COLUMNS

Practical Perl Tools

Constant as the Northern $*

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the Director of Technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ 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 one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010. dnb@ccs.neu.edu

The bad Perl wordplay in the title can mean only one thing: We have another column about programming in the Perl language underway. This issue's column is inspired and derived from an article I stumbled on by Neil Bowers. Back in 2012 he wrote an article on 21 different modules for defining constants in Perl. The original is at http://neilb.org/reviews/constants.html. If reading my take on his research gets you interested in the subject, do be sure to seek out his article. One other thing I should mention before we discuss his work: He's clearly cooler than I will ever be. When he encountered some bugs in one of the modules he reviewed, he "took over maintenance of the module and released a new version which addresses all known issues." Now that's thorough!

For this article I'm not going to discuss all of the 21 modules he reviewed. Rather, I thought it would be good to talk about why modules like this are not only a best practice sort of thing but downright handy, and then dive into some of the more accessible/interesting modules from Bowers' list.

What and Why

This may be fairly basic programming language terminology, but to make sure we're on the same page let me share one pragmatic view of what constants are and why you want to use them. Constants come into play when you want to write code that uses variables that don't change for the life of the program.

I realize that sounds a little strange—after all, why use a variable if it isn't going to change? Why not just use a value? It all comes down to code readability and maintainability. Let's say you are writing code that logs information using syslog and you want to specify which priority to log at. If we look at the C include file on the machine I'm typing on, we can see the following is defined:

```
#define LOG_EMERG      0    /* system is unusable */
#define LOG_ALERT      1    /* action must be taken immediately */
#define LOG_CRIT       2    /* critical conditions */
#define LOG_ERR        3    /* error conditions */
#define LOG_WARNING    4    /* warning conditions */
#define LOG_NOTICE     5    /* normal but significant condition */
#define LOG_INFO       6    /* informational */
#define LOG_DEBUG      7    /* debug-level messages */
```

This means I could write code that looks like:

```
log( 'It is getting kind of hot in here', 4 );
```

or

```
if ( $log_level == 4 ) { do_something_with_the_warning };
```

but unless I knew about /usr/include/sys/syslog.h, I'd still be shaking my head when I came back to this code in a year. You can just imagine the internal dialogue that would start with "'4', what the heck does '4' mean?"

A better version of those lines of code might be:

```
our $LOG_EMERG   = 0;  # system is unusable
our $LOG_ALERT   = 1;  # action must be taken immediately
our $LOG_CRIT    = 2;  # critical conditions
our $LOG_ERR     = 3;  # error conditions
our $LOG_WARNING = 4;  # warning conditions
our $LOG_NOTICE  = 5;  # normal but significant condition
our $LOG_INFO    = 6;  # informational
our $LOG_DEBUG   = 7;  # debug-level messages
```

which lets you then write lines that are considerably easier to read, like:

```
log( 'It is getting kind of hot in here', $LOG_EMERG );
               or
if ( $log_level == $LOG_EMERG)  { do_something_with_the_warning };
```

So this is all well and good until a little while later when your code base gets passed to a colleague who isn't as familiar with it and she's asked to make some "minor changes." While making these changes, she notices the use of $LOG_INFO sprinkled throughout the code and thinks, "Great, that's where I should store my log messages before they get sent out." She adds this to the code:

```
   $LOG_INFO = "Everything is peachy."; # set the log message
```

and lo and behold things start failing in a weird way (immediately if you are lucky, months later when no one remembers that changes were made if you are not). Here's a case in which you really want to use variables, but you want them to be immutable. Once you set a variable like this, you want it to stay at that value and scream bloody murder (or at least deny the request) if there are any attempts to change it from that point on.

There's no special variable type (as in scalar, list, hash) built into the Perl language to make this happen, so that's where the modules we'll be discussing come into play.

Behind the Scenes

Before we actually see any of these modules, I think it is useful to have in the back of your head a rough idea of how they work. As Bowers points out in his article, there are essentially two different ways to cast this particular spell.

First, you can use a mechanism already built into the language to associate some code with a variable. This code denies attempts to do anything but retrieve the value. Associating code with a variable is exactly what the tie() function does. There have been a number of columns here in which I've talked about the white and black magic associated with tie() so check out the archives if this notion intrigues you.

The other way some modules make variables read-only is to reach into the guts of the Perl core and use what is essentially an undocumented but well-known function called Internals::SvREADONLY. As Bowers notes in his article, the source code for the Perl interpreter has this to say where the function is defined:

```
XS(XS_Internals_SvREADONLY)    /* This is dangerous stuff. */
{
   dVAR;
   dXSARGS;
   ...
```

I realize this is a little scary. The conclusion I've come to after looking into this is SvREADONLY is well known enough and has been used in enough modules that I don't think I would be concerned about actually making use of it (indirectly via a module).

There are definitely pluses and minuses to each technique. Bowers does an excellent job of summarizing them toward the end of his article, so rather than rehashing them there, I'd recommend you look at his Comparison section.

Let's Do It

Okay, let's actually look at a number of the more straightforward modules out there. The first that should get mentioned is the one that has shipped with Perl since Perl 5.004 and is actually a pragma (a pre-processor directive). The constant pragma (to quote the doc) "allows you to declare constants at compile-time." What this means is the constant gets created before the actual program begins running (i.e., during the compilation phase when Perl is reading in the program and deciding how to execute it). I'll show you why that detail matters in just a second.

To use the pragma, you can write code like:

```
    # define a number of constants in one fell swoop
      use constant {
      LOG_EMERG              => 0,
      LOG_ALERT              => 1,
      LOG_CRIT               => 2,
      LOG_ERR                => 3,
      LOG_WARNING            => 4,
      LOG_NOTICE             => 5,
      LOG_INFO               => 6,
      LOG_DEBUG              => 7,
     }
    # ... or we could do this one at a time like this:
    #   use constant LOG_EMERG => 0;
    #   use constant LOG_ALERT => 1; ... etc.
    # now let's use it
    log('Here is a notice', LOG_NOTICE);
```

To prove the immutability of what we've defined, if we wrote

```
     LOG_NOTICE = "some other value";
```

it would immediately fail with an error message like

```
    Can't modify constant item in scalar assignment
```

Before we look at another module, let me explain the importance of the compile-time detail. To use an example modified from the docs, if I were to create a constant like

```
     use constant DEBUG => 0;
```

and use it in code like

```
     if (DEBUG) {
       # lots of debugging related code
           # yes, lots of it
           # ...
     }

   }
```

Perl will be smart enough to optimize that entire chunk of code out of the program before it runs because the value of DEBUG is false.

The second module I'd like to show you has actually made an appearance in this column before because it is the one recommended in Damian Conway's most excellent book *Perl Best Practices.* Conway recommends using the Readonly module because it allows you to do things like variable interpolation.

Quick aside: when you install Readonly, you may also want to install Readonly::XS. Readonly::XS is never called directly, but it lets Readonly use the Internals::SvREADONLY method for scalar constants (thus making it much faster than its usual use of tie()). Note: if you do want to use Readonly::XS, there is a long outstanding bug in Readonly that requires you to use Readonly::Scalar explicitly.

Here's the way Readonly gets used:

```
  use Readonly;
  Readonly my $LOG_EMERG        => 0;
  Readonly my $LOG_ALERT        => 1;
  Readonly my $LOG_CRIT         => 2;
  Readonly my $LOG_ERR          => 3;
  Readonly my $LOG_WARNING      => 4;
  Readonly my $LOG_NOTICE       => 5;
  Readonly my $LOG_INFO         => 6;
  Readonly my $LOG_DEBUG        => 7;
```

Then we do the usual:

```
    # note it is $LOG_NOTICE, not LOG_NOTICE
    log( 'Here is a notice', $LOG_NOTICE );
```

One difference between the constant pragma and Readonly is with Readonly we could write this:

```
    print "The value for the current log level is $LOG_NOTICE\n";
```

because string interpolation works. Readonly can also be used to make entire lists and hashes read-only if desired (though it does so using the slower tie() interface).

Although Readonly appears to be the most popular module of its ilk, possibly because of the Conway stamp of approval, it really hasn't seen much love in a while. The latest version on CPAN as of this writing is from April 2004 (though Readonly::XS did see a release in February of 2009). In his article, Bowers gives the nod to Const::Fast as one potentially worthy successor to Readonly. The doc for Const::Fast does indeed say it was written to work around some of Readonly's issues and actually says, "The implementation is inspired by doing everything the opposite way Readonly does it."

Like Readonly, it also lets you create read-only scalars, arrays, and hashes using Readonly-esque syntax as the example code in the doc demonstrates:

```
    use Const::Fast;

    const my $foo    => 'a scalar value';
    const my @bar    => qw/a list value/;
    const my %buz    => (a => 'hash', of => 'something');
```

Off the Beaten Path

Up until now we've looked at modules that have the same basic form and function. Before we end our time together in this column, I thought it might be interesting to look at a few modules that take this basic concept and extend it in some way.

The first module in this category is Config::Constants. Config::Constants encourages a potentially good development practice where the configuration for your program is (1) represented as constants and (2) stored in a separate file from the rest of the code. That separate file is in either XML or Perl data structure format. An example XML config file might look like this:

```
    <config>
      <module name "MyConstants">
        <constant name='LOG_EMERG'          value='0' />
        <constant name='LOG_ALERT'          value='1' />
        <constant name='LOG_CRIT'           value='2' />
        <constant name='LOG_ERR'            value='3' />
        <constant name='LOG_WARNING'        value='4' />
        <constant name='LOG_NOTICE'         value='5' />
        <constant name='LOG_INFO'           value='6' />
        <constant name='LOG_DEBUG'          value='7' />
    </module>
  </config>
```

with the equivalent Perl data structure version looking like this:

```
{
  'MyConstants' => {
          LOG_EMERG    => 0,
          LOG_ALERT    => 1,
          LOG_CRIT     => 2,
          LOG_ERR      => 3,
          LOG_WARNING  => 4,
          LOG_NOTICE   => 5,
          LOG_INFO     => 6,
          LOG_DEBUG    => 7,
           }
}
```

We'd typically create a module responsible for exporting constants to the rest of our program, as in:

```
package MyConstants;
use Config::Constants qw/LOG_EMERG LOG_ALERT LOG_CRIT
                         LOG_ERR LOG_WARNING LOG_NOTICE
                         LOG_INFO LOG_DEBUG/;
# define some functions that use these constants

    sub emerg_log {
      $message = shift;
          log($message, LOG_EMERG);
     }

 1;
```

To use this module, our main program would look like this:

```
use Config::Constants xml       => 'config.xml';
                 # or perl       => 'config.pl';
use MyConstants;

emerg_log('Houston, we have a problem');
```

Another module that also deals with the question of where the constants are defined is Constant::FromGlobal. With Constant::FromGlobal you'd write something like this:

```
use Constant::FromGlobal LOG_LEVEL =>
                                 { env     => 1,
                                   default => 0, };
```

and it will attempt to create a constant called LOG_LEVEL and set it to a value retrieved from a hierarchy:

- First it will see if there is a global variable $LOG_LEVEL set in the package, but if there is no global variable set...
- it will look for an environment variable called $MAIN_LOG_LEVEL, but if there is no environment variable set...
- it is given the default value (0).

In case you are curious about the name of the environment variable given in the second step above, Constant::FromGlobal wants the name of the current namespace prepended to variable name. By default, everything runs in the "main" namespace, although if we were using this in a module definition, we might write:

```
package Usenix;
use Constant::FromGlobal LOG_LEVEL =>
                                 { env     => 1,
                                   default => 0, };
```

and instead the module would look for an environment variable of USENIX_LOG_LEVEL instead.

Okay, last module of the day and then we can all go home. Constant::Generate module sets itself apart by being able to create values for you on the fly. Let's say you didn't care what the values were for constants, just that they had some. Constant::Generate lets you write:

```
use Constant::Generate [ qw( LOG_EMERG LOG_ALERT LOG_CRIT
                             LOG_ERR LOG_WARNING LOG_NOTICE
                             LOG_INFO LOG_DEBUG ) ];
```

and the constants get integer values starting at 0 (which coincidentally are the same values we've been setting by hand previously). For a slightly cooler self-assignment, we could instead say:

```
use Constant::Generate  [ qw( EMERG ALERT CRIT
                              ERR WARNING NOTICE
                              INFO DEBUG/)],
                                    prefix  => 'LOG_',
                                    dualvar => 1;
```

and not only do we get the LOG_something constants from before, but they act differently depending on the context they are used in, for example:

```
my $log_level = LOG_DEBUG;

print "Current log level: $log_level\n";
print "Yes, debug\n" if $log_level eq 'LOG_DEBUG';
print "Definitely debug\n" if $log_level == LOG_DEBUG;
```

In the first two print lines, the LOG_DEBUG constant is used in a string context. The constant appears to represent a string value that is identical to its name; however, in the third print statement we're making a numeric comparison and that still works fine. And with that little bit of magic, we'll stop here. Take care and I'll see you next time.