Tied Variables with Perl

Tying variables is a Perl technique which even advanced exponents of the language often ignore. The reason for this is that tying variables can subtly alter what appears to be the fundamental behaviour of the language itself, and without careful thought to program implementation, this can make scripts harder to understand and maintain. On the other hand, when used in suitable circumstances, tying variables can result in simple and convenient ways to implement application logic which would otherwise take considerable effort.

This article describes what a tied variable is, and uses a few examples to demonstrate what can be achieved using them.

What is a Tied Variable?

The name "tied variable" comes from the idea of "tying" a new set of behaviours to a variable. But hold on a moment, you might say, surely the default set of behaviours are well understood and thoroughly predicable? You assign a value to a variable, then get that same value back the next time you read from it. What other behaviour pattern could possibly be more desirable?

Well, quite a few actually. Firstly, the simplistic view of a variable's behaviour outlined above only applies to single value, scalar variables. With arrays you can certainly assign and read values of specific elements, plus you can splice and delete sections of the array, you can shift it, manage it as a stack using the push and pop commands and so on. Hash variables have other abilities, including providing lists of keys and values, and telling you if a certain key exists or not.

A variable of any type can be tied to a Perl object which can include code which overrides any or all of these standard behaviours. Such an object becomes the device which controls everything to do with the variable.

Why might such trickery be useful? Well, as an example, suppose a developer wants to store a value in a certain format. He might choose to call a formatting function each time a new value is assigned to the variable, but if the variable is used frequently, that might clutter the code somewhat. An alternative might be to put the formatting code in an object, then tie the variable to that object. Now, whenever the variable is read, the value stored in it is returned by the underlying object in the correct format. The code which uses the variable needn't run checks on the value because the tied variable is guaranteed to return its value in the right format.

This is a very simple example, and in most cases there are superior ways of handling simple examples. What makes tied variables useful, however, is the way they can occasionally be used to solve unusual and complex problems in very simple way.

Example 1 - A Tied Scalar

The object to which a variable gets tied has two responsibilities. Firstly, it has to store the data the variable is to hold. If a variable is assigned the string "Dear Sir" then the string "Dear Sir" has to be stored somewhere. It is up to the object to hold that string itself. Secondly, the object has to provide a set of methods which define the new variable behaviour. These methods have predefined names, sometimes referred to in Perl circles as "magic" names. For example, an object which can be tied to a scalar variable should contain (at least) the three magic methods `TIESCALAR`, `STORE` and `FETCH`. Note the use of uppercase, which is a Perl convention to imply something has a special or magic meaning.

The TIESCALAR method is called automatically when the object is tied to a variable. Think of it as a constructor. It is generally used to set up the internal storage of the object and initialise whatever else is needed to support the tied variable.
The STORE method is called automatically when a value is assigned to the tied variable. It is passed a parameter which holds the value the program wants stored. The STORE routine can examine that value and decide whether to store it in the object, alter it, ignore it, throw an error, or do something else.
The FETCH method is called automatically when the contents of the variable are read. The FETCH method would normally return the value the object has stored, but it can also return a modified version of it, or return something completely different.
Listing 1 shows a simple example of tying a scalar variable. The variable's value string is stored internally to the tying object, and when the string is read from the variable, the value returned is given in either all upper case or all lower case.

```
#!/usr/bin/perl -w
use strict;

package CaseControl;
use Carp;

sub TIESCALAR
  {
    my $class = shift;

    my %controlHash;
    $controlHash{internalValue} = undef;
    $controlHash{returnForm}    = "UC";

    return bless \%controlHash, $class;
  }

sub STORE
  {
    my( $self, $value ) = @_;

    $self->{internalValue} = $value;
  }

sub FETCH
  {
    my $self  = shift;

    if( $self->{returnForm} eq "UC" ) {
      return uc($self->{internalValue});
    } else {
      return lc($self->{internalValue});
    }
  }

sub setCase
  {
    my( $self, $newCase ) = @_;

    croak "setCase requires parameter of \"UC\" or \"LC\""
      if( $newCase !~ /^UC|LC$/ );

    $self->{returnForm} = $newCase;
```

```
  }


package main;

tie my $twister, "CaseControl";

$twister = "Black Bugs Blood\n";
(tied $twister)->setCase("LC");
print $twister;
(tied $twister)->setCase("UC");
print $twister;
```

The `TIESCALAR` method creates a hash which the object uses to store both the string currently assigned to the variable and an indicator of whether the string should be returned in upper case or lower case. This hash is blessed, and so becomes a Perl object.
The `STORE` method just stores the provided string in the object. The `FETCH` method looks at the value in the object which indicates whether the string should be returned as upper case or lower case, makes the appropriate adjustment to the value of the stored string, then returns the result.
The object tied to a variable can be retrieved at any point using the command `tied $variable`. This is very useful, since it allows direct access to additional methods of the object which can control its behaviour. In this example, I call the `setCase` method to change whether I want the string to always come back in upper case or lower case.

Example 2 - A Tied Array

Tying scalar variables is simple. Let's have a look at an example which uses a tied array. The object which gets tied to an array variable needs to provide code which Perl can use to store and fetch values, as for scalars, but it also needs to provide a few other things. Most useful of these are the `STORESIZE` and `FETCHSIZE` methods it provides so Perl can find values for the `$#array` and `scalar(@array)` operations. Note that any unused methods in the tying interface don't actually have to be supplied. If the program never uses the `scalar()` operator on a tied array, the tying object doesn't need to provide a `FETCHSIZE` method.
Listing 2 shows an example program which uses an object tied to one array to update a second array with the same values as the tied array. That is, if you set element `[0]` of the tied array to "hello world", element `[0]` of the second array is automatically assigned that value as well. This little program is not really useful except as a demonstration, but this general approach can be used when two data sets need to be kept in sync with one another.

```
#!/usr/bin/perl -w
use strict;

package ArrayUpdater;
use Carp;

sub TIEARRAY
  {
    my $self    = shift;
    my $destRef = shift;
```

```
    my %internalHash;
    $internalHash{array}   = ();
    $internalHash{destRef} = $destRef;

    return bless \%internalHash, $self;
  }

sub FETCH
  {
    my $self  = shift;
    my $place = shift;

    return $self->{array}[$place];
  }

sub STORE
  {
    my $self  = shift;
    my $place = shift;
    my $value = shift;

    $self->{array}[$place] = $value;

    my $destRef = $self->{destRef};
    ${@$destRef}[$place] = $value;
  }


package main;

tie my @source, "ArrayUpdater", \my @destination;

$source[0] = "A";
$source[1] = "B";
$source[2] = "C";

print $source[0],     $source[1],     $source[2],     "\n";
print $destination[0],$destination[1],$destination[2],"\n";
```

Here, the `TIEARRAY` method takes an extra parameter, which is a reference to the data set which needs to be kept in sync with the tied data set. This reference, together with the data the tied array has to hold, is stored internally in the object. The `FETCH` method simply returns the tied array data as normal. The `STORE` method is the interesting one. Here, the tied data is stored as normal, then the stored value for the alternate data set is dereferenced so its data can be updated in the same way as the tied data.
As can be seen in the main code, setting values in the source data set now automatically sets the same values in the destination data set.

Example 3 - A Tied Hash

My final example shows how a hash variable can be tied to some code which keeps a graphical representation of some stored data up to date. Tying a hash is much the same as tying an array, only with a few more magic methods required which control key looping structures such as Perl's built in `keys`, `values` and `each` functions.
By now the number of magic methods required to support the data structure is becoming quite large, which means there's a lot of code fill in. It is frequently noted that in many respects the default behaviour of the tied variable would actually be

perfectly adequate, so Perl provides a set of library modules to help out. This example uses the `Tie::StdHash` module.

`Tie::StdHash` is a class which provides an implementation of the normal behaviour of a Perl hash variable. If you create a hash and tie it to the `Tie::StdHash` class, the result will be a hash variable which behaves perfectly normally. The idea, of course, is to take the `Tie::StdHash` class and reimplement the parts of it where you want the standard behaviour to change. This leaves a class which will make a tied variable's behaviour complete in all respects, but different in those which matter to the application.

```
#!/usr/bin/perl -w
use strict;

package DisplayYear;
use Tie::Hash;
our @ISA = "Tie::StdHash";

sub updateDisplay
  {
    my $self = shift;

    system("clear");
    foreach my $key qw(Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov
Dec) {
      print $key, " | ", ("#" x $self->{$key}), "\n";
    }
  }

sub TIEHASH
  {
    my $class = shift;
    my %yearData = ( Jan => 0,
                     Feb => 0,
                     Mar => 0,
                     Apr => 0,
                     May => 0,
                     Jun => 0,
                     Jul => 0,
                     Aug => 0,
                     Sep => 0,
                     Oct => 0,
                     Nov => 0,
                     Dec => 0 );

    return bless \%yearData, $class;
  }

sub STORE
  {
    my( $self, $key, $value ) = @_;

    $self->{$key} = $value;

    $self->updateDisplay() if( grep( /$key/, (qw(Jan Feb Mar Apr May
Jun
                                        Jul Aug Sep Oct Nov Dec)) ) );
  }
```

```
package main;

tie my %_2004Hash, "DisplayYear";

while( 1 ) {
  my $month = (qw(Jan Feb Mar Apr May Jun
                  Jul Aug Sep Oct Nov Dec))[rand()*12];
  $_2004Hash{$month} = rand()*25;
  select( undef, undef, undef, 0.1 );
}
```

The class populates the hash with a set of values for each month of the year, then ensures that when a new value is entered into any of those hash keys the new data is shown on screen. The display update is entirely automatic; as soon as a new value is entered into the data set the display is updated. Students of computer science will immediately recognise this approach as a simple example of an Observer pattern. There is no practical limit to what can be achieved inside the methods of a tying class. If this example wanted to update a complex Graphical User Interface with the new information, or push it down a socket or out to a file, it could do so without problem.

The IxHash Module

A often useful example of ties at work is found in the `Tie::IxHash` module from the CPAN archive. Normally the order in which a set of key/value pairs are inserted into a hash is lost. When the keys are extracted using a loop they come out in an essentially random order. Most often this is the desired behaviour since keys are normally sorted into alphabetical order or something before being used. Sometimes, however, the order in which the values are inserted has some significance, and it makes sense for the application to pull the keys out in that same order. This is where the `Tie::IxHash` module comes in.
`Tie::IxHash` is a class which can be tied to a hash in order for the hash to remember the order its keys are inserted. Consider this simple example from the Perl command line. First, the normal behaviour:

```
>perl -w
%s = ();
$s{xyz} = "1";
$s{abc} = "2";
$s{qwe} = "3";
foreach $f (keys %s) { print $f, "\n" }
abc
qwe
xyz
```

Notice how the `(keys %s)` function is returning the keys in no particular order. Now look at this example:

```
>perl -w
use Tie::IxHash;
%s = ();
tie %s, "Tie::IxHash";
$s{xyz} = "1";
$s{abc} = "2";
$s{qwe} = "3";
foreach $f (keys %s) { print $f, "\n" }
```

```
xyz
abc
qwe
```

Once the hash is tied to the `Tie::IxHash` class, the keys are returned in the same order as they are inserted. Internally the `Tie::IxHash` class keeps an array which holds the order the keys in which were inserted. When the list of keys is requested through the magic `NEXTKEY` method, it consults this internal array and returns the keys in the order it describes.
This is a good example of when a tied variable can subtly alter its behaviour from the normal in such a way that it makes life a lot easier for the developer.

Other Tying Modules in CPAN

`Tie::IxHash` isn't the only example of a module on CPAN which uses ties. There are plenty more which use ties to good and or interesting effect.
Consider the `Tie::F`ile and `Tie::CSV_File` modules. Both of these put a tied array variable in front of text file access, with array elements corresponding to line numbers in the text. Reading from a particular element of an array actually gets you a line from a text file, and writing to an element writes the data into the file. With the `Tie::CSV_File` module, each line is represented by an array with one element for each column in the comma separated data.
`Tie::Syslog` is a module which ties not a scalar, array or hash to an object, but a file handle. This technique allows the developer to reimplement the methods which are used to open, close, read and write to the file, plus a whole lot more. The `Tie::Syslog` implementation defines a tied object which ensures that whatever is written to the file handle actually ends up in the system log.
The `Tie::DBI` module ties a hash onto a SQL database such that the hash is keyed on a chosen database column. When an entry from the hash is accessed, what actually gets returned or updated is the data in the appropriate record in the database. The tied object, of course, uses Perl's DBI module to generate, issue and interpret the appropriate SQL commands which control the database behind the scenes.
Careful consideration should be given to the use of most of these modules, and many others which use ties. While many uses of ties are very clever, and many appear to offer very simple interfaces and implementations of often used patterns, it is frequently the case that a normal procedural- or object-based approach can solve a problem in a much more readable and maintainable way. The reasoning for using ties should always be that they hide complexity, and not because they do something rather clever.

Conclusion

Tied variables are one of those handy features of Perl which just occasionally make a seemingly tricky problem very simple to solve. Ties are not used that often in normal programming since they tend to hide what is really going on and make code a lot harder to follow. A good part of the skill of using tied variables is recognising when to use them and when to employ a more regular solution. They are worth understanding, though, because when they are appropriate, they are often by far the best answer to the problem.