Using The Perl Debugger

By icarus
2003−06−25

Printed from DevShed.com
URL: http://www.devshed.com/Server_Side/Perl/PerlDebugger/

The Dark Side

Perl aficionados might find this hard to believe, but to a lot of people, Perl is − to put it bluntly − one of the most terrifying languages on the planet.

There are a number of reasons for this, and only some of them are unwarranted. Novice developers find the language's sometimes−complex syntax intimidating, and its freewheeling "there's more than one way to do it" attitude confusing. The special symbols and variables that experienced Perl developers sprinkle liberally around their code (can you say @_?), coupled with the intricacies of Perl variable scoping, regular expressions, object−oriented programming and modules, can leave them bewildered and weak at the knees, and the language's terse error messages only serve to increase their frustration when things go wrong.

I can sympathize. Not so long ago, I too was one of these terrified souls, lost in the darkness of the Perl universe. Until the day a friend introduced me to the Perl debugger. And then there was light.

Start Me Up

Most casual Perl users aren't even aware that Perl ships with a debugger, but it does...and it's quite a good one too. In addition to allowing you to arbitrarily jump around your code, it also allows you to execute statements on a line−by−line basis, set breakpoints and watch−expressions, inspect variables and trace program flow (among other things).

In order to demonstrate the features of the debugger, I'll be using some sample Perl scripts. Here's the first one, a script which accepts an identifier as input argument, connect to a database, retrieves a list of email addresses and sends them email.

```
#!/usr/bin/perl

# load module
use DBI();

unless($ARGV[0])
{
print "ERROR: Please provide a valid input ID\n\n";
}

# get user input into variable
my $id = $ARGV[0];
```

```
# connect
my $dbh = DBI−>connect("DBI:mysql:database=db198;host=localhost", "root", "secret", {'RaiseError' => 1})
or die ("Cannot connect to database");

# set up email message
my $sendmail = "/usr/sbin/sendmail −t";
my $reply_to = "Reply−to: foo\@bar.org";
my $subject = "Subject: FOOBAR";
my $content = "Hello and how are you doing? This is the message body";

# now retrieve email addresses and send mail to each
my $sth = $dbh−>prepare("SELECT email FROM users WHERE users.id = '$id'"); $sth−>execute();
while(my $ref = $sth−>fetchrow_hashref()) {
my $to = $ref−>{'email'};
open(SENDMAIL, "|$sendmail") or die "Cannot send mail: $!";
print SENDMAIL $reply_to ."\n";
print SENDMAIL $subject . "\n";
print SENDMAIL "To: " . $to . "\n";
print SENDMAIL "Content−type: text/plain\n\n";
print SENDMAIL $content;
close(SENDMAIL);
}

# clean up
$dbh−>disconnect();
```

Perl's built−in debugger can be invoked simply by adding the −d option to your Perl command line, as below:

```
$ perl −d mailer.pl 877
Loading DB routines from perl5db.pl version 1.19
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(mailer.pl:6): unless($ARGV[0])
main::(mailer.pl:7): {
DB<1>
```

This will pop you into the debugger and place you at the first line of your script.

The

```
DB<1>
```

marker is the debugger command prompt; this is where you will enter debugger commands. The number in angle braces following the prompt keeps track of the number of commands you've entered, so that you can easily reuse a previous command.

Commands entered at the prompt which are not debugger commands are evaluated as Perl statements, and executed − as illustrated below:

DB<2> print 8+10;

DB<3> 18

DB<3> $a=10/2

DB<4> print $a;

DB<5> 5

You can enter multi−line commands into the debugger by separating them with a backslash, as in the example below:

DB<2> @flavours = ("vanilla", "chocolate", "strawberry");

DB<3> for $f (@flavours) { \
cont: print "$f\n"; \
cont: };
vanilla
chocolate
strawberry

Help is available via the "h" command (don't be intimidated, most of the following will be explained over this article),

DB<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.

and you can obtain detailed help with the "h h" command

DB<1> h h

Help is currently only available for the new 580 CommandSet,
if you really want old behaviour, presumably you know what you're doing ?−)

T Stack trace.
s [expr] Single step [in expr].
n [expr] Next, steps over subroutine calls [in expr].
<CR> Repeat last n or s command.
r Return from current subroutine.
c [line|sub] Continue; optionally inserts a one−time−only breakpoint
at the specified position.
l min+incr List incr+1 lines starting at min.
l min−max List lines min through max.
l line List single line.
...

Once your script has completed exiting, you can restart it with the "R" command, as below,

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.
DB<10> R
Warning: some settings and command−line options may be lost!

Loading DB routines from perl5db.pl version 1.19
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(mailer.pl:6): unless($ARGV[0])
main::(mailer.pl:7): {
DB<10>

or exit the debugger (this works even during script execution) by typing "q", and you will be immediately returned to your command prompt.

DB<1> q

Don't do that just yet, though − after all, you're here to learn more about this strange and wonderful new animal, and it would be rude to leave so quickly. Next up, I'll show you how to step through a script and execute it on a line−by−line basis.

* Moving Up

Now that you're in the debugger, let's start with something basic: moving around your script. Type "l" at the command prompt, and watch as the debugger goes busily to work listing the first ten lines of your script.

DB<1> l
6==> unless($ARGV[0])
7 {
8: print "ERROR: Please provide a valid input ID\n\n";
9 }
10
11 # get user input into variable
12: my $id = $ARGV[0];
13
14 # connect
15 # fix these as per your local settings

Typing "l" again will take you to the next screen of code,

DB<1> l
16==> my $dbh = DBI−>connect("DBI:mysql:database=db198;host=localhost",
"secret", "", {'RaiseError' => 1}) or die ("Cannot connect to database"); 17
18 # set up email message
19: my $sendmail = "/usr/sbin/sendmail −t";
20: my $reply_to = "Reply−to: foo\@bar.org";

21: my $subject = "Subject: FOOBAR";
22: my $content = "Hello and how are you doing? This is the message body";
23
24 # now retrieve email addresses and send mail to each
25: my $sth = $dbh−>prepare("SELECT email FROM users WHERE users.id =
'$id'");

and so on.

You can have the debugger display a specific line by specifying a line number to the "l" command,

DB<3> l 19
19==> my $sendmail = "/usr/sbin/sendmail −t";

or even display a block of lines by specifying a range of line numbers.

DB<4> l 20−25
20==> my $reply_to = "Reply−to: foo\@bar.org";
21: my $subject = "Subject: FOOBAR";
22: my $content = "Hello and how are you doing? This is the message body";
23
24 # now retrieve email addresses and send mail to each
25: my $sth = $dbh−>prepare("SELECT email FROM users WHERE users.id =
'$id'");

You can display the previous line with the "−" command,

DB<5> −
10==>
11 # get user input into variable
12: my $id = $ARGV[0];
13
14 # connect
15 # fix these as per your local settings
16: my $dbh = DBI−>connect("DBI:mysql:database=db198;host=localhost",
"root", "secret", {'RaiseError' => 1}) or die ("Cannot connect to database"); 17
18 # set up email message
19: my $sendmail = "/usr/sbin/sendmail −t";

or return to the last executed line with the "." command.

DB<5> .

main::(mailer.pl:16): my $dbh =
DBI−>connect("DBI:mysql:database=db198;host=localhost", "root",
DBI−>"secret",
{'RaiseError' => 1}) or die ("Cannot connect to database");

You can view the code above and below a specific line with the "v" command,

DB<3> v 27
24: my $description = $ref−>{'description'};
25: $sth−>finish();
26
27 # now use the group ID to get the group text
28: my $sth = $dbh−>prepare("SELECT group_text FROM groups WHERE
group_id =
'$gid'");
29: $sth−>execute();
30: my $ref = $sth−>fetchrow_hashref();
31: my $gtext = $ref−>{'group_text'};
32: $sth−>finish();
33

or even perform basic search operations within the debugger.

DB<3> /FOOBAR
21: my $subject = "Subject: FOOBAR";

This ability to move around your script comes in very handy when dealing with large and complex scripts.

Step By Step

When you first enter the debugger, you're placed at the first executable line of your Perl script.

$ perl −d mailer.pl 877
Loading DB routines from perl5db.pl version 1.19
Editor support available.

Enter h or `h h' for help, or `man perldebug' for more help.

main::(mailer.pl:6): unless($ARGV[0])
main::(mailer.pl:7): {
DB<1>

It's important to note that this line has not yet been executed; rather, it is the line that will be executed next. This is a very common mistake made by newbies to the Perl debugger, so be warned!

In order to execute the next line of the script, type "s".

```
main::(mailer.pl:6): unless($ARGV[0])
main::(mailer.pl:7): {
DB<1> s
main::(mailer.pl:12): my $id = $ARGV[0];
DB<1> s
main::(mailer.pl:16): my $dbh =
DBI−>connect("DBI:mysql:database=db198;host=localhost", "root",
DBI−>"secret",
{'RaiseError' => 1}) or die ("Cannot connect to database");
DB<1> s DBI::connect(/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm
:442):
442: my $class = shift;
DB<1> s DBI::connect(/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm
:443):
443: my($dsn, $user, $pass, $attr, $old_driver) = @_;
DB<1> s DBI::connect(/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm
:444):
444: my $driver;
DB<1> s DBI::connect(/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm
:445):
445: my $dbh;
DB<1> s DBI::connect(/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm
:448):
448: ($old_driver, $attr) = ($attr, $old_driver) if $attr and
!ref($attr);
DB<1> s
```

The "s" command steps through each line of the script, ducking into subroutines as and when needed − as clearly illustrated in the snippet above (note how the status message next to each line of the script changes to display which subroutine the script is currently executing).

If you'd like to "step over" subroutines, you can use the "n" command instead, which executes subroutines as a single step. This is useful if you have a large number of subroutine calls in your code and are more interested in general program flow than specific or isolated problems.

```
main::(mailer.pl:6): unless($ARGV[0])
main::(mailer.pl:7): {
DB<1> n
main::(mailer.pl:12): my $id = $ARGV[0];
```

DB<1> n
main::(mailer.pl:16): my $dbh =
DBI−>connect("DBI:mysql:database=test;host=localhost", "root", "",
{'RaiseError' => 1}) or die ("Cannot connect to database");
DB<1> n
nmain::(mailer.pl:19): my $sendmail = "/usr/sbin/sendmail −t";
DB<1>
main::(mailer.pl:20): my $reply_to = "Reply−to: foo\@bar.org";
DB<1> n
main::(mailer.pl:21): my $subject = "Subject: FOOBAR";
DB<1> n
main::(mailer.pl:22): my $content = "Hello and how are you doing?
This is the message body";

You can hit the "Enter" key repeatedly to repeat the last "n" or "s" command.

You can use the "l" command you learnt on the previous page to view the contents of a subroutine, simply by specifying the package/subroutine name after the "l" command.

DB<11> l DBI::connect
Switching to file '/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm'.
441 sub connect {
442: my $class = shift;
443: my($dsn, $user, $pass, $attr, $old_driver) = @_;
444: my $driver;
445: my $dbh;
446
447 # switch $old_driver<−>$attr if called in old style
448: ($old_driver, $attr) = ($attr, $old_driver) if $attr and
!ref($attr);
449
450: my $connect_meth = (ref $attr) ? $attr−> :
undef;

If you're currently inside a subroutine and would like to execute all its remaining statements until it generates a return value, simply use the "r" command. The debugger will continue through the subroutine, and print the return value from the subroutine once it finishes execution.

DB<1> s DBI::connect(/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm
:443):
443: my($dsn, $user, $pass, $attr, $old_driver) = @_;
DB<1> r
scalar context return from DBI::connect: empty hash
main::(mailer.pl:19): my $sendmail = "/usr/sbin/sendmail −t";

Digging Deeper

One of the Perl debugger's most oft−used features is its ability to X−ray any variable or object and display its contents. This is accomplised via the "x" command, as illustrated in the following example:

DB<7> @friends = qw(Rachel Ross Joey Monica Chandler Phoebe);

DB<8> x @friends
0 'Rachel'
1 'Ross'
2 'Joey'
3 'Monica'
4 'Chandler'
5 'Phoebe

You can use the "p" command to print the value of any variable,

DB<4> p $sendmail
/usr/sbin/sendmail −t

DB<7> $movie="Star Wars"

DB<8> p $movie
Star Wars

or the "V" command, followed by a package name, to view all the variables in that package.

DB<32> V DBI
%DBI_methods = (
'db' => HASH(0x81e292c)
'CLEAR' => HASH(0x8281ff8)
'O' => 4
'DESTROY' => undef
'EXISTS' => HASH(0x8281ff8)
−> REUSED_ADDRESS
'FETCH' => HASH(0x808b198)
'O' => 1028
'FIRSTKEY' => HASH(0x8281ff8)
−> REUSED_ADDRESS
'NEXTKEY' => HASH(0x8281ff8)
−> REUSED_ADDRESS
'STORE' => HASH(0x8284968)

'O' => 1040
'_not_impl' => undef
'begin_work' => HASH(0x83155a8)
'O' => 1024
'U' => ARRAY(0x831556c)
0 1
1 2
2 '[\%attr]'
'column_info' => HASH(0x81e2680)
'O' => 512
'U' => ARRAY(0x81e2644)
0 1
1 6
2 '$catalog, $schema, $table, $column [, \%attr]'
'commit' => HASH(0x8315608)
'O' => 1152
'U' => ARRAY(0x83155d8)
0 1
1 1
...

The "X" command displays a complete list of all the variables the Perl script knows about, including environment and shell variables, special Perl built−ins and variables local to the script itself.

DB<32> X
FileHandle(STDIN) => fileno(0)
$^V = "\cE\cH\c@"
= ''
= 'main'
$movie = 'Star Wars'
$^WARNING_BITS = "\c@\c@\c@\c@\c@\c@\c@\c@\c@\c@\c@\c@"
$< = 515
FileHandle(stdin) => fileno(0)
@ARGV = (
0 'ueruir'
)
@INC = (
0 '/usr/lib/perl5/5.8.0/i386−linux−thread−multi'
1 '/usr/lib/perl5/5.8.0'
2 '/usr/lib/perl5/site_perl/5.8.0/i386−linux−thread−multi'
3 '/usr/lib/perl5/site_perl/5.8.0'
4 '/usr/lib/perl5/site_perl'
5 '/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi'
6 '/usr/lib/perl5/vendor_perl/5.8.0'
7 '/usr/lib/perl5/vendor_perl'

8 '.'
)
%INC = (
'AutoLoader.pm' => '/usr/lib/perl5/5.8.0/AutoLoader.pm'
'Carp.pm' => '/usr/lib/perl5/5.8.0/Carp.pm'
'Carp/Heavy.pm' => '/usr/lib/perl5/5.8.0/Carp/Heavy.pm'
'Config.pm' => '/usr/lib/perl5/5.8.0/i386−linux−thread−multi/Config.pm'
'DBD/mysql.pm' => '/usr/lib/perl5/site_perl/5.8.0/i386−linux−thread−multi/DBD/mysql.pm'
...

You can also use the "M" command to display a list of all loaded modules (together with version numbers),

DB<32> M
'AutoLoader.pm' => '5.59 from /usr/lib/perl5/5.8.0/AutoLoader.pm'
'Carp.pm' => '1.01 from /usr/lib/perl5/5.8.0/Carp.pm' 'Carp/Heavy.pm' =>
'/usr/lib/perl5/5.8.0/Carp/Heavy.pm'
'Config.pm' => '/usr/lib/perl5/5.8.0/i386−linux−thread−multi/Config.pm'
'DBD/mysql.pm' => '2.0416 from /usr/lib/perl5/site_perl/5.8.0/i386−linux−thread−multi/DBD/mysql.pm'
'DBI.pm' => '1.30 from /usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm'
'DynaLoader.pm' => '1.04 from /usr/lib/perl5/5.8.0/i386−linux−thread−multi/DynaLoader.pm'
'Exporter.pm' => '5.566 from /usr/lib/perl5/5.8.0/Exporter.pm' 'Exporter/Heavy.pm' => '5.566 from
/usr/lib/perl5/5.8.0/Exporter/Heavy.pm'
'Term/Cap.pm' => '1.07 from /usr/lib/perl5/5.8.0/Term/Cap.pm' 'Term/ReadLine.pm' => '1.00 from
/usr/lib/perl5/5.8.0/Term/ReadLine.pm'
'dumpvar.pl' => '/usr/lib/perl5/5.8.0/dumpvar.pl'
'perl5db.pl' => '1.19 from /usr/lib/perl5/5.8.0/perl5db.pl' 'strict.pm' => '1.02 from
/usr/lib/perl5/5.8.0/strict.pm' 'vars.pm' => '1.01 from /usr/lib/perl5/5.8.0/vars.pm' 'warnings.pm' => '1.00
from /usr/lib/perl5/5.8.0/warnings.pm' 'warnings/register.pm' => '1.00 from
/usr/lib/perl5/5.8.0/warnings/register.pm'

and the "m" command, followed by a package name, to view the methods available in that package.

DB<33> m DBI
CLONE
_clone_dbis
_dbtype_names
_debug_dispatch
_rebless
_rebless_dbtype_subclass
_set_isa
_setup_driver
available_drivers
bootstrap
carp

confess
connect
connect_cached
connect_test_perf
constant
croak
data_sources
dbi_profile
via Exporter: export_ok_tags
via Exporter: export_tags
via Exporter: export_to_level
via DynaLoader: AUTOLOAD
via DynaLoader: boot_DynaLoader
via DynaLoader: dl_install_xsub
...

Finally, you can obtain a complete list of subroutines in your script (and all its linked packages) with the "S" command,

DB<35> S
AutoLoader::AUTOLOAD
AutoLoader::BEGIN AutoLoader::__ANON__[/usr/lib/perl5/5.8.0/AutoLoader.pm:96]
AutoLoader::import
AutoLoader::unimport
Carp::BEGIN
Carp::caller_info
Carp::carp
Carp::cluck
Carp::confess
Carp::croak
Carp::export_fail
Carp::format_arg
Carp::get_status
DBD::mysql::AUTOLOAD
DBD::mysql::BEGIN
DBD::mysql::_OdbcParse
DBD::mysql::_OdbcParseHost
DBD::mysql::db::ANSI2db
DBD::mysql::db::BEGIN
DBD::mysql::db::_SelectDB
DBD::mysql::db::admin
DBI::BEGIN
DBI::CLONE
DBI::DBI_tie::STORE
...

and filter that list down to a specific subset by adding a search pattern.

DB<35> S connect
DBD::_::db::disconnect
DBD::_::dr::connect
DBD::_::dr::connect_cached
DBD::_::dr::disconnect_all
DBD::mysql::dr::connect
DBI::connect
DBI::connect_cached
DBI::connect_test_perf
DBI::disconnect
DBI::disconnect_all

Breaking Free

The Perl debugger also allows you to define a variety of triggers within your script; these triggers − breakpoints, watch−expressions and actions − come in handy when you need to keep an eye on the values of different variables, since they can be set to automatically notify you in the event of a change.

Breakpoints can be set with the "b" command, which may be followed with either a subroutine name or line number. The following line sets a breakpoint at the first line of the subroutine install_driver() of the DBI module:

DB<14> b DBI::install_driver
DB<15> L
/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm:
576: my $class = shift;
break if (1)

When the debugger reaches this line, it will halt and wait for a new command.

DB<15> n
main::(mailer.pl:12): my $id = $ARGV[0];
DB<15> n
main::(mailer.pl:16): my $dbh =
DBI−>connect("DBI:mysql:database=db198;host=localhost", "root",
DBI−>"secret",
{'RaiseError' => 1}) or die ("Cannot connect to database");
DB<15> n DBI::install_driver(/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi
/DBI.pm:576):
576: my $class = shift;

DB<15>

You can set a one−time breakpoint with the "c" command; this command continues executing the script until the breakpoint is reached, at which point it stops and waits for a command.

DB<1> c 19
main::(mailer.pl:19): my $sendmail = "/usr/sbin/sendmail −t";

You can even place a breakpoint within your Perl code, simply by adding the line

```
$DB::single=1;
```

or

```
$DB::single=2;
```

at the appropriate point in your script. When the debugger encounters this statement while running the script, it will automatically halt and wait for a new command.

You can delete breakpoints with the "B" command, which needs either a line number

DB<8> B 19

or the * wildcard to delete all breakpoints.

*DB<9> B **
Deleting all breakpoints...

A Watchful Eye

You can set a watch−expression (an expression which triggers a halt if its value changes) with the "w" command, as in the following example:

DB<1> w $count
DB<<3>> L
Watch−expressions:

$count

Now that the watch−expression has been set, you can see how it works by writing some code to alter the value of the $count variable. The debugger will display a message every time the variable's value changes.

DB<2> for ($count=0; $count<=10; $count++) { print "Hello!"; };
Watchpoint 0: $count changed:
old value: undef
new value: '0'
DB<<3>> s
main::((eval 13)[/usr/lib/perl5/5.8.0/perl5db.pl:17]:2):
2: for ($count=0; $count<=10; $count++) { print "Hello!"; };;
DB<<3>>
Watchpoint 0: $count changed:
old value: '0'
new value: '1'
main::((eval 13)[/usr/lib/perl5/5.8.0/perl5db.pl:17]:2):
2: for ($count=0; $count<=10; $count++) { print "Hello!"; };;
DB<<3>> Hello!
main::((eval 13)[/usr/lib/perl5/5.8.0/perl5db.pl:17]:2):
2: for ($count=0; $count<=10; $count++) { print "Hello!"; };;
DB<<3>> s
Watchpoint 0: $count changed:
old value: '1'
new value: '2'
for ($count=0; $count<=10; $count++) { print "Hello!"; };

Watch−expressions can be deleted with the "W" command; either specify the expression

DB<<3>> W $count

or delete all currently−set expressions with the * wildcard.

*DB<<4>> W **
Deleting all watch expressions ...

Note that adding watch−expressions can result in a performance penalty, so you should try and restrict yourself to not more than three or four at any given time.

Acts Of Madness

The Perl debugger also allows you to define actions − code that is executed when a specific line is reached. This comes in handy when you need to try a piece of code with different variable values, or print notification before specific lines are executed.

Actions are set with the "a" command, which needs a line number and a line of code to be executed before the debugger executes the specified line. Consider the following example, which demonstrates:

```
DB<7> l 24−44
24==> foreach $line (@file)
25 {
26 # split on record delimiter
27: @record = split(":", $line);
28
29 # print username if UID >= 500
30: if ($record[2] >= 500)
31 {
32: print "$record[0]($record[2])\n";
33: $count++
34 }
35 }
DB<10> b 30
DB<11> a 30 print "Checking UID $record[2]\n"
```

In this case, when the debugger reaches line 30, it will automatically halt (because I set a breakpoint) and also print the value of the second element of the array $record (because I told it to via the "a" command).

```
DB<21> c
Checking UID 3
main::(passwd.pl:30): if ($record[2] >= 500)
main::(passwd.pl:31): {
DB<21> c
Checking UID 534
all(534)
main::(passwd.pl:30): if ($record[2] >= 500)
main::(passwd.pl:31): {
Checking UID 516
...
```

Actions can be deleted with the "A" command, which can be supplied with either a line number

```
DB<12> A 27
```

or the * wildcard.

*DB<13> A **
Deleting all actions...

* Following The Trail

As you might have guessed from my frequent use of this command in some of the previous examples, the "L" command provides a list of all currently−set breakpoints, actions and watch−expressions.

DB<4> L
passwd.pl:
27: @record = split(":", $line);
break if (1)
action: print $line
Watch−expressions:
@record

You can also obtain a list of previously−entered debugger commands with the "H" command, which displays a command history,

DB<7> H
6: b 11
*5: A **
*4: B **
3: l 19−25
2: v 18
1: l 12

and even repeat previous commands with the "!" command.

DB<7> ! 4
*B **
Deleting all breakpoints...

You can obtain a stack trace with the "T" command, as in the following example:

DB<5> T
. = DBI::_setup_driver('DBD::mysql') called from file `/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm' line 629 $ = DBI::install_driver('DBI', 'mysql') called from file `/usr/lib/perl5/vendor_perl/5.8.0/i386−linux−thread−multi/DBI.pm' line 497 $ = DBI::connect('DBI', 'DBI:mysql:database=db198;host=localhost', 'root', 'secret', ref(HASH)) called from file

`mailer.pl' line 16

This is fairly easy to decipher, so long as you start with the last line first. In the example above, it's clear to see that the script called the
DBI::connect() function, which called DBI::install_driver(), which in turn invoked DBI::_setup_driver() from the DBI.pm module. This stack trace is useful to evaluate the control flow within your program and the libraries it interacts with.

You can also customize the debugger with the "o" command, which lets you alter debugger options. Try typing it at the debugger prompt and you should see something like this:

DB<1> o
hashDepth = 'N/A'
arrayDepth = 'N/A'
CommandSet = '580'
dumpDepth = 'N/A'
DumpDBFiles = 'N/A'
DumpPackages = 'N/A'
DumpReused = 'N/A'
compactDump = 'N/A'
veryCompact = 'N/A'
quote = 'N/A'
HighBit = 'N/A'
undefPrint = 'N/A'
globPrint = 'N/A'
PrintRet = '1'
UsageOnly = 'N/A'
frame = '0'
AutoTrace = '0'
TTY = '/dev/tty'
noTTY = 'N/A'
ReadLine = '1'
NonStop = '0'
LineInfo = '/dev/tty'
maxTraceLen = '400'
recallCommand = '!'
ShellBang = '!'
pager = '|/usr/bin/less −isr'
tkRunning = 'N/A'
ornaments = 'us,ue,md,me'
signalLevel = '1'
warnLevel = '1'
dieLevel = '1'
inhibit_exit = '1'
ImmediateStop = 'N/A'

bareStringify = 'N/A'
CreateTTY = '3'
RemotePort = 'N/A'
windowSize = '10'

As you may have guessed, these are all internal debugger options which can be customized as per your requirements. To alter a particular options, simply use the "o" command followed by the option name and its value, as
below:

DB<4> o windowSize=15
windowSize = '15'

Here's a brief list of the more interesting options available for
customization:

"RecallCommand" − the command used to repeat previous commands ("!) by default);

"pager" − the program to use when paging through long screens of output;

"hashDepth" − the depth to which hashes are displayed;

"arrayDepth" − the depth to which arrays are displayed;

"dumpDepth" − the depth to which structures are displayed;

"inhibit_exit" − continue debugger session even after the script has ended;

"windowSize" − number of lines of code to display with "v" and "l" commands;

Test Drive

Now that you've seen what the debugger can do, let's take it out for a quick test drive. Consider the following simple Perl program, which contains a deliberate error:

```
#!/usr/bin/perl

# multiply two numbers
sub multiply()
{
my $a, $b = @_;
return $a * $b;
}
```

```
# set range for multiplication table
@values = (1..10);

# get number from command−line
foreach $value (@values)
{
print "$ARGV[0] x $value = " . &multiply($ARGV[0], $value) . "\n"; }
```

This is pretty simple − it accepts a number as input and creates a multiplication table for it. If, for example, you invoked it with the number 4 as argument, you'd expect a multiplication table like this:

4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40

Instead, what you get is this:

4 x 1 = 0
4 x 2 = 0
4 x 3 = 0
4 x 4 = 0
4 x 5 = 0
4 x 6 = 0
4 x 7 = 0
4 x 8 = 0
4 x 9 = 0
4 x 10 = 0

If you're an experienced Perl programmer, you already know the problem. Don't say a word; instead, just follow along as I run this through the debugger.

$ perl −d multiply.pl 4

```
Loading DB routines from perl5db.pl version 1.19
Editor support available.
```

Enter h or `h h' for help, or `man perldebug' for more help.

main::(multiply.pl:11): @values = (1..10);
DB<1>

Now, since I'm having a problem with the product of the two numbers, and that product is being calculated by the multiply() subroutine, it makes sense to set a breakpoint at that subroutine with the "b" command.

DB<1> b multiply

I can now step through the script with the "r" command, which will display the return value from the subroutine every time it is invoked.

DB<2> r
main::multiply(multiply.pl:6): my $a, $b = @_;
DB<2> r
scalar context return from main::multiply: 0
4 x 1 = 0
main::(multiply.pl:14): foreach $value (@values)
main::(multiply.pl:15): {
DB<2> r
main::multiply(multiply.pl:6): my $a, $b = @_;
DB<2> r
scalar context return from main::multiply: 0
4 x 2 = 0
main::(multiply.pl:14): foreach $value (@values)
main::(multiply.pl:15): {
DB<2> r
main::multiply(multiply.pl:6): my $a, $b = @_;
DB<2> r
scalar context return from main::multiply: 0
4 x 3 = 0
main::(multiply.pl:14): foreach $value (@values)
main::(multiply.pl:15): {

Hmmm...a return value of zero at every stage is obviously an indication that something is going seriously wrong inside the subroutine. This is probably the reason for the long string of zeroes in the output of my script. Now, if only we could look inside to see what was really going on...

DB<3> s
main::(multiply.pl:16): print "$ARGV[0] x $value = " .
&multiply($ARGV[0], $value) . "\n";
DB<3> s

main::multiply(multiply.pl:6): my $a, $b = @_;
DB<3> s
*main::multiply(multiply.pl:7): return $a * $b;*
DB<3> x $a
0 undef
DB<4> x $b
0 2

Curiouser and curiouser. As the "x" command clearly shows, the multiply() subroutine is obviously not using correct parameters when performing the
calculation: $a is undefined, $b is 2, and both are wrong.

Now, there are two possible reasons for this: either the subroutine is being passed incorrect values correctly from the main script, or the subroutine is not reading them properly.

Luckily, the first theory is easy enough to test − all I need to do is run the "x" command on the @_ array:

DB<6> s
*main::multiply(multiply.pl:7): return $a * $b;*
DB<6> x @_
0 4
1 5

Perfect. So the main script is obviously passing the input arguments correctly to the multiply() subroutine, as evidenced by the fact that the @_ array contains the correct values. The problem is therefore related to the extraction of these values from the @_ array into regular scalars − specifically, with line 6 of the script:

DB<10> l 6
6: my $a, $b = @_;

And now the error should be fairly obvious. As any Perl coder knows, when retrieving subroutine arguments from the @_ array into scalars, the list of scalars should be enclosed in parentheses. Corrected, the statement above would read:

DB<10> l 6
6: my ($a, $b) = @_;

Now, when I re−run the script and step through it, the "x" command shows that the subroutine's $a and $b scalars are being populated with the correct values.

DB<1> s
main::multiply(multiply.pl:6): my ($a, $b) = @_;

DB<1> s
*main::multiply(multiply.pl:7): return $a * $b;*
DB<1> x $a
0 4
DB<2> x $b
0 4

And when I run the script, the output now makes sense:

4 x 1 = 4
4 x 2 = 8
4 x 3 = 12
4 x 4 = 16
4 x 5 = 20
4 x 6 = 24
4 x 7 = 28
4 x 8 = 32
4 x 9 = 36
4 x 10 = 40

Experienced Perl programmers will, of course, point out that I could have saved myself all this trouble either by placing the

```
use strict();
```

construct at the top of my script, or by adding the "−w" switch when running the script from the command line.

$ perl −w multiply.pl
Use of uninitialized value in concatenation (.) or string at multiply.pl line 16. Use of uninitialized value in multiplication (*) at multiply.pl line 7. x 1 = 0 Use of uninitialized value in concatenation (.) or string at multiply.pl line 16. Use of uninitialized value in multiplication (*) at multiply.pl line 7. x 2 = 0 ...

Both these methods would have brought the error to my notice without requiring me to use the debugger. However, in some cases, especially cases involving logical errors that a syntax checker will not catch, the debugger can be an invaluable tool in diagnosing problems with your code, since it allows you to step through each line of code and inspect (or change) the variables being used at every stage to catch potentially−serious bugs.

And that's about all for the moment. As you saw over the last few pages, the Perl debugger is a powerful utility, one that can substantially reduce your stress levels when dealing with recalcitrant Perl code. Go ahead,

play with it a little...and, until next time, stay healthy!

Note: Examples in this article have been tested on Linux/i586 with Perl 5.8.0. Examples are illustrative only, and are not meant for a production environment. Melonfire provides no warranties or support for the source code described in this article. YMMV!