DAVID N. BLANK-EDELMAN

practical Perl tools: Hi-ho the merry-o, debugging we will go

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 *Perl for System Administration*. He has spent the past 22+ 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.

dnb@ccs.neu

DEBUGGING CODE IS SUCH A NATURAL part of the software development process that it behooves us to do it as efficiently as possible. In my experience, many Perl programmers don't know about all of the many resources available that can make this process easier. We'll be taking a look at a grab bag of hints for debugging Perl.

Before we dive into the main subject, I feel compelled, some would say obsessively so, to mention that the most efficient way to get rid of bugs in your code is to avoid putting them there in the first place. Several of my previous columns have touted test-first and functional programming methodologies as two ways to work toward that goal, so I'll harp no more on them in this column.

Fun with the Perl Debugger

The Perl debugger is one of the best tools in your arsenal, so it is well worth your time to get to be best buddies with it. The output of `perldoc perldebug` and `perldoc perldebtut` plus the slim book *Perl Debugger Pocket Reference* by Richard Foley are required reading for this purpose. Here are a few tips surrounding the debugger that might not stand out for you on your first read through the material.

perl -de 0

Typing that command gives you a REPL (to use the computer-sciency term). A REPL is a Read-Eval-Print Loop. This basically provides a way to interactively type Perl code into a running Perl engine and receive the response back as fast as Perl can provide it. It can be very helpful to try out or hone small snippets of Perl code using this technique. As an aside, it should be noted that the REPL idea isn't even remotely new; several other languages (Python . . . cough . . . cough, etc.) even default to providing a REPL if you run their executable with no filename or other input specified.

$db::single

Another tip people often miss during a cursory read of the perldebug man page is the ability to write code that adds an automatic breakpoint for the Perl debugger when it is executed. If you set $DB::single to true (`$DB::single = 1` will do) and run the program under the debugger (`perl -d <filename>`), it will automatically stop at that point in your program and wait for instructions. This is helpful if you know "here be dragons" at a certain point in your code. You can run the code until it hits this breakpoint and further scrutinize it from there. This is easier than having to search for that point in your program and set a manual breakpoint each time.

a [ln] cmd, w expr

These two debugger commands let the debugger do some work for you. For the first one, you can set an "action," which will fire when the specified line is reached in your program. That action is just a Perl expression. For example, if you type the following at the debugger prompt:

```
a 28 print "Contents of a flakey variable: $flakey\n"
```

the debugger will print that message, complete with the current value of some variable we might be concerned about (in this case, `$flakey`) every time it hits line 28 in the program.

If you don't know the specific line of the program that is messing with a variable you care about, you might want to know every time the contents of the variable changes. It is possible to set a global watch expression to look for these changes by using something like:

```
w $flakey
```

Now every time the contents of $flakey get modified, you'll see something like this in the debugger:

```
Watchpoint 0:    $flakey changed:
  old value:   ''
  new value:   '1'
```

x %something VS. x \%something

The x command dumps the contents of variables and entire data structures. You can ask it to dump a hash-based data structure by itself (e.g., `%something`), but for best results, give it a reference to that data structure (i.e., x `\%something`) instead. The output is much nicer. Here's an example:

```
DB<1> %s = ( 'fred' => 'protagonist', 'barney' => 'foil,' )
DB<2> x %s
0 'barney'
1 'foil'
2 'fred'
3 'protagonist'
DB<3> x \%s
0 HASH(0x3c24b0)
  'barney' => 'foil'
  'fred' => 'protagonist'
```

Finding Where You Are Using Devel:: Modules

There are a ton of modules to help the Perl programmer with the development process in the Devel:: namespace. Let me show you a few Devel:: modules that may be useful to you. The `t` command in the Perl debugger can show you a trace of what lines are being executed during your program's run, but a few Devel:: modules, such as Devel::Trace, Devel::Xray, and Devel::LineTrace, can improve on that basic idea.

Since we almost always call in another module with a `use` statement, you might have forgotten it is possible to bring another module to bear using the `-d:` switch. To use Devel::Trace, you would type `perl -d:Trace {filename}`. When you do this, you get something that resembled the output of the -x flag when using the (Bourne, etc.) shell. Here's the example output from the documentation:

```
>> ./test:4:   print "Statement 1 at line 4\n";
>> ./test:5:   print "Statement 2 at line 5\n";
>> ./test:6:   print "Call to sub x returns ", &x(), " at line 6.\n";
>> ./test:12:  print "In sub x at line 12.\n";
>> ./test:13:  return 13;
>> ./test:8:   exit 0;
```

The second module in our list, Devel::XRay, gets loaded in the conventional manner:

```
use Devel::XRay 'all'; # to trace everything, you can be more specific
```

The end result (again, an example from the docs) is something like this:

```
# Hi-res seconds        # package::sub
[1092265261.834574]   main::init
[1092265261.836732]   Example::Object::new
[1092265261.837563]   Example::Object::name
[1092265261.838245]   Example::Object::calc
[1092265261.839443]   main::cleanup
```

This shows which subroutines or methods are being executed, along with a hi-res counter so you can have some sense of how long the program spent in each part of your code.

Finally, Devel::LineTrace is a bit of a strange duck, because it requires you to have a separate configuration file (by default, `perl-line-traces.txt`) describing just how you'd like it to behave. That file contains a list of filenames with line numbers, along with code that should be run for each line specified. This is essentially the same as the `a` (action) command from the debugger I mentioned earlier but makes it easier to associate debugging code to specific lines in your program without having to actually add that code to the program in question or type it into the debugger.

Once you have a config file, you run it like Devel::Trace, that is:

```
perl -d:LineTrace {script filename}
```

Inspecting the Data

One time-tested method for debugging since the dawn of the modern computer age is Ye Olde Printf Methode. This is the technique whereby you attempt to understand the program's state, or at least the state of a particular variable in question, by liberally sprinkling printf or the nearest language-appropriate equivalent all over the code. It isn't particularly efficient but it still works for some debugging scenarios.

In fact, there are a number of Perl modules that I won't get into here that allow for more refined versions of that basic model. They allow you to put conditional debugging statements into or around your program that fire when in debug mode only. One particularly clever one, Devel::StealthDebug, places these constructs in comments within the program, thus keeping them from interfering with the program's logic.

The use of print statements or their equivalent in modules like these tends to be less helpful when one is dealing with more complex data structures (although some of the more complicated modules in the class of those I just mentioned can handle this as well). For instance, it is all well and good to be able to write:

```
print STDERR "fred: $fred\n";
```

when $fred is a scalar value, but if it is a reference to a different data structure, that command prints out something like this:

```
fred: HASH(0x919fec)
```

which is much less helpful. The standard way to show the full data structure is to load the Data::Dumper module and call its Dumper() routine:

```
use Data::Dumper;
my $fred = { bob => 1 };
print Dumper($fred);
```

This prints out:

```
$VAR1 = {
          'bob' => 1
        };
```

Data::Dumper ships with Perl, which makes it a good first choice, but many people don't know that it has some limitations. For example, it can't handle code references. The code:

```
use Data::Dumper;
my $sub = sub { print "Meet you in the yurt\n"};

print Dumper ($sub);
```

yields:

```
$VAR1 = sub { "DUMMY" };
```

Though it isn't included with Perl by default, Data::Dump::Streamer is well worth installing because it can often do a better job than Data::Dumper. For example, the first code example, when changed to this:

```
use Data::Dump::Streamer;
my $fred = { bob => 1 };
print Dump($fred);
```

prints something a little prettier and easier to understand:

```
$HASH1 = { bob => 1 };
```

The second example showing the code reference problem, when changed in a similar fashion, yields this output instead:

```
$CODE1 = sub {
           print "Meet you in the yurt\n";
         };
```

which is far more useful.

One final tip for dealing with complex data structures: If you are more of a visual person, you may find the modules that graph data structures mentioned in October 2007's column (GraphViz::Data::Grapher and GraphViz::Data::Structure) to be helpful in visualizing a more involved data structure.

Debugging Regular Expressions

One of Perl's strengths is its regular expression functionality. It is also an area that could use help when it comes to debugging. Regular expressions can be considered a language unto themselves (and come Perl 6 this will become even more apparent). The "code" we write in this language within a Perl program often itself requires some debugging.

There are a number of programs (both commercial and free) to help with this process. Let me mention two of the free ones. The command-line regex debugger rred (http://www.csn.ul.ie/~hannes/code/rred/) attempts to show you just how your regular expression matches against a given string. For example, let's match against the string "It was a dark and stormy night." We set that by entering the string with a `t` prepended:

```
rred> tIt was a dark and stormy night.
```

We can then specify a regular expression to match against it by using an `e` as the first character of the line:

```
rred> ea(n|r)
```

The output produced looks like this:

```
$text ="It was a dark and stormy night."; $text =~ m/a(n|r)/g;
  It was a dark and stormy night.
$&          ^^
$1           ^
  It was a dark and stormy night.
$&               ^^
$1                ^
```

rred echoes back the test text and the regexp (it adds the `_g_` flag by default to the regexp). After that, we can see that on the first pass `$&` will be set to the "a" and "r" in "dark" and `$1` is set to the "r" in that word. On the second pass, `$&` now points to "an" in "and" and `$1` now contains the "n" from "and".

The second tool I want to mention, and this is the last for the column, is the re_graph utility found at http://www.oualline.com/sw/. This utility makes it easy to spot some common mistakes. Here's an example of a mistake I find a fair number of beginners make when I first introduce them to regular expressions in the context of email header parsing:

```
/^From|To:/
```

The student who writes this is trying to find either the From: or the To: header but instead has constructed a regexp that will match "From" at the beginning of a line and "To:" any place in the text. The person probably meant:

```
/^(From|To):/
```

(As an aside, I'll mention that the person very likely would want to use a non-capturing indicator (?:), but I'll leave that out of this example for simplicity's sake.)

Finding this mistake becomes really easy when using re_graph. The first regular expression yields the graph in Figure 1.

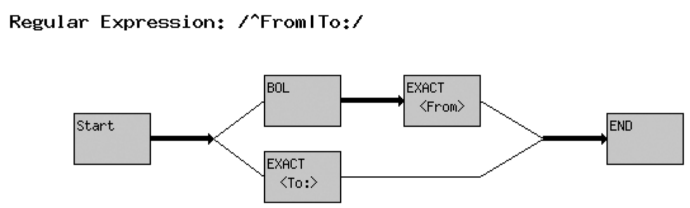

FIGURE 1: GRAPHING /^FROM|TO:/

The second regular expression produces the graph found in Figure 2.

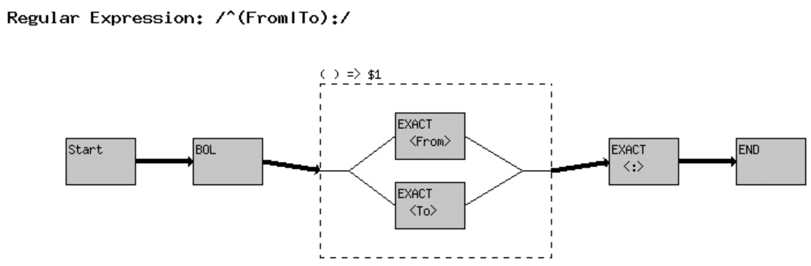

FIGURE 2: GRAPHING /^(FROM|TO):/

Even a cursory glance at the second graph makes it apparent that the second version has the "From" or "To" correctly preceded by a Beginning of Line (BOL). After either of the two headers is matched in an equal fashion a colon is required. This is definitely more correct than the first graph, which shows a BOL requirement only before the "From" match.

We have to bring things to a close now. Debugging is one of those surprisingly deep topics, so perhaps we'll revisit it in the future. Take care, and I'll see you next time.