DAVID BLANK-EDELMAN

practical Perl tools: tie me up, tie me down (part 1)

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the book *Perl for System Administration* (O'Reilly, 2000). He has spent the past 20 years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and is one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

WITH APOLOGIES TO PEDRO ALMODÓVAR

and to the kink community readers who will be disappointed when they realize this isn't the column they were hoping for, I'd like to dive deeply into one of my favorite Perl features: the tie(). After reading a bit on this topic, I suspect you'll either have your jaw on the floor or will have slapped your forehead in disgust at those wacky Perl people (what will they think of next?).

Let's start with some basic background. Once upon a time, relatively early in Perl's history, there existed a function called dbmopen(). dbmopen() made a hash variable special by tying it to an on-disk database backend. Usually a hash variable keeps its keys and values in memory, but when dbmopen'd the data would actually live in an {n,s,g}dbm or Berkeley DB database on disk. Accessing the data would cause a transparent fetch or store of the data from or to the database without any effort on the programmer's part.

This meant three great things:

- Much larger data sets could be used because all of the data didn't all have to live in memory at once.
- It was trivial to have the data persist after the program had quit.
- All you needed to know was the usual hash semantics; no advanced fiddling or faddling was necessary.

Zoom forward in time to Perl 5. Perl 5 added a tie() operator to the language which allowed for the same sort of magic to be applied to other kinds of variables. It also abstracted the mechanism even further, such that you could tie more than just a database to a variable. What sorts of things could be used? That's where the jaw dropping starts. This two-part column will give you a taste of the amazing things that have been done with this simple concept and how to use it for your own ideas.

Before we get to the fireworks, I feel compelled to mention that not everyone is enamored with tie(). For instance, in *Perl Best Practices*, the Damian Conway book mentioned in this column before, he says:

> Don't tie variables or filehandles. . . . Tied variables make any code that uses them less maintainable, because they make normal variable operations behave in unexpected, non-standard ways.

Using my most cogent debate skills honed in elementary school, my rebuttal is, "Yes, I definitely agree with this sentiment. But I don't care."

Just to be clear, I'm all for maintainable Perl code (and have actually written some in my time), but I think trade-off can be worth it. This particular abstraction is too powerful to give up just because it has the potential to impact maintainability. An even more compelling argument for me is the amount of creativity this particular feature of the language has unleashed in the Perl community.

In the interests of self-disclosure, I should mention that these may be the words of someone addicted to the power of tie(), so you should make up your own mind about what is important to you before you go down this path. In next issue's column on this topic we'll talk about how to implement our own tie()-based code. At that time we'll also discuss the recommended alternatives to tie(), just so you have all of the tools available when making that decision.

So let's jump on the bus and take the first part of a whirlwind tour of some of the more interesting things I've seen done with tie().

More Complex Backends

Early on we mentioned that tie() had its origins in a mechanism for storing and retrieving values from a simple database. An easy evolutionary step from that notion is the ability to retrieve information from a more complex backend. For example, let's say we had a table in a relational database of network hosts that looked like this:

ether (primary key)	Name	ipaddr
00:16:cb:b7:c8:81	Omphaloskepsis	192.168.0.1
00:04:E2:07:AC:17	Dave	192.168.0.4
00:0C:F2:24:9A:45	Otherdave	192.168.0.7

With Tie::DBI, we could write the following:

```
use Tie::DBI;
tie my %hosts, 'Tie::DBI', {
        table   => 'hosts',
        key     => 'ether',
        CLOBBER => 1   # allows read-write access to database
};
```

Now we have a hash called %hosts whose keys correspond to the primary key column (ether) of the database. This means that:

```
print join ("\n",keys %hosts);
```

will print a list of the Ethernet addresses stored in the database. If we use any of those keys to retrieve a value from %hosts, we get back a reference to an anonymous hash containing that record's information. To see an example, we could add these lines:

```
use Data::Dumper;
print Dumper($hosts{'00:04:E2:07:AC:17'});
```

and this would yield:

```
$VAR1 = {
        'ether' => '00:04:E2:07:AC:17',
        'name' => 'dave',
        'ipaddr' => '192.168.0.4'
};
```

To access an individual field, the syntax is your standard hash of a hash syntax:

```
print $hosts{'00:04:E2:07:AC:17'}->{'name'},"\n";
                # arrow not strictly necessary
```

We could change the data in the database (providing the CLOBBER flag is set at tie() time) with a plain ol' assignment operation:

```
$hosts{'00:16:cb:b7:c8:81'}->{'name'} = "shouldhavebeendave";
```

If we put standard databases aside for a moment, it is worth noting that people have applied the same principle to other less obvious backends, for instance:

```
use Tie::DNS;

tie my %resolve,'Tie::DNS';
print $dns{'example.com'},"\n"; # prints "192.0.34.166"
```

In the second part of this series we'll go over the steps necessary to use another backend of your choosing.

Transformers

No, I'm not talking about the robots from Hasbro that turn into trucks. This is a class of modules where the data stored in a tie()'d variable is transformed during the retrieval of that value. Here's a simple example:

```
use Tie::Comma;                 # loads a magic tied hash called %comma

my $a = "12345678";
print "$comma{$a}\n";           # prints 12,345,678
print "$comma{$a,2}\n";         # prints 12,345,678.00
```

Here we've created a magical hash that transforms the format of a value simply by looking that value up in the hash. Yes, this looks a bit like a fancy printf() statement, but I'm just trying to limber you up. Shortly we'll get into some very strange territory around variable retrieval, so I want you prepared for when things start to deviate from the usual understanding of reality.

Fancy Lookups

Normally we don't think very hard about the actual retrieval process with hashes. We've always been taught that hashes store a set of key/value pairs. To retrieve a certain value, you need to present the hash with the unique key associated with that value (hence the term *associative arrays*). But what if we could play around a bit with this assumption?

What if, for instance, we could make those lookups be case-insensitive? Imagine you had a hash with the following in it:

```
my %banks = ('sasquatch trust' => 3000);
```

To get the value from this hash for that bank, you have to say $banks{'sasquatch trust'}; $banks{'Sasquatch Trust'} doesn't work. However, if you use the Tie::CPHash module, it will:

```
use Tie::CPHash;

tie my %banks, 'Tie::CPHash';
%banks = ('sasquatch trust' => 3000);

print $banks{'sasquatch trust'},"\n";       #prints 3000
print $banks{'Sasquatch Trust'},"\n";       #prints 3000
```

You could even use a key based on a trendy, creative capitalization scheme:

```
print $banks{'saSqUatCh Trust'},"\n"; prints 3000
```

Tie::CPHash retains the original capitalization of the key when first stored in the hash and makes that information available if you need it.

Case-insensitive lookup is peanuts compared to our next example. What if you could store a date range for a key? Tie::RangeHash lets you do that (and more):

```
use Tie::RangeHash;

tie my %semester,'Tie::RangeHash';
$semester{'2006-09-06,2006-12-15'} = 'Fall semester';
$semester{'2007-01-08,2007-04-27'} = 'Spring semester';
$semester{'2007-05-08,2007-08-21'} = 'Summer semester';

print $semester{'2007-04-16'},"\n";  # prints 'Spring semester'
print $semester{'2007-06-01'},"\n";  # prints 'Summer semester'
```

If date and other ranges aren't powerful enough for you, how about regular-expression lookups? Tie::RegexpHash stores regular expressions as hash keys:

```
use Tie::RegexpHash;

tie my %rhash, 'Tie::RegexpHash';
$rhash{qr/[sS]asquatch/}   = "Sasquatch Trust and Savings";
$rhash{qr/\d{5}(-\d{4})?/} = 'US zip code';
$rhash{qr/^bucky/}         = 'invented by Buckminster Fuller';

print $rhash{'Sasquatch Bank'},"\n";
                  # prints "Sasquatch Trust and Savings"
print $rhash{'02114'},"\n";         # prints "US zip code"
print $rhash{'02114-2132'},"\n";  # prints "US zip code"
print $rhash{'buckyball'},"\n";     # prints "invented by Buckminster Fuller"
```

In this code we've defined keys based on some simple regular expressions (which could have been arbitrarily complex) instead of using your standard scalar keys. When presented with a key to look up in the hash, if a regular expression matched, the corresponding value is returned. Looking up a key that didn't match against any regular expression previously stored in the hash will return undef, just as expected.

By now I'm hoping that your creative juices are flowing. By perverting the usual lookup conventions of a hash we can unlock some pretty interesting programming possibilities. Let me show you one more example of this and then we'll get polymorphously perverse with our Perl variables. One of my favorite examples for fancy lookup modules based on tie() is Tie::NetAddr::IP. In this case, instead of providing regular expressions as keys, you instead provide IP range definitions (in CIDR notation):

```
use Tie::NetAddr::IP;

tie my %network, 'Tie::NetAddr::IP';
# load a list of our IP networks
$network{'192.168.0.0/24'} = 'server net';
```

```
$network{"192.168.1.0/24"} = "UNIX net";
$network{"192.168.2.0/24"} = "PC network";
```

With those definitions in place, we can now look up addresses. For example, if you found a machine had the address 192.168.0.24 and wanted to know what network it was on, it would be as simple as this:

```
print $network{'192.168.0.24'},"\n";  # prints "server net"
```

Magic Return Values

Your grip on (Perl) reality should be a little looser by now, so I trust you won't be too put out if I show you a couple of examples where you get more out of a scalar than you'd ordinarily expect:

```
use Tie::Scalar::Timestamp;

tie my $timestamp, 'Tie::Scalar::Timestamp';
print $timestamp,"\n";  # prints the current timestamp in
                        # (by default) ISO8601 format
```

With Tie::Scalar::Timestamp, you are creating a magic timestamp scalar that returns the current timestamp (in a format of your choosing) each time you access this variable. Sure, you could write a subroutine to do this, but that subroutine won't interpolate into strings as nicely as a Tie::Scalar::Timestamp tied variable.

A little more interesting magic can be found in the various modules that create scalars that can return a value from a predefined set of values each time you retrieve the contents. Tie::Cycle and Tie::Scalar::RingBuffer work this way. There are also bivalue modules, such as Tie::FlipFlop and Tie::Toggle, that switch between two possible outputs each time the variable is accessed. Here's one of these modules in action:

```
use Tie::Cycle;

tie my $round, 'Tie::Cycle', [qw( row row your boat )];

# each time we access $round, it returns the _next_ value in the list
print $round,"\n"; # prints "row"
print $round,"\n"; # prints "row"
print $round,"\n"; # prints "your"
print $round,"\n"; # prints "boat"
print $round,"\n"; # prints "row"
print $round,"\n"; # prints "row"
```

This is obviously a contrived example (unless you do a lot of campfire computing), but you can see how this might be useful in those situations where you need to repeatedly cycle over a set of values.

Wish Lists (warning: cliff hanger!)

In the final section of this part of the series we are going to further blur your notion of how hashes and other variables ""should"" work. To do that, let's go wild and assemble a wish list of things we wish a hash could do:

- Have elements that would automatically expire after a certain amount of time had elapsed.
- Keep a history of all changes made to it over time.
- Restrict the updates that are possible.
- Always keep track of the top N values or the rank of the values stored in it.

- Always return the keys in a sorted order based on the values in that hash.
- Transparently encrypt and decrypt itself.
- Easily store and retrieve multiple values per key.

As you have probably guessed, all of these things and more are possible thanks to tie()-based modules, and that's just using hashes. Rather than rush through the steps necessary to make this magic happen, we're going to hold off until next time to learn how to fulfill all of these wishes. Also, in the next part of this series, we'll look into how to actually write our own tie()-based module [and its tie()-less equivalent for those of you disenchanted with tie()]. If you get desperate before the next column to learn how this is done, please see the modules on CPAN in the Tie:: namespace plus the perltie man page that ships with Perl. Until then, take care, and I'll see you next time.

PROFESSORS, CAMPUS STAFF, AND STUDENTS—

DO YOU HAVE A USENIX REPRESENTATIVE ON YOUR CAMPUS?

IF NOT, USENIX IS INTERESTED IN HAVING ONE!

The USENIX Campus Rep Program is a network of representatives at campuses around the world who provide Association information to students, and encourage student involvement in USENIX. This is a volunteer program, for which USENIX is always looking for academics to participate. The program is designed for faculty who directly interact with students. We fund one representative from a campus at a time. In return for service as a campus representative, we offer a complimentary membership and other benefits.

A campus rep's responsibilities include:

- Maintaining a library (online and in print) of USENIX publications at your university for student use
- Distributing calls for papers and upcoming event brochures, and re-distributing informational emails from USENIX
- Encouraging students to apply for travel grants to conferences
- Providing students who wish to join USENIX with information and applications
- Helping students to submit research papers to relevant USENIX conferences
- Providing USENIX with feedback and suggestions on how the organization can better serve students

In return for being our "eyes and ears" on campus, representatives receive a complimentary membership in USENIX with all membership benefits (except voting rights), and a free conference registration once a year (after one full year of service as a campus rep).

To qualify as a campus representative, you must:

- Be full-time faculty or staff at a four year accredited university
- Have been a dues-paying member of USENIX for at least one full year in the past

For more information about our Student Programs, see http://www.usenix.org/students

USENIX contact: Anne Dickison, Director of Marketing, anne@usenix.org