Table of Contents

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Chapter 6. Debugging

Hacks 51-59

Not all programs work the first time. Even if you use test-driven development and know exactly what you need to write and how to write it, you will eventually encounter code that you don't understand and which doesn't quite work right. One frequent (and frequently bad) problem-solving technique is voodoo programming, where you change a line or character here and there, hoping to stumble upon the correct incantation.

You can do better! Mastering a few Perl tricks and understanding a few tips can help you wrestle unwieldy code into submission. Amaze your coworkers. Save precious time. Find and fix failures faster! Here's how.

Hack 51. Find Compilation Errors Fast

Trace problem code as quickly as possible.

As helpful as Perl is, sometimes a missing semicolon, parenthesis, or closing quotation mark send it into a morass of confusion. Error messages clutter your logs or your console window and, try as hard as you might, you can't see what's not there or what's just a little bit wrong.

When trouble strikes, try a simple technique to zoom in on the error as quickly as possible.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The Hack

When your program really goes kablooey, the best thing to do is not to let Perl try to run it, even through your test programs. If things are going that badly wrong, take a tip from the Haskell world and convince yourself that if it at least compiles, it has to be fairly okay.[1] Just make it compile already!

[1] I kid because I like.

Go to the command line and tell Perl to compile the program with warnings and then stop. If your program is *what_went_wrong.pl*, use:

```
$ perl -wc what_went_wrong.pl
```

If there's no error, Perl will report a happy okay message. Great! Go on to making your tests pass. Otherwise, grab the first error message and figure out how to solve it.

Binary searching for bugs

What if that error message makes no sense? Perl does its best to figure out the offending line, but because of the flexible syntax that allows you to span multiple lines and use postfix conditional expressions, sometimes the best it can say is "something's wrong". In that case, narrow down your search time considerably with a binary search.

Pick a place somewhere in the middle of the file, preferably between subroutines or methods. Add the `__END__` token at the start of a line. Effectively this turns the rest of the code into data, so Perl will ignore it. Run `perl -wc` again. If the error message occurs, the error is in the first half of the file. If the error disappears, the error is in the second half of the file. Move the `__END__` token appropriately halfway between whichever end of the file has the error and your current position and try again.

Sometimes you can't move the token without breaking up a block, which definitely causes compilation errors. In that case, use a pair of `=cut` POD directives to comment out the offending code. Within a handful of iterations, you'll zero in on the problem and you should have an easier time deciding *how* to fix it.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hacking the Hack

This technique also works decently for figuring out where an error occurs, especially if you don't have logging or tracing statements in the code. Instead of commenting out code selectively, dump appropriate data with YAML or another serialization module at appropriate places to narrow down the error.

This approach often works, but it can fail when you do odd things, such as using a source filter. Beware.

Hack 52. Make Invisible Characters Apparent

See what your variables really contain.

Perl has a handful of good debugging techniques. For example, you can fire up the debugger [Hack #59] or write test cases [Hack #53]. If you're just experimenting, or need a quick-and-dirty answer right now, sometimes the easiest technique is to add a few `print( )` statements here and there.

This has its drawbacks, though, especially when the printed output *looks* correct but obviously isn't. Before you flip through the debugger documentation and rejig your debugging statements into test cases, consider a few tricks to make the invisible differences that your computer sees visible to you too. (Then make your test cases, use the debugger, and smarten your comments.)

Bracket Your Variables

A very common mistake is to forget to `chomp( )` data read from external sources. Suppose that you're processing a list of files read from another file:

```
while (<$file_list>)
{
    warn "Processing $_";
    next unless -e $_;
```

```
    process_file( $_ );
}
```

All of the files *look* correct in the `warn( )` output, but the `process_file( )` code never occurs.

`warn( )` is better than `print( )` because it goes to STDERR by default, which makes it redirectable separately.

Change the debugging line to make the filename more visible:

```
while (<$file_list>)
{
    warn "Processing '$_'";
    next unless -e $_;
    process_file( $_ );
}
```

Adding single quotes (or any other visible character) around the filename will likely reveal that all of the filenames within the loop have newlines at the end (or whatever the current input record separator, `$/`, contains). The solution is obvious:

```
while (<$file_list>)
{
    chomp;
    next unless -e $_;
    process_file( $_ );
}
```

Bracket Interpolated Lists

The previous technique only works well on scalar variables. Lists and arrays are more tricky. Fortunately, the special punctuation variable `$"` controls the separator used when interpolating a list into a string.

Suppose that you're writing a ranking system for a table-tennis league. You've come to the end of the program and you want to display the top ten players:

```
my @ranked;

for my $rank ( 1 .. 10 )
{
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
    $ranked[$rank] = get_player_by_rank( $rank );
}
```

Of course, players may tie—leaving two players at the third rank and no players at the fourth rank. The naïve approach of assuming the array contains exactly ten entries may fail, especially if `get_player_by_rank( )` always only returns a single player or potentially returns an array reference of multiple players.

Printing the array may be no help:

```
for my $rank ( 1 .. 10 )
{
    $ranked[$rank] = get_player_by_rank( $rank );
}

warn "Ranks: [@ranked]\\n";
```

Everything interpolates into a single string, leaving you to count to see which is missing.

Instead, set `$"` to a nice, visible string:

```
local $" = '] [';
warn "Ranks: [@ranked]\\n";
```

This puts the delimiters between the entries, making it much easier to see which slot is empty.

If only you could override stringification on that particular array and print the index as well as the element at that index...roll on, Perl 6!

Serialize Your Data

If this is too much for you to handle manually, bring in the big guns of a serialization module to do your debugging for you. `Data::Dumper` has enjoyed a long reign as a nice debugging aid, but `YAML` provides even more readability and conciseness. To see a nested data structure (or even a scalar, array, or hash without having to worry about the right delimiters), use the `Dump( )` function:

```
my $user = User->load( id => 54272 );
warn Dump( $user );
```

This prints a nice, compact representation of the data in the `$user` object without the excessive indentation and indirection that `Data::Dumper` can often provide.

Another good, if complex option, is using `Devel::Peek` to see exactly what Perl thinks of your variable. When you need it, you really need it. The rest of the time, take a breath, and then spend two minutes writing the test case.

Hack 53. Debug with Test Cases

Make exploratory code reusable.

Many programmers have subdirectories full of little test snippets; it's common to write a few programs to explore a feature of the language or a new library. It's also common to do this with false laziness, eyeballing output and tweaking an idea here or there.

Usually that's okay, but occasionally you *know* you wrote code to explore something you need to know *right now*—if only you could find it and decipher what you were thinking.

If you know how to write test cases with Perl's standard testing tools, you can end this madness and make even your experiments reusable and maintainable.

The Hack

Suppose you've just learned that Perl's default sorting algorithm changed from unstable to stable for Perl 5.8.0. The Internet reveals that, with a stable sorting algorithm, elements that have the same position in the sorting order will retain the positions relative to each other that they had in the input.

Writing test code

What does that really mean in practice? It's time to write some code:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
my @elements =
(
    [ 2, 2 ], [ 2, 1 ], [ 2, 0 ],
    [ 1, 0 ], [ 1, 1 ], [ 1, 2 ],
);

my @sorted   = sort { $a->[0] <=> $b->[0] } @elements;

local $"     = ', ';
print "[ @$_ ]\\n" for @sorted;
```

A stable sorting algorithm should produce the output:

```
[ 1, 0 ]
[ 1, 1 ]
[ 1, 2 ]
[ 2, 2 ]
[ 2, 1 ]
[ 2, 0 ]
```

Because the algorithm sorts only on the first element, all of the ones should come before the twos. Because the algorithm is stable, all of the second values of the ones should increase and all of the second values of the twos should decrease.

From test code to test cases

Of course, six months later that code may be somewhat impenetrable. It has decent variable names, but it's quick and dirty and likely uncommented. What does it prove? Why? Even worse, the first time it ran it has no debugging information—it's easy to misread the output when flipping back and forth between it and the code to recreate the algorithm in your head.

That's the point of test cases: removing tedium and making expectations clear, unambiguous, and automatable. Here's the same file rewritten as executable tests:

```
use Test::More tests => 4;

my @elements =
(
    [ 2, 2 ], [ 2, 1 ], [ 2, 0 ],
    [ 1, 0 ], [ 1, 1 ], [ 1, 2 ],
);
my @sorted   = sort { $a->[0] <=> $b->[0] } @elements;

is( $sorted[0][0], 1, 'numeric sort should put 1 before 2'     );
is( $sorted[0][1], 0, '... keeping stability of original list' );
is( $sorted[2][1], 2, '... through all elements'               );
is( $sorted[3][1], 2, '... not accidentally sorting them'      );
```

With a little more work up front, your expectations are clearer. If there's a failure, you see where it fails without having to trace the algorithm in your head again. You can also see

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

which part of your assumptions (or code) failed in detail as fine-grained as you care to test. Even better, you can add more tests to check further behavior, such as mingling the definition of the ones and twos further.

In case an upgrade changes the behavior of your production code, you can also run the test cases to narrow down the problem.

Hacking the Hack

Ideally, someone's already tested this sort of code—the Perl 5 porters. If you have access to the source code of Perl (in this case) or the library you're testing, you can skim the test suite for examples to borrow and modify or learn from outright. In this case, code in *t/op/sort.t* tests Perl's stable sort. Even just skimming the test descriptions can reveal a lot of information about the ideas behind the implementation.

Hack 54. Debug with Comments

Let your documentation help you remove bugs.

There are two types of people who debug code: those who fire up Perl's built-in debugger and those who sprinkle `print` statements through their code. If you're in the second group, you probably know that one big problem with debugging by hand is that, once you remove the bugs, you have to go through and remove all the debugging statements as well.

What if you could safely leave them in the code? After all, if you needed them once, you'll probably need them again, when the next bug appears.

The Hack

"Something left in the code, but ignored" is pretty much the definition of a comment, so it's no surprise that you can use comments to turn off debugging statements. There's a much more interesting alternative, however: using comments to turn *on* debugging statements.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The `Smart::Comments` CPAN module does just that: it turns comments *into* debugging statements.

Displaying variables

When you `use Smart::Comments`, any subsequent comment with three or more leading `#`s becomes a debugging statement and prints whatever the comment says to STDERR. For example, if you can't work out why your `@play_calls` variable is getting more elements than you expected:

```
my $call       = "26, 17, 22, hut!";
my @play_calls = split /\\s*,?\\s*/, $call;
```

insert some smart comments to report what's happening:

```
# make '###' magical...
use Smart::Comments;

my $call       = "26, 17, 22, hut!";

### $call

my @play_calls = split /\\s*,?\\s*/, $call;

### @play_calls
```

When you run that code, `Smart::Comments` will find the triple-`#` comments and print out whatever they contain:

```
$ perl play_book.pl

### $call: '26, 17, 22, hut!'

### @play_calls: [
###                 '2',
###                 '6',
###                 '1',
###                 '7',
###                 '2',
###                 '2',
###                 'h',
###                 'u',
###                 't',
###                 '!'
###               ]

$
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Immediately you can see that the `split` is splitting your text at every single character (because your splitting pattern, `/\\s*,?\\s*/`, matches an empty string so `split` splits everywhere).

The real smartness comes in, however, when you write more structured comments:

```
use Smart::Comments;

my $call       = "26, 17, 22, hut!";

### input: $call

my @play_calls = split /\\s*,?\\s*/, $call;

### split to: @play_calls
```

which produces the output:

```
$ perl play_book.pl

### input: '26, 17, 22, hut!'

### split to: [
###               '2',
###               '6',
###               '1',
###               '7',
###               '2',
###               '2',
###               'h',
###               'u',
###               't',
###               '!'
###             ]

$
```

Making Assertions

Even more useful, the module also allows you to write comments that act like assertions:

```
use Smart::Comments;

my $call       = "26, 17, 22, hut!";

my @play_calls = split /\\s*,?\\s*/, $call;

#### require: @play_calls = = 4
```

Assertion comments like this only produce a report (and an exception!) if the assertion fails. In that case, the smartness really shows through, because the smart comments not only

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

report the failure, but they also automatically report all the variables used in the test, so you can see *why* the assertion failed:

```
$ perl play_book_with_assertion.pl

### @play_calls = = 4 was not true at play_book_with_assertion.pl line 7.
###     @play_calls was: [
###                        '2',
###                        '6',
###                        '1',
###                        '7',
###                        '2',
###                        '2',
###                        'h',
###                        'u',
###                        't',
###                        '!'
###                      ]

$
```

Best of all, when you finish debugging, you can switch off all the debugging statements simply by removing—or just commenting out—the `use Smart::Comments` statement:

```
# use Smart::Comments;

my $call       = "26, 17, 22, hut!";

### input: $call

my @play_calls = split /\\s*,?\\s*/, $call;

### split to: @play_calls
```

Because the code no longer loads the module, triple-# comments are no longer special. They remain ordinary comments, and Perl consequently ignores them:

```
$ perl play_book.pl

$
```

Configuring smartness levels

By the way, you might have noticed that the `require:` assertion in the third example used `####` instead of `###` as its comment introducer. Using differing numbers of #s allows you to be selective about turning smart comments on and off. If you load the module and explicitly tell it which comment introducers are smart, then it will only activate comments with those particular introducers. For example:

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
use Smart::Comments '####';   # Only ####... comments are "smart"
                              # Any ###... comments are ignored

my $call       = "26, 17, 22, hut!";

### $call

my @play_calls = split /\\s*,?\\s*/, $call;

### @play_calls

#### require: @play_calls = = 4
```

This final example turns off the debugging statements, leaving only the assertion active.

If editing your source code to enable and disable `Smart::Comments` is too onerous, consider making a shell alias [Hack #4] to load the module and execute a named program. The appropriate command line to run a program with `Smart::Comments` enabled is:

```
$ perl -MSmart::Comments split_test.pl
```

To activate only specific comment introducers, as in the earlier example, write:

```
$ perl -MSmart::Comments="" split_test.pl
```

with the appropriate number of `#` characters in the quotes.

Hack 55. Show Source Code on Errors

Don't guess which line is the problem—see it!

Debugging errors and warning messages isn't often fun. Instead, it can be tedious. Often even finding the problem takes too long.

Perl can reveal the line number of warnings and errors (with `warn` and `die` and the `warnings` pragma in effect); why can't it show the source code of the affected line?

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The Hack

The code to do this is pretty easy, if unsubtle:

```
package SourceCarp;

use strict;
use warnings;

sub import
{
    my ($class, %args) = @_;

    $SIG{__DIE__}  = sub { report( shift, 2 ); exit } if $args{fatal};
    $SIG{__WARN__} = \\&report                         if $args{warnings};
}

sub report
{
    my ($message, $level)  = @_;
    $level               ||= 1;
    my ($filename, $line)  = ( caller( $level - 1 ) )[1, 2];
    warn $message, show_source( $filename, $line );
}

sub show_source
{
    my ($filename, $line) = @_;
    return '' unless open( my $fh, $filename );

    my $start = $line - 2;
    my $end   = $line + 2;

    local $.;
    my @text;
    while (<$fh>)
    {
        next unless $. >= $start;
        last if     $. >  $end;
        my $highlight   = $. = = $line ? '*' : ' ';
        push @text, sprintf( "%s%04d: %s", $highlight, $., $_ );
    }

    return join( '', @text, "\\n" );
}

1;
```

The magic here is in three places. `report( )` looks at the call stack leading to its current position, extracting the name of the file and the line number of the calling code. It's possible to call this function directly with a message to display (and an optional level of calls to ignore).

`show_source( )` simply reads the named file and returns a string containing two lines before and after the numbered line, if possible. It also highlights the specific line with an

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

asterisk in the left column. Note the `local`ization and use of the `$.` magic variable to count the current line in the file.

`import( )` adds global handlers for warnings and exceptions, if requested from the calling module. The difference between the handlers is that when Perl issues a lexical warning, it doesn't affect the call stack in the same way that it does when it throws an exception.

Running the Hack

This short program shows all three ways of invoking `SourceCarp`:

```
#!/usr/bin/perl

use strict;
use warnings;

use lib 'lib';
use SourceCarp fatal => 1, warnings => 1;

# throw warning
open my $fh, '<', '/no/file';
print {$fh}...

# report from subroutine
report_with_level( );

sub report_with_level
{
    SourceCarp::report( "report caller, not self\\n", 2 );
}

# throw error
die "Oops!";
```

Hacking the Hack

There's no reason to limit your error and warning reporting to showing the file context around the calling line. `caller( )` offers much more information, including the variables passed to each function in certain circumstances.[2] It's possible to provide and present this information in a much more useful manner.

[2] See `perldoc -f caller`.

Overriding the global `__WARN__` and `__DIE__` handlers is serious business as it can interfere with large programs. A more robust implementation of this hack might work nicely with

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

`Carp`, not only because it is more widely compatible, but also because that module offers more features. Another possibility is to integrate this code somehow with `Log::Log4perl`.

Hack 56. Deparse Anonymous Functions

Inspect the code of anonymous subroutines.

Perl makes it really easy to generate anonymous subroutines on the fly. It's very handy when you need a bunch of oh-so similar behaviors which merely differ on small points. Unfortunately, slinging a bunch of anonymous subroutines around quickly becomes a headache when things go awry.

When an anonymous sub isn't doing what you expect, how do you know what it is? It's anonymous, fer cryin' out loud. Yet Perl knows what it is—and you can ask it.

The Hack

Suppose that you've written a simple `filter` subroutine which returns all of the lines from a file handle that match your filter criteria.

```
sub filter
{
    my ($filter) = @_;

    if ('Regexp' eq ref $filter)
    {
        return sub
        {
            my $fh = shift;
            return grep { /$filter/ } <$fh>;
        };
    }
    else
    {
        return sub
        {
            my $fh = shift;
            return grep { 0 <= index $_, $filter } <$fh>;
        };
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
        }
    }
```

Using the subroutine is simple. Pass it a precompiled regex and it will return lines which match the regular expression. Pass it a string and it will return lines which contain that string as a substring.

Unfortunately, later on you wonder why the following code returns every line from the file handle instead of just the lines which contain a digit:

```
my $filter = filter(/\\d/);
my @lines  = $filter->($file_handle);
```

`Data::Dumper` is of no use here:

```
use Data::Dumper;
print Dumper( $filter );
```

This results in:

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

Running the Hack

Using the `Data::Dump::Streamer` serialization module allows you to see inside that subroutine:

```
use Data::Dump::Streamer;
Dump( $filter );
```

Now you can see the body of the subroutine more or less as Perl sees it.

```
my ($filter);
$filter = undef;
$CODE1 = sub {
           my $fh = shift @_;
           return grep({0 <= index($_, $filter);} <$fh>);
         };
```

From there, it's pretty apparent that Perl didn't recognize that you were trying to pass in a regular expression and the bug is trivial to fix:

```
my $filter = filter(qr/\\d/);
my @lines  = $filter->($file_handle);
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Hacking the Hack

Behind the scenes, `Data::Dump::Streamer` uses the core module `B::Deparse`. In essence it does the following:

```
use B::Deparse;
my $deparse = B::Deparse->new( );
print $deparse->coderef2text($filter);
```

which outputs:

```
{
    my $fh = shift @_;
    return grep({0 <= index($_, $filter);} <$fh>);
}
```

The primary difference is that `Data::Dump::Streamer` also shows the values of any variables that the subroutine has closed over. See "Peek Inside Closures" [Hack #76] for more details. This technique is also good for displaying diagnostics when you `eval` code into existence or receive a subroutine reference as an argument and something goes wrong when you try to execute it.

The `B::Deparse` documentation gives more information about the arguments that you can pass to its constructor for even better control over the output.

Hack 57. Name Your Anonymous Subroutines

Trade a little anonymity for expressivity.

Despite the apparently oxymoronic name, "named anonymous subroutines" are an undocumented feature of Perl. Originally described by "ysth" on Perl Monks, these are a wonderful feature.

Suppose your program merrily runs along with a carefree attitude—but then dies an ugly death:

```
Denominator must not be zero! at anon_subs.pl line 11
      main::__ANON__(0) called at anon_subs.pl line 17
```

What the heck is `main::__ANON__(0)`? The answer may be somewhere in code such as:

```
use Carp;

sub divide_by
{
    my $numerator = shift;
    return sub
    {
        my $denominator = shift;
        croak "Denominator must not be zero!" unless $denominator;
        return $numerator / $denominator;
    };
}

my $seven_divided_by = divide_by(7);
my $answer           = $seven_divided_by->(0);
```

In this toy example, it's easy to see the problem. However, what if you're generating a ton of those `divide_by` subroutines and sending them all throughout your code? What if you have a bunch of subroutines all generating subroutines (for example, if you've breathed too deeply the heady fumes of Mark Jason Dominus' *Higher Order Perl* book)? Having a bunch of subroutines named `__ANON__` is very difficult to debug.

`$seven_divided_by` is effectively a *curried* version of `divide_by ( )`. That is, it's a function that already has one of multiple arguments bound to it. There's a piece of random functional programming jargon to use to impress people.

The Hack

Creating an anonymous subroutine creates a glob named `*__ANON__` in the current package. When `caller( )` and the rest of Perl's guts look for names for anonymous subroutines, they look there. Using `carp` and `croak` will quickly reveal this.

The solution is therefore to override this name temporarily. The easy way is to have the parent subroutine name the anonymous one:

```
sub divide_by
{
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
    my $numerator = shift;
    my $name      = (caller(0))[3];
    return sub
    {
        local *__ANON__ = "__ANON__$name";
        my $denominator = shift;
        croak "Denominator must not be zero!" unless $denominator;
        return $numerator / $denominator;
    };
}
```

Running the program now produces the output:

```
Denominator must not be zero! at anon_subs.pl line 12
      __ANON__main::divide_by(0) called at anon_subs.pl line 18
```

Hacking the Hack

While that's better and it may fit your needs, it's not the most flexible solution. If you create several anonymous subroutines, they will all have the same name. It's more powerful to name the anonymous subroutines by passing the creator subroutine a name—or taking it from an argument, as appropriate.

```
use Carp;

sub divide_by
{
    my ($name, $numerator) = @_;
    return sub
    {
        local *__ANON__ = "__ANON__$name";
        my $denominator = shift;
        croak "Denominator must not be zero!" unless $denominator;
        return $numerator / $denominator;
    };
}

my $three_divided_by = divide_by( 'divide_by_three', 3 );
my $answer           = $three_divided_by->(0);
```

The output looks like you expect:

```
Denominator must not be zero! at anon_subs.pl line 12
      __ANON__main::divide_by_three(0) called at anon_subs.pl line 18
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

Note that this code as written does *not* work under the debugger. The solution is to disable a debugger flag *before* Perl compiles the anonymous subroutines:

```
my $old_p;
BEGIN { $old_p = $^P; $^P &= ~0x200; }

sub divide_by
{
     # ...
}

BEGIN { $^P = $old_p; }
```

See `perldoc perlvar` for an explanation of `$^P`.

Hack 58. Find a Subroutine's Source

Find out where subroutines come from.

There are few things more annoying than finding a misbehaving subroutine and not being able to figure out where it came from. Some modules export subroutines automatically. Sometimes someone will have imported absolutely everything by using the `:all` tag in the `use` line.

Whatever the cause, the first step in fixing an errant subroutine is locating it.

The Hack

You *could* muck around in your symbol table [Hack #72] and use introspection to find the CV and check its STASH information [Hack #78], but Rafael Garcia-Suarez's `Sub::Identify` does this for you (using the invaluable `B` backend module internally).

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

The B module is uncommon, but very handy when necessary. It effectively allows you to explore Perl's inner workings. In this example, svref_2object( ) takes a code reference and returns an object blessed into the B::CV class. You won't actually find this class declared anywhere, but it's part of the B module internally.

Running the Hack

Just use the stash_name( ) function:

```
package My::Package;

use Sub::Identify ':all';
use HTML::Entities 'encode_entities';
print stash_name( \\&encode_entities );
```

Run this code; it will print HTML::Entities. Even if another module has re-exported &encode_entities into your namespace, Sub::Identify will still report HTML::Entities as the source of the subroutine.

For descriptions of the class hierarchy of these objects and the methods that you can call on them, see OVERVIEW OF CLASSES and SV-RELATED CLASSES in perldoc B. Unfortunately, much of the documentation is rather sparse and reading the source code of this module and the header files of the various Perl data structures, as well as pestering P5P with questions, is often the best way to figure out what you're doing. See also Chapter 8.

Hack 59. Customize the Debugger

Write your own debugger commands.

Adding a command to the debugger (or modifying an existing one) by editing the debugger is a difficult job; to do this, you have to patch the debugger source in *perl5db.pl* and replace

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

it. Sometimes you don't have the necessary privileges to do this, and given the complexity of the debugger, it's a difficult job—especially because you can't debug the debugger.

Yet modifying your tools the way you want them is important. Fortunately, `Devel::Command` module makes this much simpler. With `Devel::Command`, you write simple modules to define your commands, and the debugger finds them and loads them for you automatically.

The Hack

Writing a command is simple. There are only a few things to remember:

Input and output
: The debugger reads input from `DB::IN` and writes to `DB::OUT`. If you want your command to work just like a native debugger command, you need to use these filehandles for input and output. Generally, you'll only need to print to `DB::OUT`.

Debugger context versus program context
: To evaluate an expression in the context of the program that's being debugged (for example, you want to pass the value of a variable in the program to your command), call the *subroutine* `&eval` on it. To evaluate something in the *debugger's* context, use plain old `eval`.

A "hello, world" command looks like:

```
package Devel::Command::HelloWorld;
use base 'Devel::Command';

sub command
{
    print DB::OUT "Hello world!\\n";
    1;
}

1;
```

`Devel::Command` defaults to using the `command( )` as the actual command code. Run this by putting it somewhere in your `@INC` and then start the debugger:

```
flatbox ~ $ perl -de0
Default die handler restored.
Patching with Devel::Command::DBSub::DB_5_8

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or \Qh h' for help, or \Qman perldebug' for more help.

main::(-e:1):    0
  DB<1> cmds
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
cmds
helloworld
  DB<2> helloworld
Hello world!
  DB<3> q
flatbox ~ $
```

The message that begins `Patching with...` lets you know that `Devel::Command` has successfully activated. `cmds` lists the commands and typing `helloworld` runs your command.

Overriding a debugger command

Overriding a command is simple: just return true if your command routine wants to handle the command or false if you don't.

```
package Devel::Command::X;

use base 'Devel::Command';

sub command
{
    my ($cmd) = @_;

    if ($cmd  =~ /x marks/)
    {
        print DB::OUT "Arrrrr....\\n";
        return 1;
    }
    else
    {
        return 0;
    }
}

1;
```

Now the `x` command knows to be piratical when it sees a command beginning with `x marks`.

```
flatbox ~ $ perl -de0
Default die handler restored.
Patching with Devel::Command::DBSub::DB_5_8

Loading DB routines from perl5db.pl version 1.07
Editor support available.

Enter h or \Qh h' for help, or \Qman perldebug' for more help.

main::(-e:1):    0
  DB<1> $x = [1,2,3]
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
  DB<2> x $x
0  ARRAY(0x804e2f4)
   0  1
   1  2
   2  3
  DB<3> x marks the spot
Arrrrr....
  DB<4> q
flatbox ~ $
```

Running the Hack

Create a module in the `Devel::Command::` namespace. Install `Devel::Command` from CPAN, and then tell the debugger to load it by adding one line to your debugger initialization file, *.perldb* (or *perldb.ini*, for non-Unix systems):

```
use Devel::Command;
```

That's it. This makes the debugger automatically search `@INC` for modules in the `Devel::Command::` namespace, load them, and install them as commands. By default, it picks a name for the command by downcasing the last namespace qualifier (so, for example, `Devel::Command::My::DoStuff` ends up as the `dostuff` command).

`Devel::Command` also installs its own `cmds` command, which lists all commands that it found and loaded, and dynamically patches the debugger's command processing subroutine with a modified version which knows how to find the commands installed by `Devel::Command`.

Hacking the Hack

To develop tests while using the debugger, try the `Devel::Command::Tdump` module on CPAN. This module loads `Test::More` for you and lets you actually write tests and save them from the debugger.

If you want to see drawings of your data structure in the debugger, `Devel::Command::Viz` and the `graphviz` package will let you do it. Install those, then use the `viz` command on a variable:

```
flatbox ~ $ perl5.8.5 -de0
Patching with Devel::Command::DBSub::DB_5_8_5

Loading DB routines from perl5db.pl version 1.27
Editor support available.
```


Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006

```
Enter h or \Qh h' for help, or \Qman perldebug' for more help.

main::(-e:1):    0
  DB<1> use WWW::Mechanize

  DB<2> $m = WWW::Mechanize->new( )

  DB<3> viz $m
```

You'll see a graphical depiction of the `WWW::Mechanize` object in a pop-up window.

Perl Hacks By , Damian Conway, Curtis "Ovid" Poe ISBN: 0596526741 Publisher: O'Reilly
Print Publication Date: 5/1/2006