Practical Perl Tools
CLI Me a River

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 28+ 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 2005 conference and one of the LISA 2006 Invited Talks co-chairs. David is honored to be the recipient of the 2009 SAGE Outstanding Achievement award and to serve on the USENIX Board of Directors.
dnb@ccs.neu.edu

If Neil Stephenson's 1999 60+ page essay "In the Beginning was the Command Line" [1] left you feeling "ooh, a famous science fiction author really gets me, he really understands what is in my heart," then this column is for you. Today we're going to talk about a few ways you can write a better command-line program in Perl.

Switch It Up

One of the first things that contributes to someone's aesthetic pleasure of using a command-line tool is how well it handles arguments/switches. There are at least two sets of choices at work here. The first is a design one that Perl isn't going to help one whit with. Coming up with switch names that make sense for your program, are the same as or like the names used in similar programs in the same domain, are clear, and so on is up to you. This is by no means an easy task, because it requires careful thought.

The second set of choices does have a technical solution. The second set of choices is the one where you decide how your program will accept the arguments. Will there be spaces between them? Can you abbreviate and/or combine switches? Are some mandatory? And so on . . . This all matters because you want, whenever possible, for someone to try the arguments using the first way that comes into her head and have it work.

Where Perl helps with this is there are modules (oh so many modules) that handle argument-parsing for you. A number of them will handle all of the fiddly details for you so that your program can be liberal in how the arguments are specified (one dash, two dashes, abbreviated, abbreviated to single letters, optional and required arguments, and so on). The variety is dizzying. Before I show you one of these modules, I should mention that the Perl interpreter actually implements a built-in argument processor in the form of the -s switch. This means you can write code that looks like this:

```
#!/usr/bin/perl -s

if ($add)    { print "You want to add $add\n"; }
if ($remove) { print "You want to remove $remove\n"; }


if (${-help}) { print "this variable is crazy!\n"; }
```

which, when run, gives you:

```
$ s.pl -add=fred
You want to add fred
$ s.pl -remove
You want to remove 1
$ s.pl --help
this variable is crazy!
```

But don't write code that looks like that. The -s switch takes anything passed in with a dash, strips off the dash, and puts it in a variable with the name of the argument. This has all sorts of fun ramifications, a couple of which are mentioned in the perlrun doc:

> Do note that a switch like --help creates the variable "${-help}", which is not compliant with "use strict 'refs'". Also, when using this option on a script with warnings enabled you may get a lot of spurious "used only once" warnings.

In short, this means that you can kiss "use strict;", the thing everyone tells you to put first in our programs, goodbye unless you are willing to turn off some of the strictness.

Out of the crazy number of command argument parsing modules out there I'm only going to pick one to demonstrate. This is clearly a subject Perl authors like to riff on, so if it doesn't float your boat I'd encourage you to spend some time searching CPAN for one that does. And if you are a budding Perl module author who has aspirations of writing your own command argument-parsing module, I'd beseech you to check CPAN multiple times for something that works for you before reinventing yet another wheel.

The module we're going to explore is one of the most popular modules in this space, perhaps because it actually ships with Perl. Let's take a quick look at Getopt::Long. Getopt::Long can do so many things that the long manual page might be a bit daunting on first glance. We'll start with its sample code and then spice things up as we go along:

```
use Getopt::Long;
my $data   = "file.dat";
my $length = 24;
my $verbose;
GetOptions ("length=i" => \$length,    # numeric
            "file=s"   => \$data,      # string
            "verbose"  => \$verbose)   # flag
or die("Error in command line arguments\n");
```

The key function here is the GetOptions() call. The variable assignments before it are both to keep a "use strict" line (omitted in the sample code for space reasons) happy and probably just to reaffirm what kind of data is being referenced in the GetOptions call. Let's take that call apart.

In general, GetOptions takes a hash that defines the name of an argument, what kind of value it must or can be set to (numeric, string, etc.), any special characteristics (like "required" or "optional"), and a reference to a place to put the information parsed from the command-line arguments. For example, this part:

```
"length=i" => \$length,
```

says if we get an argument called length (--length), it must take a value and that value has to be an integer. That value will be stored in $length (i.e., --length 2 will put '2' in $length). In the case of a flag (like --verbose), the variable gets set to "1" so that Boolean tests like "if ($verbose)" will act as expected.

Two quick things to note before we start to add to this example code. The "or die()" that follows GetOptions works because GetOptions returns true if it can parse the options according to your wishes, false if that failed (e.g., someone passed in an argument you hadn't specified). The other thing to note is Getopt::Long by default will let you abbreviate unambiguous arguments on the command line and will handle multiple formats. This means I could call the program with:

```
$ s2.pl --length 2
$ s2.pl --length=2
$ s2.pl --l 2
$ s.2pl --l=2
$ s.2pl --le 2
```

and so on. Note that I don't have to code anything special to handle all of these different variations. This is what I mean by having Perl make it easier to make better command-line programs.

A moment ago, I said we could add to the sample code, so let me give you a list of how we can make the argument processing even fancier:

- Optional values (using : instead of = as in length:i)
- Multiple values per flag (pass a reference to an array instead of a scalar)
- Negated flags (i.e., --noverbose, which then sets $verbose to 0 instead of 1, specified by using an exclamation mark after the argument name)
- Cumulative flags (i.e., -v -v -v will give you more verbose output, specified by using + after the argument name)
- Argument name aliases (use different names for the same argument, specified by using a pipe character in the name, as in "verbose|chatty|moar" => \$verbose)

Getopt::Long has a few other tricks up its sleeve that I encourage you to go read about. The only one I want to mention before we move on is one I use on a regular basis. I haven't been very explicit about this, but hopefully you've sussed out that the way the rest of your program can determine which arguments and values were specified on the command line is through the variables being set by GetOptions(). I prefer to be able to find all of my options in a single place vs. a bunch of unconnected variables. To do that, we can tell GetOptions to store everything in a single hash by providing a reference to that hash as the first argument like so:

```
my %options = ();
GetOptions(\%options,
            "length=i"  => \$length,
            "file=s"    => \$data,
            "verbose"   => \$verbose);
```

When you do it that way, you can reference $options{length}, $options{file} and $options{verbose}. To check to see if an option has been set, you'll want to do something like

```
if ( exists $options{verbose} ) { ... }
```

As I mentioned before, there are tons of variations on the argument-parsing theme. Some of the variations I found most compelling are those that construct the argument specification from a script's internal documentation (e.g., in POD form). This leads nicely into the next topic.

Do the Doc

In the previous section I brought up the notion that we are endeavoring to design things like switch names to be intuitive and sensical to the script's users. But even if you manage to intuit or sense the heck out of your users (if that is even a term), there are still going to be times where those users will want to see a list of possible arguments and, ideally, some documentation for them.

That's where the module Pod::Usage comes into play. We've talked about this module back in 2006 and earlier this year, but I still want to remind you about it because having a mechanism for providing this documentation is pretty key to a good command-line program. You'll forgive me if I do as I did in one of those columns and reproduce the sample code from the Pod::Usage documentation, because it really does offer the best example for how to use the module. Plus, it even uses Getopt::Long, tying nicely into the last subject. Here's the sample code minus the actual specification of the USAGE and manual page in POD form:

```
use Getopt::Long;
use Pod::Usage;

my $man = 0;
my $help = 0;
## Parse options and print usage if there is a syntax error,
## or if usage was explicitly requested.
GetOptions('help|?' => \$help, man => \$man) or pod2usage(2);
pod2usage(1) if $help;
pod2usage(-verbose => 2) if $man;

## If no arguments were given, then allow STDIN to be used only
## if it's not connected to a terminal (otherwise print usage)
pod2usage("$0: No files given.")  if ((@ARGV == 0) && (-t STDIN));
__END__
```

Okay, so let's see what is going on here. Our newfound friend, GetOptions() from Getopt::Long, is being called to look for either an argument called "help" or "man". When it gets one of those two arguments, it calls pod2usage() with a return code and/or a "verbosity" level. A verbosity level of 0 shows an abridged USAGE message: 1 spits out the full USAGE message and 2 will print out the entire man page. Pod::Usage has rules about default error codes and verbosity levels in the doc that (as they say) mostly do the right thing. As an extra special trick, instead of calling die() as our previous Getopt::Long example did when it couldn't parse the arguments successfully, it now calls pod2usage() to spit out the usage message before exiting.

Welcome to My Shell

Just as some people believe that every program that increases in complexity over time eventually grows the ability to send email if it gets complex enough, I think you can make a good case that the more complex command line programs often grow an interactive mode. This interactive mode is usually like a mini-shell. If you find this happens to you, don't panic! Instead, let me offer you a tool to help make your interactive mode more pleasant for the people who will use it.

When building an interactive mode like this, you have to decide what level of help you want from a Perl module. Do you want something to just handle prompt parsing/validation (e.g., using IO::Prompt)? Do you want something to handle terminal interaction so someone can edit her or his commands in place (e.g., using Term::Readline)? Do you want something that will provide a list of valid commands with doc, etc.? Let's see one that gives us the full monty: Term::ShellUI.

Here's the first set of sample code described in the Term::ShellUI doc. I'm showing it to you because it demonstrates a whole host of things about what Term::ShellUI can do and how to do it:

```
use Term::ShellUI;
my $term = new Term::ShellUI(
    commands => {
        "cd" => {
            desc    => "Change to directory DIR",
            maxargs => 1,
            args    => sub { shift->complete_onlydirs(@_); },
            proc    => sub { chdir( $_[0] ||
                                 $ENV{HOME} ||
                                 $ENV{LOGDIR} ); },
     },
     "chdir" => { alias => 'cd' },
     "pwd"   => {
        desc    => "Print the current working directory",
        maxargs => 0,
        proc     => sub { system('pwd'); },
     },
        "quit" => {
           desc    => "Quit this program",
           maxargs => 0,
           method  => sub { shift->exit_requested(1); },
        }
     },
     history_file => '~/.shellui-synopsis-history',
  );
  print 'Using ' . $term->{term}->ReadLine . "\n";
  $term->run();
```

Let's look at the overall structure first. The code creates a new Term::ShellUI object by passing a specification into the module with a few hash keys. Reading from the bottom up to take the simpler one first, you can see we specify history_file, which tells Term::ShellUI to keep a history file. This will make it possible to repeat a previous command (even after you have quit and reentered the program). The more interesting hash key is "commands", the one before history_file. This is where we define which commands our mini-shell will accept and what to do for each command. Let's read from the top down and look at the arguments.

The first command that is defined by this code is a command for changing directories. It has a description to that effect (desc => ...), takes a single argument ("maxargs => 1"), provides "tab completion" for its arguments ("args => ...", which in this case calls complete_onlydirs() to only offer directory names as part of that completion) and actually performs the command via the Perl function chdir(). The next command, "chdir" shows how easy it is to define another name for a command that will be treated like the original one. The only part of the other commands worth mentioning is the line in the quit command that says:

```
method  => sub { shift->exit_requested(1); }
```

This tells the module to run the exit_requested() method of the object, which sets a flag that requests the module cease asking for more commands. Term::ShellUI has tons of other functionality you'll find described in the doc. Hopefully from this little snippet, it is obvious that you can get a full-fledged interactive mode/shell added to your script with little work.

With that, I hope I've given you a few tools to make more awesome command-line programs. Take care and I'll see you next time.

References

[1] http://www.cryptonomicon.com/beginning.html.

BECOME A USENIX SUPPORTER AND REACH YOUR TARGET AUDIENCE

The USENIX Association welcomes industrial sponsorship and offers custom packages to help you promote your organization, programs, and products to our membership and conference attendees.

Whether you are interested in sales, recruiting top talent, or branding to a highly targeted audience, we offer key outreach for our sponsors. To learn more about becoming a USENIX Supporter, as well as our multiple conference sponsorship packages, please contact sponsorship@usenix.org.

Your support of the USENIX Association furthers our goal of fostering technical excellence and innovation in neutral forums. Sponsorship of USENIX keeps our conferences affordable for all and supports scholarships for students, equal representation of women and minorities in the computing research community, and the development of open source technology.

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

www.usenix.org/usenix-supporter-program