NAME

perldebtut - Perl debugging tutorial

DESCRIPTION

A (very) lightweight introduction in the use of the perl debugger, and a pointer to existing, deeper sources of information on the subject of debugging perl programs.

There's an extraordinary number of people out there who don't appear to know anything about using the perl debugger, though they use the language every day. This is for them.

use strict

First of all, there's a few things you can do to make your life a lot more straightforward when it comes to debugging perl programs, without using the debugger at all. To demonstrate, here's a simple script, named "hello", with a problem:

```
#!/usr/bin/perl

$var1 = 'Hello World'; # always wanted to do that :-)
$var2 = "$varl\n";

print $var2;
exit;
```

While this compiles and runs happily, it probably won't do what's expected, namely it doesn't print "Hello World\n" at all; It will on the other hand do exactly what it was told to do, computers being a bit that way inclined. That is, it will print out a newline character, and you'll get what looks like a blank line. It looks like there's 2 variables when (because of the typo) there's really 3:

```
$var1 = 'Hello World';
$varl = undef;
$var2 = "\n";
```

To catch this kind of problem, we can force each variable to be declared before use by pulling in the strict module, by putting 'use strict;' after the first line of the script.

Now when you run it, perl complains about the 3 undeclared variables and we get four error messages because one variable is referenced twice:

```
Global symbol "$var1" requires explicit package name at ./t1 line 4.
Global symbol "$var2" requires explicit package name at ./t1 line 5.
Global symbol "$varl" requires explicit package name at ./t1 line 5.
Global symbol "$var2" requires explicit package name at ./t1 line 7.
Execution of ./hello aborted due to compilation errors.
```

Luvverly! and to fix this we declare all variables explicitly and now our script looks like this:

```
#!/usr/bin/perl
use strict;

my $var1 = 'Hello World';
my $varl = undef;
my $var2 = "$varl\n";

print $var2;
exit;
```

We then do (always a good idea) a syntax check before we try to run it again:

```
> perl -c hello
hello syntax OK
```

And now when we run it, we get "\n" still, but at least we know why. Just getting this script to compile has exposed the '$varl' (with the letter 'l') variable, and simply changing $varl to $var1 solves the problem.

Looking at data and -w and v

Ok, but how about when you want to really see your data, what's in that dynamic variable, just before using it?

```
#!/usr/bin/perl
use strict;

my $key = 'welcome';
my %data = (
 'this' => qw(that),
 'tom' => qw(and jerry),
 'welcome' => q(Hello World),
 'zip' => q(welcome),
);
my @data = keys %data;

print "$data{$key}\n";
exit;
```

Looks OK, after it's been through the syntax check (perl -c scriptname), we run it and all we get is a blank line again! Hmmmm.

One common debugging approach here, would be to liberally sprinkle a few print statements, to add a check just before we print out our data, and another just after:

```
print "All OK\n" if grep($key, keys %data);
print "$data{$key}\n";
print "done: '$data{$key}'\n";
```

And try again:

```
> perl data
All OK

done: ''
```

After much staring at the same piece of code and not seeing the wood for the trees for some time, we get a cup of coffee and try another approach. That is, we bring in the cavalry by giving perl the '**-d**' switch on the command line:

```
> perl -d data
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::(./data:4):     my $key = ’welcome’;
```

Now, what we've done here is to launch the built-in perl debugger on our script. It's stopped at the first line of executable code and is waiting for input.

Before we go any further, you'll want to know how to quit the debugger: use just the letter '**q**', not the words 'quit' or 'exit':

```
 DB<1> q
 >
```

That's it, you're back on home turf again.

help

Fire the debugger up again on your script and we'll look at the help menu. There's a couple of ways of calling help: a simple '**h**' will get the summary help list, '**|h**' (pipe-h) will pipe the help through your pager (which is (probably 'more' or 'less'), and finally, '**h h**' (h-space-h) will give you the entire help screen. Here is the summary page:

D**1**h

```
 List/search source lines:               Control script execution:
  l [ln|sub]  List source code             T           Stack trace
  - or .      List previous/current line   s [expr]    Single step [in expr]
  v [line]    View around line             n [expr]    Next, steps over subs
  f filename  View source in file          <CR/Enter>  Repeat last n or s
  /pattern/ ?patt?   Search forw/backw     r           Return from
subroutine
  M           Show module versions         c [ln|sub]  Continue until
position
 Debugger controls:                        L           List
break/watch/actions
  o [...]     Set debugger options         t [expr]    Toggle trace [trace
expr]
  <[<]|{[{]|>[>] [cmd] Do pre/post-prompt b [ln|event|sub] [cnd] Set
breakpoint
  ! [N|pat]   Redo a previous command      B ln|*      Delete a/all
breakpoints
  H [-num]    Display last num commands    a [ln] cmd  Do cmd before line
  = [a val]   Define/list an alias         A ln|*      Delete a/all actions
  h [db_cmd]  Get help on command          w expr      Add a watch
expression
  h h         Complete help page           W expr|*    Delete a/all watch
exprs
  |[|]db_cmd  Send output to pager         ![!] syscmd Run cmd in a
subprocess
  q or ^D     Quit                         R           Attempt a restart
 Data Examination:     expr     Execute perl code, also see: s,n,t expr
  x|m expr       Evals expr in list context, dumps the result or lists
methods.
  p expr         Print expression (uses script’s current package).
  S [[!]pat]     List subroutine names [not] matching pattern
  V [Pk [Vars]]  List Variables in Package.  Vars can be ~pattern or
!pattern.
```

```
  X [Vars]       Same as "V current_package [Vars]".
  y [n [Vars]]   List lexicals in higher scope <n>.  Vars same as V.
For more help, type h cmd_letter, or run man perldebug for all docs.
```

More confusing options than you can shake a big stick at! It's not as bad as it looks and it's very useful to know more about all of it, and fun too!

There's a couple of useful ones to know about straight away. You wouldn't think we're using any libraries at all at the moment, but '**M**' will show which modules are currently loaded, and their version number, while '**m**' will show the methods, and '**S**' shows all subroutines (by pattern) as shown below. '**V**' and '**X**' show variables in the program by package scope and can be constrained by pattern.

```
DB<2>S str
dumpvar::stringify
strict::bits
strict::import
strict::unimport
```

Using 'X' and cousins requires you not to use the type identifiers ($@%), just the 'name':

```
DM<3>X ~err
FileHandle(stderr) => fileno(2)
```

Remember we're in our tiny program with a problem, we should have a look at where we are, and what our data looks like. First of all let's view some code at our present position (the first line of code in this case), via '**v**':

```
DB<4> v
1       #!/usr/bin/perl
2:      use strict;
3
4==>    my $key = 'welcome';
5:      my %data = (
6               'this' => qw(that),
7               'tom' => qw(and jerry),
8               'welcome' => q(Hello World),
9               'zip' => q(welcome),
10      );
```

At line number 4 is a helpful pointer, that tells you where you are now. To see more code, type 'v' again:

```
DB<4> v
8               'welcome' => q(Hello World),
9               'zip' => q(welcome),
10      );
11:     my @data = keys %data;
12:     print "All OK\n" if grep($key, keys %data);
13:     print "$data{$key}\n";
14:     print "done: '$data{$key}'\n";
15:     exit;
```

And if you wanted to list line 5 again, type 'l 5', (note the space):

```
DB<4> l 5
5:      my %data = (
```

In this case, there's not much to see, but of course normally there's pages of stuff to wade through, and 'l' can be very useful. To reset your view to the line we're about to execute, type a lone period '.':

```
DB<5> .
main::(./data_a:4):     my $key = 'welcome';
```

The line shown is the one that is about to be executed **next**, it hasn't happened yet. So while we can print a variable with the letter '**p**', at this point all we'd get is an empty (undefined) value back. What we need to do is to step through the next executable statement with an '**s**':

```
DB<6> s
main::(./data_a:5):     my %data = (
main::(./data_a:6):             'this' => qw(that),
main::(./data_a:7):             'tom' => qw(and jerry),
main::(./data_a:8):             'welcome' => q(Hello World),
main::(./data_a:9):             'zip' => q(welcome),
main::(./data_a:10):    );
```

Now we can have a look at that first ($key) variable:

```
DB<7> p $key
welcome
```

line 13 is where the action is, so let's continue down to there via the letter '**c**', which by the way, inserts a 'one-time-only' breakpoint at the given line or sub routine:

```
DB<8> c 13
All OK
main::(./data_a:13):    print "$data{$key}\n";
```

We've gone past our check (where 'All OK' was printed) and have stopped just before the meat of our task. We could try to print out a couple of variables to see what is happening:

```
DB<9> p $data{$key}
```

Not much in there, lets have a look at our hash:

```
DB<10> p %data
Hello Worldziptomandwelcomejerrywelcomethisthat


DB<11> p keys %data
Hello Worldtomwelcomejerrythis
```

Well, this isn't very easy to read, and using the helpful manual (**h h**), the '**x**' command looks promising:

```
DB<12> x %data
0  'Hello World'
1  'zip'
2  'tom'
3  'and'
4  'welcome'
5  undef
6  'jerry'
7  'welcome'
8  'this'
```

```
9  'that'
```

That's not much help, a couple of welcomes in there, but no indication of which are keys, and which are values, it's just a listed array dump and, in this case, not particularly helpful. The trick here, is to use a **reference** to the data structure:

```
DB<13> x \%data
0  HASH(0x8194bc4)
   'Hello World' => 'zip'
   'jerry' => 'welcome'
   'this' => 'that'
   'tom' => 'and'
   'welcome' => undef
```

The reference is truly dumped and we can finally see what we're dealing with. Our quoting was perfectly valid but wrong for our purposes, with 'and jerry' being treated as 2 separate words rather than a phrase, thus throwing the evenly paired hash structure out of alignment.

The '**-w**' switch would have told us about this, had we used it at the start, and saved us a lot of trouble:

```
> perl -w data
Odd number of elements in hash assignment at ./data line 5.
```

We fix our quoting: 'tom' => q(and jerry), and run it again, this time we get our expected output:

```
> perl -w data
Hello World
```

While we're here, take a closer look at the '**x**' command, it's really useful and will merrily dump out nested references, complete objects, partial objects - just about whatever you throw at it:

Let's make a quick object and x-plode it, first we'll start the debugger: it wants some form of input from STDIN, so we give it something non-committal, a zero:

```
> perl -de 0
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::(-e:1):   0
```

Now build an on-the-fly object over a couple of lines (note the backslash):

```
DB<1> $obj = bless({'unique_id'=>'123', 'attr'=> \
cont:  {'col' => 'black', 'things' => [qw(this that etc)]}}, 'MY_class')
```

And let's have a look at it:

```
  DB<2> x $obj
0  MY_class=HASH(0x828ad98)
    'attr' => HASH(0x828ad68)
      'col' => 'black'
```

```
        'things' => ARRAY(0x828abb8)
           0  'this'
           1  'that'
           2  'etc'
      'unique_id' => 123
   DB<3>
```

Useful, huh? You can eval nearly anything in there, and experiment with bits of code or regexes until the cows come home:

```
 DB<3> @data = qw(this that the other atheism leather theory scythe)

 DB<4> p 'saw -> '.($cnt += map { print "\t:\t$_\n" } grep(/the/, sort
@data))
 atheism
 leather
 other
 scythe
 the
 theory
 saw -> 6
```

If you want to see the command History, type an '**H**':

```
 DB<5> H
 4: p 'saw -> '.($cnt += map { print "\t:\t$_\n" } grep(/the/, sort @data))
 3: @data = qw(this that the other atheism leather theory scythe)
 2: x $obj
 1: $obj = bless({'unique_id'=>'123', 'attr'=>
 {'col' => 'black', 'things' => [qw(this that etc)]}}, 'MY_class')
 DB<5>
```

And if you want to repeat any previous command, use the exclamation: '**!**':

```
 DB<5> !4
 p 'saw -> '.($cnt += map { print "$_\n" } grep(/the/, sort @data))
 atheism
 leather
 other
 scythe
 the
 theory
 saw -> 12
```

For more on references see *perlref* and *perlreftut*

Stepping through code

Here's a simple program which converts between Celsius and Fahrenheit, it too has a problem:

```
 #!/usr/bin/perl -w
 use strict;

 my $arg = $ARGV[0] || '-c20';

 if ($arg =~ /^\-(c|f)((\-|\+)*\d+(\.\d+)*)$/) {
```

```
 my ($deg, $num) = ($1, $2);
 my ($in, $out) = ($num, $num);
 if ($deg eq 'c') {
  $deg = 'f';
  $out = &c2f($num);
 } else {
  $deg = 'c';
  $out = &f2c($num);
 }
 $out = sprintf('%0.2f', $out);
 $out =~ s/^((\-|\+)*\d+)\.0+$/$1/;
 print "$out $deg\n";
} else {
 print "Usage: $0 -[c|f] num\n";
}
exit;

sub f2c {
 my $f = shift;
 my $c = 5 * $f - 32 / 9;
 return $c;
}

sub c2f {
 my $c = shift;
 my $f = 9 * $c / 5 + 32;
 return $f;
}
```

For some reason, the Fahrenheit to Celsius conversion fails to return the expected output. This is what it does:

```
> temp -c0.72
33.30 f

> temp -f33.3
162.94 c
```

Not very consistent! We'll set a breakpoint in the code manually and run it under the debugger to see what's going on. A breakpoint is a flag, to which the debugger will run without interruption, when it reaches the breakpoint, it will stop execution and offer a prompt for further interaction. In normal use, these debugger commands are completely ignored, and they are safe - if a little messy, to leave in production code.

```
my ($in, $out) = ($num, $num);
$DB::single=2; # insert at line 9!
if ($deg eq 'c')
 ...

> perl -d temp -f33.3
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::(temp:4): my $arg = $ARGV[0] || ’-c100’;
```

We'll simply continue down to our pre-set breakpoint with a '**c**':

```
  DB<1> c
main::(temp:10):                if ($deg eq ’c’) {
```

Followed by a view command to see where we are:

```
DB<1> v
7:              my ($deg, $num) = ($1, $2);
8:              my ($in, $out) = ($num, $num);
9:              $DB::single=2;
10==>           if ($deg eq ’c’) {
11:                     $deg = ’f’;
12:                     $out = &c2f($num);
13              } else {
14:                     $deg = ’c’;
15:                     $out = &f2c($num);
16              }
```

And a print to show what values we're currently using:

```
DB<1> p $deg, $num
f33.3
```

We can put another break point on any line beginning with a colon, we'll use line 17 as that's just as we come out of the subroutine, and we'd like to pause there later on:

```
DB<2> b 17
```

There's no feedback from this, but you can see what breakpoints are set by using the list 'L' command:

```
DB<3> L
temp:
  17:            print "$out $deg\n";
    break if (1)
```

Note that to delete a breakpoint you use 'd' or 'D'.

Now we'll continue down into our subroutine, this time rather than by line number, we'll use the subroutine name, followed by the now familiar 'v':

```
DB<3> c f2c
main::f2c(temp:30):             my $f = shift;

DB<4> v
24:     exit;
25
26      sub f2c {
27==>           my $f = shift;
28:             my $c = 5 * $f - 32 / 9;
29:             return $c;
```

```
30      }
31
32      sub c2f {
33:             my $c = shift;
```

Note that if there was a subroutine call between us and line 29, and we wanted to **single-step** through it, we could use the '**s**' command, and to step over it we would use '**n**' which would execute the sub, but not descend into it for inspection. In this case though, we simply continue down to line 29:

```
DB<4> c 29
main::f2c(temp:29):             return $c;
```

And have a look at the return value:

```
DB<5> p $c
162.944444444444
```

This is not the right answer at all, but the sum looks correct. I wonder if it's anything to do with operator precedence? We'll try a couple of other possibilities with our sum:

```
DB<6> p (5 * $f - 32 / 9)
162.944444444444

DB<7> p 5 * $f - (32 / 9)
162.944444444444

DB<8> p (5 * $f) - 32 / 9
162.944444444444

DB<9> p 5 * ($f - 32) / 9
0.722222222222221
```

:-) that's more like it! Ok, now we can set our return variable and we'll return out of the sub with an 'r':

```
DB<10> $c = 5 * ($f - 32) / 9

DB<11> r
scalar context return from main::f2c: 0.722222222222221
```

Looks good, let's just continue off the end of the script:

```
DB<12> c
0.72 c
Debugged program terminated.  Use q to quit or R to restart,
  use O inhibit_exit to avoid stopping after program termination,
  h q, h R or h O to get additional info.
```

A quick fix to the offending line (insert the missing parentheses) in the actual program and we're finished.

Placeholder for a, w, t, T

Actions, watch variables, stack traces etc.: on the TODO list.

```
a
```

```
w

t

T
```

REGULAR EXPRESSIONS

Ever wanted to know what a regex looked like? You'll need perl compiled with the DEBUGGING flag for this one:

```
> perl -Dr -e '/^pe(a)*rl$/i'
Compiling REx '^pe(a)*rl$'
size 17 first at 2
rarest char
 at 0
   1: BOL(2)
   2: EXACTF <pe>(4)
   4: CURLYN[1] {0,32767}(14)
   6:   NOTHING(8)
   8:   EXACTF <a>(0)
  12:   WHILEM(0)
  13: NOTHING(14)
  14: EXACTF <rl>(16)
  16: EOL(17)
  17: END(0)
floating ''$ at 4..2147483647 (checking floating) stclass 'EXACTF <pe>'
anchored(BOL) minlen 4
Omitting $' $& $' support.

EXECUTING...

Freeing REx: '^pe(a)*rl$'
```

Did you really want to know? :-) For more gory details on getting regular expressions to work, have a look at *perlre*, *perlretut*, and to decode the mysterious labels (BOL and CURLYN, etc. above), see *perldebguts*.

OUTPUT TIPS

To get all the output from your error log, and not miss any messages via helpful operating system buffering, insert a line like this, at the start of your script:

```
$|=1;
```

To watch the tail of a dynamically growing logfile, (from the command line):

```
tail -f $error_log
```

Wrapping all die calls in a handler routine can be useful to see how, and from where, they're being called, *perlvar* has more information:

```
BEGIN { $SIG{__DIE__} = sub { require Carp; Carp::confess(@_) } }
```

Various useful techniques for the redirection of STDOUT and STDERR filehandles are explained in *perlopentut* and *perlfaq8*.

CGI

Just a quick hint here for all those CGI programmers who can't figure out how on earth to get past that 'waiting for input' prompt, when running their CGI script from the command-line, try something like this:

```
 > perl -d my_cgi.pl -nodebug
```

Of course *CGI* and *perlfaq9* will tell you more.

GUIs

The command line interface is tightly integrated with an **emacs** extension and there's a **vi** interface too.

You don't have to do this all on the command line, though, there are a few GUI options out there. The nice thing about these is you can wave a mouse over a variable and a dump of its data will appear in an appropriate window, or in a popup balloon, no more tiresome typing of 'x $varname' :-)

In particular have a hunt around for the following:

ptkdb perlTK based wrapper for the built-in debugger

ddd data display debugger

PerlDevKit and **PerlBuilder** are NT specific

NB. (more info on these and others would be appreciated).

SUMMARY

We've seen how to encourage good coding practices with **use strict** and **-w**. We can run the perl debugger **perl -d scriptname** to inspect your data from within the perl debugger with the **p** and **x** commands. You can walk through your code, set breakpoints with **b** and step through that code with **s** or **n**, continue with **c** and return from a sub with **r**. Fairly intuitive stuff when you get down to it.

There is of course lots more to find out about, this has just scratched the surface. The best way to learn more is to use perldoc to find out more about the language, to read the on-line help (*perldebug* is probably the next place to go), and of course, experiment.

SEE ALSO

perldebug, *perldebguts*, *perldiag*, *dprofpp*, *perlrun*

AUTHOR

Richard Foley <richard@rfi.net>

CONTRIBUTORS

Various people have made helpful suggestions and contributions, in particular:

Ronald J Kimball <rjk@linguist.dartmouth.edu>

Hugo van der Sanden <hv@crypt0.demon.co.uk>

Peter Scott <Peter@PSDT.com>