and return it. For example:

```
sub POP {
my $self = shift;
return pop @{$self->{ARRAY}};
}
```

SHIFT this

Remove the first element of the array (shifting other elements down) and return it. For example:

```
sub SHIFT {
my $self = shift;
return shift @{$self->{ARRAY}};
}
```

UNSHIFT this, LIST

Insert LIST elements at the beginning of the array, moving existing elements up to make room. For example:

```
sub UNSHIFT {
my $self = shift;
my @list = @_;
my $size = scalar( @list );
# make room for our list
@{$self->{ARRAY}}[ $size .. $#{$self->{ARRAY}} + $size ]
= @{$self->{ARRAY}};
$self->STORE( $_, $list[$_] ) foreach 0 .. $#list;
}
```

SPLICE this, offset, length, LIST

Perform the equivalent of `splice` on the array.

offset is optional and defaults to zero, negative values count back from the end of the array.

length is optional and defaults to rest of the array.

LIST may be empty.

Returns a list of the original *length* elements at *offset*.

In our example, we'll use a little shortcut if there is a *LIST*:

```
sub SPLICE {
my $self = shift;
my $offset = shift || 0;
my $length = shift || $self->FETCHSIZE() - $offset;
my @list = ();
if ( @_ ) {
tie @list, __PACKAGE__, $self->{ELEMSIZE};
@list = @_;
}
return splice @{$self->{ARRAY}}, $offset, $length, @list;
}
```

UNTIE this

Will be called when `untie` happens. (See "The `untie` Gotcha" below.)

DESTROY this

This method will be triggered when the tied variable needs to be destructed. As with the scalar 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.

Tying Hashes

Hashes were the first Perl data type to be tied (see *dbmopen()*). A class implementing a tied hash 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. SCALAR is triggered when the tied hash is evaluated in scalar context. UNTIE is called when `untie` happens, and DESTROY is called when the tied variable is garbage collected.

If this seems like a lot, then feel free to inherit from merely the standard Tie::StdHash module for most of your methods, redefining only the interesting ones. See Tie::Hash 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 dot files. You index into the hash with the name of the file (minus the dot) and you get back that dot file's contents. For example:

```
use DotFiles;
tie %dot, 'DotFiles';
if ( $dot{profile} =~ /MANPATH/ ||
$dot{login} =~ /MANPATH/ ||
$dot{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 dot files live

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

LIST
: the hash of dot file 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 be 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, because 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, because 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(my $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";
```

Another way to lay hands on a reference to the underlying object is to use the *tied()* function, so they might alternately have set clobber using:

```
tie %daemon_dots, 'daemon';
tied(%daemon_dots)->clobber(1);
```

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};
my $success = unlink($file);
carp "@{[&whowasi]}: can't unlink $file: $!" unless $success;
$success;
}
```

The value returned by DELETE becomes the return value of the call to *delete()*. If you want to emulate the normal behavior of *delete()*, you should return whatever FETCH would have returned for this key. In this example, we have chosen instead to return a value which tells the caller whether the file was successfully deleted.

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 dot files! 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 dot files 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()*, *values()*, or *each()* call.

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

FIRSTKEY is always called in scalar context and it should just return the first key. *values()*, and *each()* in list context, will call FETCH for the returned keys.

NEXTKEY this, lastkey

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

NEXTKEY is always called in scalar context and it should just return the next key. *values()*, and *each()* in list context, will call FETCH for the returned keys.

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

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

SCALAR this

This is called when the hash is evaluated in scalar context. In order to mimic the behaviour of untied hashes, this method should return a false value when the tied hash is considered empty. If this method does not exist, perl will make some educated guesses and return true when the hash is inside an iteration. If this isn't the case, FIRSTKEY is called, and the result will be a false value if FIRSTKEY returns the empty list, true otherwise.

However, you should **not** blindly rely on perl always doing the right thing. Particularly, perl will mistakenly return true when you clear the hash by repeatedly calling DELETE until it is empty. You are therefore advised to supply your own SCALAR method when you want to be absolutely sure that your hash behaves nicely in scalar context.

In our example we can just call `scalar` on the underlying hash referenced by `$self->{LIST}`:

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

UNTIE this

This is called when `untie` occurs. See "The `untie` Gotcha" below.

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 lists 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 is partially implemented now.

A class implementing a tied filehandle should define the following methods: TIEHANDLE, at least one of PRINT, PRINTF, WRITE, READLINE, GETC, READ, and possibly CLOSE, UNTIE and DESTROY. The class can also provide: BINMODE, OPEN, EOF, FILENO, SEEK, TELL - if the corresponding perl operators are used on the handle.

When STDERR is tied, its PRINT method will be called to issue warnings and error messages. This feature is temporarily disabled during the call, which means you can use `warn()` inside PRINT without starting a recursive loop. And just like `__WARN__` and `__DIE__` handlers, STDERR's PRINT method may be called to report parser errors, so the caveats mentioned under "%SIG" in perlvar(1) apply.

All of this is especially useful when perl is embedded in some other program, where output to STDOUT and

STDERR may have to be redirected in some special way. See nvi and the Apache module for examples.

When tying a handle, the first argument to `tie` should begin with an asterisk. So, if you are tying STDOUT, use `*STDOUT`. If you have assigned it to a scalar variable, say `$handle`, use `*$handle`. `tie $handle` ties the scalar variable `$handle`, not the handle inside it.

In our example we're going to create a shouting handle.

```
package Shout;
```

TIEHANDLE classname, LIST

This is the constructor for the class. That means it is expected to return a blessed reference of some sort. The reference can be used to hold some internal information.

```
sub TIEHANDLE { print "<shout>\n"; my $i; bless \$i, shift }
```

WRITE this, LIST

This method will be called when the handle is written to via the `syswrite` function.

```
sub WRITE {
$r = shift;
my($buf,$len,$offset) = @_;
print "WRITE called, \$buf=$buf, \$len=$len, \$offset=$offset";
}
```

PRINT this, LIST

This method will be triggered every time the tied handle is printed to with the `print()` or `say()` functions. Beyond its self reference it also expects the list that was passed to the print function.

```
sub PRINT { $r = shift; $$r++; print join($,,map(uc($_),@_)),$\ }
```

`say()` acts just like `print()` except $\ will be localized to `\n` so you need do nothing special to handle `say()` in `PRINT()`.

PRINTF this, LIST

This method will be triggered every time the tied handle is printed to with the `printf()` function. Beyond its self reference it also expects the format and list that was passed to the printf function.

```
sub PRINTF {
shift;
my $fmt = shift;
print sprintf($fmt, @_);
}
```

READ this, LIST

This method will be called when the handle is read from via the `read` or `sysread` functions.

```
sub READ {
my $self = shift;
my $bufref = \$_[0];
my(undef,$len,$offset) = @_;
print "READ called, \$buf=$bufref, \$len=$len, \$offset=$offset";
# add to $$bufref, set $len to number of characters read
$len;
}
```

READLINE this

This method is called when the handle is read via `<HANDLE>` or `readline HANDLE`.

As per `readline`, in scalar context it should return the next line, or `undef` for no more data. In list context it should return all remaining lines, or an empty list for no more data. The strings returned should include the input record separator `$/` (see perlvar), unless it is `undef` (which means "slurp" mode).

```
sub READLINE {
my $r = shift;
if (wantarray) {
return ("all remaining\n",
"lines up\n",
"to eof\n");
} else {
return "READLINE called " . ++$$r . " times\n";
}
}
```

GETC this

This method will be called when the `getc` function is called.

```
sub GETC { print "Don't GETC, Get Perl"; return "a"; }
```

EOF this

This method will be called when the `eof` function is called.

Starting with Perl 5.12, an additional integer parameter will be passed. It will be zero if `eof` is called without parameter; `1` if `eof` is given a filehandle as a parameter, e.g. `eof(FH)`; and `2` in the very special case that the tied filehandle is `ARGV` and `eof` is called with an empty parameter list, e.g. `eof()`.

```
sub EOF { not length $stringbuf }
```

CLOSE this

This method will be called when the handle is closed via the `close` function.

```
sub CLOSE { print "CLOSE called.\n" }
```

UNTIE this

As with the other types of ties, this method will be called when `untie` happens. It may be appropriate to ‘‘auto CLOSE’’ when this occurs. See "The `untie` Gotcha" below.

DESTROY this

As with the other types of ties, this method will be called when the tied handle is about to be destroyed. This is useful for debugging and possibly cleaning up.

```
sub DESTROY { print "</shout>\n" }
```

Here’s how to use our little example:

```
tie(*FOO,'Shout');
print FOO "hello\n";
$a = 4; $b = 6;
print FOO $a, " plus ", $b, " equals ", $a + $b, "\n";
print <FOO>;
```

UNTIE this

You can define for all tie types an UNTIE method that will be called at *untie()*. See "The `untie` Gotcha" below.

The `untie` Gotcha

If you intend making use of the object returned from either *tie()* or *tied()*, and if the tie’s target class defines a destructor, there is a subtle gotcha you *must* guard against.

As setup, consider this (admittedly rather contrived) example of a tie; all it does is use a file to keep a log of the values assigned to a scalar.

```
package Remember;

use strict;
use warnings;
use IO::File;

sub TIESCALAR {
my $class = shift;
my $filename = shift;
my $handle = IO::File->new( "> $filename" )
or die "Cannot open $filename: $!\n";

print $handle "The Start\n";
bless {FH => $handle, Value => 0}, $class;
}

sub FETCH {
my $self = shift;
return $self->{Value};
}

sub STORE {
my $self = shift;
my $value = shift;
my $handle = $self->{FH};
print $handle "$value\n";
$self->{Value} = $value;
}

sub DESTROY {
my $self = shift;
my $handle = $self->{FH};
print $handle "The End\n";
close $handle;
}

1;
```

Here is an example that makes use of this tie:

```
use strict;
use Remember;

my $fred;
tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

This is the output when it is executed:

```
The Start
1
4
5
The End
```

So far so good. Those of you who have been paying attention will have spotted that the tied object hasn't been used so far. So lets add an extra method to the Remember class to allow comments to be included in the file; say, something like this:

```
sub comment {
my $self = shift;
my $text = shift;
my $handle = $self->{FH};
print $handle $text, "\n";
}
```

And here is the previous example modified to use the `comment` method (which requires the tied object):

```
use strict;
use Remember;

my ($fred, $x);
$x = tie $fred, 'Remember', 'myfile.txt';
$fred = 1;
$fred = 4;
comment $x "changing...";
$fred = 5;
untie $fred;
system "cat myfile.txt";
```

When this code is executed there is no output. Here's why:

When a variable is tied, it is associated with the object which is the return value of the TIESCALAR, TIEARRAY, or TIEHASH function. This object normally has only one reference, namely, the implicit reference from the tied variable. When *untie()* is called, that reference is destroyed. Then, as in the first example above, the object's destructor (DESTROY) is called, which is normal for objects that have no more valid references; and thus the file is closed.

In the second example, however, we have stored another reference to the tied object in `$x`. That means that when *untie()* gets called there will still be a valid reference to the object in existence, so the destructor is not called at that time, and thus the file is not closed. The reason there is no output is because the file buffers have not been flushed to disk.

Now that you know what the problem is, what can you do to avoid it? Prior to the introduction of the optional UNTIE method the only way was the good old `-w` flag. Which will spot any instances where you call *untie()* and there are still valid references to the tied object. If the second script above this near the top `use warnings 'untie'`or w as run with the `-w` flag, Perl prints this warning message:

```
untie attempted while 1 inner references still exist
```

To get the script to work properly and silence the warning make sure there are no valid references to the tied object *before untie()* is called:

```
undef $x;
untie $fred;
```

Now that UNTIE exists the class designer can decide which parts of the class functionality are really associated with `untie` and which with the object being destroyed. What makes sense for a given class depends on whether the inner references are being kept so that non-tie-related methods can be called on the object. But in most cases it probably makes sense to move the functionality that would have been in DESTROY to the UNTIE method.

If the UNTIE method exists then the warning above does not occur. Instead the UNTIE method is passed the count of ''extra'' references and can issue its own warning if appropriate. e.g. to replicate the no UNTIE case this method can be used:

```
sub UNTIE
{
my ($obj,$count) = @_;
carp "untie attempted while $count inner references still exist"
if $count;
}
```

SEE ALSO

See DB_File or Config for some interesting *tie()* implementations. A good starting point for many *tie()* implementations is with one of the modules Tie::Scalar, Tie::Array, Tie::Hash, or Tie::Handle.

BUGS

The bucket usage information provided by `scalar(%hash)` is not available. What this means is that using `%tied_hash` in boolean context doesn't work right (currently this always tests false, regardless of whether the hash is empty or hash elements).

Localizing tied arrays or hashes does not work. After exiting the scope the arrays or the hashes are not restored.

Counting the number of entries in a hash via `scalar(keys(%hash))` or `scalar(values(%hash))` is inefficient since it needs to iterate through all the entries with FIRSTKEY/NEXTKEY.

Tied hash/array slices cause multiple FETCH/STORE pairs, there are no tie methods for slice operations.

You cannot easily tie a multilevel data structure (such as a hash of hashes) to a dbm file. The first problem is that all but GDBM and Berkeley DB have size limitations, but beyond that, you also have problems with how references are to be represented on disk. One module that does attempt to address this need is DBM::Deep. Check your nearest CPAN site as described in perlmodlib(1) for source code. Note that despite its name, DBM::Deep does not use dbm. Another earlier attempt at solving the problem is MLDBM, which is also available on the CPAN, but which has some fairly serious limitations.

Tied filehandles are still incomplete. *sysopen()*, *truncate()*, *flock()*, *fcntl()*, *stat()* and -X can't currently be trapped.

AUTHOR

Tom Christiansen

TIEHANDLE by Sven Verdoolaege <*skimo@dns.ufsia.ac.be*> and Doug MacEachern <*dougm@osf.org*>

UNTIE by Nick Ing-Simmons <*nick@ing-simmons.net*>

SCALAR by Tassilo von Parseval <*tassilo.von.parseval@rwth-aachen.de*>

Tying Arrays by Casey West <*casey@geeknest.com*>