The Perl Debugger

Daniel Allen

Abstract

Sticking in extra print statements is one way to debug your Perl code, but a full-featured debugger can give you more information.

Debugging is an annoying necessity in any language, whether it's debugging your own code or somebody else's that you've been given to make work on your system. Anything you can do to make debugging easier is a big win. Perl includes a command-line debugger that can make your debugging job considerably easier. This article covers the basics of the debugger and shows off a few tricks you may find useful.

Avoiding Bugs with Warnings and Strict

An astounding number of bugs can be caught by Perl automatically by turning on warnings and strict at the beginning of your program. If your program includes the line **`use warnings;`** you can catch dozens of common errors, including variables used only once, which often are typos; scalar variables used before they are set; and redefined subroutines.

These diagnostic messages can be explained further by including the line **`use diagnostics;`**, which prints an explanation for each warning. Or, you can look up the explanations using **`man perldiag`**.

If your Perl version is older than 5.6, instead of use warnings; you have to use the -w option on the first line of the script, like this: **`#!/usr/bin/perl -w`**.

Finally, you can catch additional common errors with **`use strict;`**, which in effect, forbids a few unsafe programming shortcuts. The rules that use strict turns on are as follows:

- Variables must be declared before use, with **`my`** or **`our`**, or imported with **`use vars`** or fully qualified with a package name.
- Bare words must be subroutines, not strings, such as **`$string = blah;`**.
- References cannot be symbolic; see sidebar.

As you can see, warnings and strict tighten up a few of Perl's features that can be used for good but also can be abused. These commands make debugging easier, because Perl catches these bugs for you.

Symbolic References

Symbolic references are different from regular hard references, where a variable refers to another variable. Symbolic references are created when the programmer uses a string as a reference. For example, this normally is valid Perl code:

```
$name = "username";
$$name = "da";      # sets $username
```

This code easily can cause a case of the interpreter doing what you said, not what you meant. It is easy to put a symbolic reference where a hard reference was intended or to confuse the generated variable name because it never appears in the code. A much safer way to accomplish the same thing is to use a hash to store such variables and to turn on strict variable checking with use strict.

What's Wrong with Print Statements?

You might ask at this point, what's wrong with debugging by scattering print statements in your code? Nothing is wrong with this debugging technique, but you have more power with the interactive debugger. You can examine all aspects of the program and environment, not only those you thought of when you ran the program, and you can see more clearly what the program actually does. Hopefully, by the end of this article you will agree that investing a little effort in learning the debugger pays off in saved time.

Starting the Debugger

The debugger is run from the command line by passing Perl the -d option:

```
perl -d filename.pl
```

If you are debugging a CGI program written with CGI.pm, simply call it on the command line with the arguments you'd like to pass, along with -d:

```
perl -d filename.pl param=value param2=value
```

Instead of using the command line, you could use the Perl debugger as part of certain IDEs, such as GNU Emacs or Activestate Komodo, or from debugger GUI front ends, such as ddd or ptkdb. For space reasons, I discuss only the command line in this article, but the principles hold for a GUI debugger as well.

If you're using the command-line debugger, it is useful to have the Term::ReadLine module installed, which enables cursoring through the command history.

Here's an example program we use in this article. Copy the following to a file called sample.pl:

```
#!/usr/bin/perl

use warnings;
use strict;

my $name = "Pengu";

foreach (1..20) {
  &shout($name);
}

sub shout {
  my $name = shift;
  print "*** $name ***\n";
}
```

Essential Debugger Commands

The following seven commands are sufficient for basic debugging:

- s: single-step execute the next line, stepping into subroutines.
- n: single-step execute the next line, stepping over subroutines.
- r: nonstop execute until the return from the current subroutine.
- c <line-number>: nonstop execute until a particular line.
- l <line-number, range or subroutine>: list source code.
- x <expression>: evaluate and pretty-print <expression>.
- q: quit debugger.

To try these out, run the test program with the debugger:

```
perl -d sample.pl
```

You should see debugger startup information:

```
Default die handler restored.
Loading DB routines from perl5db.pl version 1.07
Editor support available.
Enter h or h h for help or
man perldebug for more help:

main::(sample.pl:6):    my $name = "Pengu";
  DB<1>
```

This is the state before the program starts running. The next-to-last line has useful information about the debugging status: you're in the main package, file sample.pl line 6, and it displays the line that is about to be run.

The last line is a prompt with the command number (incrementing as you enter more commands) and angle brackets, where the number of angle brackets signifies nested commands. You don't need to worry about those here.

Type **s** at the prompt and press Enter to single-step one line into the program:

```
  DB<1> s
main::(sample.pl:8):    foreach (1..20) {
  DB<1>
```

To repeat the command, press Enter; repeat this as long as you like to be convinced that the program is stepping through its paces. Every time you pass the print statement, it is echoed to the screen, interspaced with the debugging materials.

Now, try the command to step over subroutines (**n**), and press Enter a few times. You go through the loop and receive your subroutine results right away, without stepping through each command in the subroutine.

Next, try the command to return from the current subroutine (**r**). But wait—if you do it now, it will run until the program finishes, because you're “returning” from the main program. First, do a couple repetitions of **s** to step into the subroutine. Then, with an **r**, you should see something like:

```
  DB<1> s
main::(sample.pl:8):    foreach (1..20) {
  DB<1>
main::(sample.pl:9):      &shout($name);
  DB<1>
main::shout(sample.pl:13):        my $name = shift;
  DB<1> r
*** Pengu ***
void context return from main::shout
main::(sample.pl:8):    foreach (1..20) {
  DB<1>
```

Notice the **`void context return from main::shout`** line. If we had asked for a return value in the main loop, we would see it displayed here. In Perl, functions and subroutines can return different values based on the context of the caller (scalar, array or void). A nice feature of the Perl debugger is the r command, which tells you what context was requested by the caller. It can find the bug if you ask your subroutine for a scalar, but you mistakenly have the subroutine return an array.

Next, we have the l command. Try it now:

```
  DB<1> l
8==>    foreach (1..20) {
9:        &shout($name);
10      }
11
12      sub shout {
13:       my $name = shift;
14:       print "*** $name ***\n";
15      }
  DB<1>
```

Alone, **`l`** lists a page of the source code, starting at the next line to be executed, with a text arrow pointing to the next line. You also can list a range by specifying the line numbers, such as **`l 200-230`**. Additionally, you can list a subroutine by naming it: **`l shout`**.

The **`c`** command continues execution until you hit a particular line number, so you can jump ahead to a particular piece of code that is interesting:

```
  DB<1> c 14
main::shout(sample.pl:14):  print "*** $name ***\n";
  DB<1>
```

You can execute any Perl expression, including code that changes the running program, by typing it at the prompt. This can include setting variables in the program by hand.

The **`x`** command evaluates and pretty-prints any expression, prepending a numbered index on each line of output, dereferencing anything that can be dereferenced and indenting each new level of dereferencing. As an example, below we set an array, @sample, and then display it:

```
  DB<1> @sample = (1..5)

  DB<2> x @sample
0  1
1  2
2  3
```

```
3  4
4  5
  DB<3>
```

Notice that hashes are displayed with keys and values, each one on a line. You can display hashes properly by preceeding the hash with a \, which turns the hash into a hash reference, which is properly dereferenced. This looks like:

```
  DB<4> %sample = (1 .. 8)

  DB<5> x \%sample
0  HASH(0x83d53bc)
   1 => 2
   3 => 4
   5 => 6
   7 => 8
  DB<6>
```

When you are satisfied with the results, quit the debugging exercise with **q**.

Four More Debugger Commands

Many people use the Perl debugger with no more than these commands. Once you are comfortable with those, however, an additional four commands can make your debugging more efficient, especially for programs that use object-oriented code:

- /<pattern>: lists source code at next regular expression match.
- ?<pattern>: lists source code at previous regular expression match.
- S: lists all subroutines and methods available to the program.
- m <object or package>: lists all methods available on the given object or package.

You can search and display code that matches a string or regular expression with / for forward searches and ? for backward searches. There should be no space before the string you're looking for:

```
  DB<6> /name
6:      my $name = "Pengu";
```

The S and m commands are useful for exploring what subroutines or methods are available: **S** lists every subroutine and method available to the program. These are in reverse order of when they were loaded by use or require, and they include routines loaded from the debugger, such as Term::ReadLine. The m command lists every method available to an object or by way of a package. Here is a sample:

```
  DB<7> use CGI

  DB<8> $q = new CGI

  DB<9> m $q
AUTOLOAD
DESTROY
XHTML_DTD
```

```
_compile
_make_tag_func
_reset_globals
_setup_symbols
add_parameter
all_parameters
[...]
```

Actions, Breakpoints and Watchpoints

Actions, breakpoints and watchpoints provide even more control over the debugger and the running program. You may prefer using them from a graphical Perl debugging front end, such as ddd, ptkdb or Activestate Komodo. The most common complaint about the Perl debugger is remembering the proper command-line shortcut for each command, and these commands add still more shortcuts to remember.

Additionally, in Perl 5.8 some of the keyboard commands have changed to make them more internally consistent. Often, though, people need to use both 5.8 and an earlier version, so it may be easier to use a GUI. I describe the commands from the command line below; the principles remain the same.

An action is used to wedge code into your program without modifying the source file. It can be useful when the code is in production and you want to test a change. It's also useful if you're in the middle of a debugging run and want to change code without restarting the debugging session from scratch.

You set an action like so, **`a <line-number> <code>`**. An example could be:

```
DB<10> a 9 $index = $_;
```

This adds a new command inside the foreach loop that stores the index count, which is incremented each time through. If you list the program, you see an **`a`** next to the line number that has the action. The action is executed before the line to which it is attached. You can list actions you've set with **`L`** and delete an action by specifying **`a`** with the line number without a command. The previous is for Perl 5.6 and earlier; in Perl 5.8, delete an action with **`A`** and the line number of the action.

Breakpoints and watchpoints return control to the debugger from continuous execution, such as from **`r`** and **`c`** described above. They are useful for jumping ahead to the particular iteration of a loop that is having problems, without repeatedly stepping through the loop by hand.

A breakpoint is set on a line number or subroutine, with an optional condition that must be met. A breakpoint is set with **`b`** as shown here:

```
b shout
```

If you list the program, you can see a **`b`** next to the line number at the first line of the subroutine shout. Press C to continue execution, and it stops inside the subroutine.

If you followed the previous example and set the action on line 9, you could set a breakpoint to stop on a particular iteration of this loop:

```
b shout ($index eq 8)
```

This should give you an idea of the power of actions and breakpoints, if you imagine debugging a longer program with complex conditional statements and external data sources.

You can list breakpoints with **L** and delete one with **d** in Perl 5.6 and earlier. In Perl 5.8, you delete a breakpoint with **B**.

A watchpoint probably is better known as a watch expression. It halts the program as soon as a specified expression changes. In Perl 5.6, it is set with **W** as shown here:

```
W $name
```

You can list watchpoints with **L** and delete all of them by specifying no parameter to **W**. In Perl 5.8, add a watchpoint with **w** and delete it with **W**.

Customizing the Perl Debugger

The first thing to know is that the debugger is simply a Perl library that takes advantage of hooks in the Perl interpreter. You could replace the debugger completely, if you like, by copying the file somewhere and requiring the file in your code in a BEGIN loop:

```
cp /usr/lib/perl5/5.6.1/perl5db.pl ~/myperl5db.pl
```

And, place this line in your program:

```
BEGIN { require "~/myperl5db.pl" }
```

You might do this, for example, if you preferred the syntax and operation of the 5.6 version debugger over the 5.8 version.

You also can specify an alternative debugger with the -d command switch. Perl versions 5.6 onward include DProf, a profiler that uses debugger hooks. You can use it like this:

```
perl -d:DProf mycode.pl
```

You also can use the debugger hooks in your own programs. You can set a breakpoint directly in your code by setting the variable **$DB::single = 1;**, which is useful if you need to debug code in a BEGIN block. Otherwise, they are executed before the debugger gives you a prompt. Or, you could use the hooks to run particular code whenever any subroutine is run. To find out more about these and other hooks, check the perldebug man page.

The debugger has a set of internal variables, also described in the perldebug man page. To change these variables you can use a configuration file, .perldb in the current directory or in your home directory. This configuration file has Perl code that is run when the debugger starts. For example, you can add new commands of your own, like this:

```
$DB::alias{'quit'} = 's/^quit(\s*)/q/';
```

This allows you to quit the debugger by typing **quit** at the prompt. The perldebug man page describes a few similar aliases that might be useful.

A number of debugger options can be set inside the debugger with the O command. The only one I have used changes the pager:

```
O pager=|less
```

This way, any command that would print more than a screen of output can be sent through your favorite pager by using a pipe character before the command: `|L`.

Resources for this article: http://www.linuxjournal.com/article/7962.