Table of Contents

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

Chapter 8. Know Thy Code

Hacks 70-83

Introspection isn't just a self-help exercise. It's a way of asking Perl what it thinks about your program.

Why does that matter? There are plenty of advanced techniques that, properly applied, will save you much time, effort, and trouble. That word "properly" is the sticky one though—unless you know what's proper and what's not, you'll have difficulty mastering advanced Perl.

Despite all the rich nooks and crannies and hidden corners of the core, there are only a few techniques you absolutely must understand. Study well the hacks here and you'll absorb higher lore and unlock secrets that will help you customize Perl, the language, for your specific needs.

Hack 70. Understand What Happens When

Tell compile time from runtime.

Dynamic languages are flexible, neither requiring you to know all of the code you're ever going to run in a program at compile time nor necessarily failing if it's not there at runtime. Perl can live with some ambiguity about seeing functions you haven't defined yet (if ever) and referring to variables that don't necessarily have any values yet.

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

That doesn't always make life easier for programmers. While Perl's pretty good about knowing what happens when, reading the source code doesn't always make it clear. While it may seem obvious to you that program execution happens top to bottom that's not always how it works.

The Hack

Here's what actually happens.

Compilation

When you first run your program, Perl reads the file and starts compiling from top to bottom. At this point, it looks for symbols (variables and subroutines), registers them appropriately, and converts the text of the program into an internal representation that it can execute. If it encounters syntax errors, it aborts and reports an error message.

Of course, some constructs aren't syntax errors in normal use:

```
#!/usr/bin/perl

my $age = 10;
print $aeg;
```

Perl will only complain about an undeclared variable `$aeg` when running under the `strict` pragma. However, the question of how this works is less obvious when you consider that Perl reports this error *before* running the code. Consider:

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

my $age = 10;
print $aeg;
```

The secret is that `use` internally becomes:

```
BEGIN
{
    require 'strict';
    strict->import( ) if strict->can( 'import' );
}
```

Perl then loads the `strict` module, if it can, and starts compiling that, returning to the main program when it finishes.

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

Whenever Perl encounters a BEGIN block, it executes its contents immediately, just as it encounters them. Of course, it doesn't execute code outside of the block that a programmer might think is important to the block:

```
my $name = 'Spot';

BEGIN { print "Hello, $name!\\n" }
```

Though the BEGIN block executes as soon as Perl encounters it, and though Perl has already associated $name with the appropriate storage spot inside and outside of the block, the assignment will not happen until runtime; the BEGIN block executes *before* the assignment happens, even if it comes later in the file.

Even though it may seem correct that this would work if it were part of a module loaded from the main program (at least with use), the internal BEGIN will still execute before the rest of the code in the file.

Initialization

As soon as Perl finishes compiling, it runs any CHECK blocks found—but in reverse order of their declaration. For example:

```
#!/usr/bin/perl

BEGIN { print "First!\\n"  }
CHECK { print "Third!\\n"  }
CHECK { print "Second!\\n" }
```

prints:

```
First!
Second!
Third!
```

INIT blocks run *after* all CHECK blocks in order of their appearance:

```
#!/usr/bin/perl

BEGIN { print "First!\\n"  }
INIT  { print "Fourth!\\n" }
CHECK { print "Third!\\n"  }
CHECK { print "Second!\\n" }
INIT  { print "Fifth!\\n"  }
```

prints:

```
First!
Second!
```


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

```
Third!
Fourth!
            Fifth!
```

Runtime

When running, execution order happens as you might expect. There aren't any surprises unless you do something tricky (as per most of the rest of the book). One change is that running code by `eval`ing a string—after runtime starts—will only execute any `BEGIN` blocks found as a result of that operation, not `CHECK` or `INIT` blocks.[1] This program:

[1] This is why `Attribute::Handlers` and persistent interpreters such as mod_perl do not get along by default.

```
#!/usr/bin/perl

  BEGIN { print "First!\\n"  }
  INIT  { print "Fourth!\\n" }
  CHECK { print "Third!\\n"  }
  CHECK { print "Second!\\n" }
  INIT  { print "Fifth!\\n"  }

  eval <<END_EVAL;
             BEGIN { print "BEGIN in eval\\n!" }
             CHECK { print "CHECK in eval\\n!" }
             INIT  { print "INIT in eval\\n!"  }
             END_EVAL
```

prints:

```
First!
Second!
Third!
Fourth!
Fifth!
BEGIN in eval!
```

Cleanup

Finally, when it comes time for the program to `exit` (but not with a compilation error), Perl runs all `END` blocks in reverse order of their appearance:

```
#!/usr/bin/perl

  BEGIN { print "First!\\n"  }
  INIT  { print "Fourth!\\n" }
  CHECK { print "Third!\\n"  }
  CHECK { print "Second!\\n" }
  INIT  { print "Fifth!\\n"  }

  eval <<END_EVAL;
  BEGIN { print "BEGIN in eval\\n!" }
  CHECK { print "CHECK in eval\\n!" }
  INIT  { print "INIT in eval\\n!"  }
```


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

```
END   { print "Sixth!\\n"          }
            END_EVAL
            END   { print "Seventh!\\n"       }
            END   { print "Eighth!\\n"        }
```

prints:

```
First!
Second!
Third!
Fourth!
Fifth!
BEGIN in eval!
Sixth!
            Seventh!
            Eighth!
```

Why did the `END` block in the `eval` execute first? Although it's an `END` block and Perl encountered the string first, it executes that block at runtime, so it's the final `END` block compiled and, thus, the first to execute.

See `perlmod` for more details.

Hack 71. Inspect Your Data Structures

Peek into a reference and see how far down it goes.

How do you know the structure of a Perl reference? Is the reference to a hash, an array, an object, a scalar, or something else? Many people suggest the use of `Data::Dumper`. This module has a method that dumps the data structure as a text string. It works very well, but its main purpose is to serialize a data structure into a string you can `eval` to recreate the reference and its data.

Most of the time I don't want to save state or `eval` anything. I just want to see a text representation of the reference. I really like the representation of a data structure that using the `x` command within the Perl debugger provides.

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

Dumping References Outside the Debugger

Is it possible to produce this from a Perl program without using the debugger? Yes!

```
use strict;
use Dumpvalue;

my $d    = Dumpvalue->new( );
my $hash =
{
     first_name => 'Tim',
     last_name  => 'Allwine',
     friends    => [ 'Jon','Nat','Joe' ],
};
$d->dumpValue(\\$hash);
```

This produces the output:

```
-> HASH(0x80a190)
    'first_name' => 'Tim'
    'friends' => ARRAY(0x800368)
       0  'Jon'
       1  'Nat'
       2  'Joe'
    'last_name' => 'Allwine'
```

This is the same output that the debugger produces. The `HASH` line says that `$hash` is a hash reference. The next level of indentation shows the keys of the hash and their corresponding values. `first_name` for example points to the string `Tim` but `friends` points to an array reference. The contents of that array appear, one at a time, indented one step further with their indices within the array and their values.

This technique is handy when you have to maintain code written by other people. Suppose that you're editing a web program with over 2,000 lines of code. Deep in the code, you find a reference named `$someref` and you want to see its contents. At the top of the file, add the lines:

```
use Dumpvalue;
my $d = Dumpvalue->new( );
```

Then while `$d` and `$someref` are in scope, add the line:

```
$d->dumpValue(\\$someref);
```

When you run it, the code will dump `$someref`.

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

Printing to a File

One complaint with this technique is where `dumpValue( )` prints. It usually prints to STDOUT, by default, but actually it prints to the currently selected output filehandle. That's a hint. Add a couple of lines to change the filehandle:

```
open my $fh, '>dump.out';
my $old_fh = select($fh);
$d->dumpValue(\\$ref);
close $fh;
select($old_fh);
```

Now when you run the program, the dump string will end up in a file called *dump.out*.

Output in CGI or mod_perl Programs

Dumping the output in CGI or mod_perl programs is more complex. Often you don't want to print to a filehandle at all, as it may change the rendering of the output drastically. Instead, use `IO::Scalar` to create a filehandle to a string and select that filehandle. Then, undirected `print` or `dumpValue( )` calls will go to this new filehandle. Select the old filehandle and carry on with your program, printing `$dump_str` when you want.

```
use IO::Scalar;

my $dump_str;
my $io  = IO::Scalar->new(\\$dump_str);
my $oio = select($io);

print '<pre>',"\\n";       # goes to $dump_str
$d->dumpvalue(\\$someref); # as does this
print '</pre>';           # and this too

select($oio);             # old filehandle
print $dump_str;          # stdout again when you want it to
```

Hack 72. Find Functions Safely

Look for code to execute without risking explosions.

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

The ultimate goal of designing reusable code is genericity—being able to write useful pieces of code that allow future expansion without (much) difficulty or modification. Complete genericity is difficult, as your code has to make *some* assumptions *somewhere*.

Perl's very flexible about how you interact with other code. You can fold, spindle, mutilate, and mangle symbols in any package you want at almost any time. Although this flexibility makes it possible to find code in other packages, sometimes it makes it difficult to know if the function you want is really there, at least safely and without digging through symbol tables.

You can do it, though.

The Hack

If you can avoid the problem, avoid it.

One of the most common ways to interact with other code is to provide an interface you expect it to fulfill. This may be through suggesting that all plug-ins inherit from a base class that provides default methods to overload or through documenting that your code will always call plug-in methods and pass specified arguments.

Subclassing can be fragile, though, especially in Perl where your implementation choices affect everyone else who writes code. (See the implementation of `HTTP::Daemon` and how it stores instance data, for example.)

If you only need to know that plug-ins or extensions conform to an interface, consider using a Perl 6-ish module such as `Class::Roles` or `Class::Trait`. Though there's a little bit of theory to learn before you understand the code, you can make your code more flexible and generic without enforcing more on the extensions than you really need to enforce.

Get cozy with can( )

If you *can't* entirely force a separate interface, as in the case where you want to make some methods publicly callable from user requests on a web site and other methods private to the world, consider namespacing them. For example, imagine a web program that performs mathematical operations based on the contents of the `action` parameter:

```
sub dispatch_request
{
    my ($self, $q) = @_;
```


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

```
    my $action     = $q->param( 'action' );
    $self->$action( );
}
```

This technique isn't *quite* as bad as invoking `$action` directly as a symbolic reference, but it provides little safety. An attacker could provide an invalid action, at best crashing the program as Perl tries to invoke an unknown method, or provide the name of a private method somewhere that he really shouldn't call, revealing sensitive data or causing unexpected havoc.

To verify that the method exists somewhere, use the `can( )` method (provided by the `UNIVERSAL` ancestor of all classes):

```
sub dispatch_request
{
    my ($self, $q) = @_;
    my $action     = $q->param( 'action' );
    return unless $self->can( $action );
    $self->$action( );
}
```

That prevents attackers from calling undefined methods, but it's little advantage over wrapping the whole dispatch in an `eval` block. If you change the names of all valid methods to start (or end) with a known token, you can prevent calling private methods:

```
sub dispatch_request
{
    my ($self, $q) = @_;
    my $action     = 'action_' . $q->param( 'action' );
    return unless $self->can( $action );
    $self->$action( );
}
```

Now when the user selects the `login` action, the request dispatches to the method `action_login`. For even further protection, see "Control Access to Remote Objects" [Hack #48].

Find functions, not methods!

That works for methods, but what about functions? Perl 5 at least makes very few internal distinctions between methods and subroutines. `can( )` works just as well on package names to find subroutines as it does class names to find methods. If you know you've loaded a plug-in called `Logger` and want to know if it can `register( )`, try:

```
my $register_subref = Logger->can( 'register ');
$register_subref->( ) if $register_subref;
```

`can( )` returns a reference to the found function if it exists and `undef` otherwise.

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

This also works if you have the package name in a variable:

```
my $register_subref = $plugin->can( 'register ');
$register_subref->( ) if $register_subref;
```

If you don't know for sure that `$plugin` contains a valid package name, wrap the `can( )` method call in an `eval` block:[2]

[2] Some people recommend calling `UNIVERSAL::can( )` as a function, not a method. That's silly; what if the package overrides `can( )`? You'll get the wrong answer!

```
my $register_subref = eval { $plugin->can( 'register ') };
$register_subref->( ) if $register_subref;
```

If the `eval` fails, `$register_subref` will be false.

Hack 73. Know What's Core and When

Keep track of the core modules you're using and not using.

Not every Perl installation is fortunate enough to be able to install the latest released version of the core as soon as it is available or to install the freshest modules off of the CPAN as soon as they hit the index. Some developers on legacy systems have to be very careful to avoid the wrath of their system administrators for whom stability is a way of life, not just a goal.

Though Perl 5's standard library has always provided a lot of features, it has grown over time. What's standard and usable everywhere as of Perl 5.8.7 isn't the same as what existed as of Perl 5.004. How can you know, though, without either digging through release notes or watching your carefully constructed code break when put on the testing machine?

Use `Module::CoreList`.

The Hack

Suppose you've read `perldoc perlport` and have resolved never to write unportable, hardcoded file paths anymore. You've browsed the documentation of `File::Spec` and realize that a few careful calls can make your code more likely to work on platforms as exotic as Windows (or even VMS, but that's scary).

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

Unfortunately for your good intentions, your development platform is a box that shipped with Perl 5.004 from way back when the network really was the computer. Before replacing all of your `join( '/', @path, $filename )` code with calls to shiny `catfile( )`, your sense of duty and due diligence causes you to ask "Wait, when did `File::Spec` enter the core?"

Install `Module::CoreList` from the CPAN, and then bring up a command line:

```
$ perl -MModule::CoreList -e 'print Module::CoreList->first_release(
            "File::Spec" ), "\\n"'
5.005
```

Good thing you checked. Now you have three choices: submit the paperwork to upgrade Perl on that machine to something released this millennium,[3] bribe the sysadmin to install `File::Spec` on that machine, or sadly give up on the idea that this code will work unmodified on Mac OS classic.

[3] Perl 5.004_05 released in April 1999.

Checking by version

Maybe knowing the first occurrence of the module isn't good enough. Consider the case of poor `Test::Simple`. Though the first versions were useful and good, it wasn't until release 0.30 and the introduction of `Test::Builder` that the golden age of Perl testing began. If you haven't upgraded Perl in a couple of years and rely on the universal Perl testing backend, what's the minimum version of Perl you can use without having to install a newer `Test::Simple`?

Pass an optional second value to `first_release( )`, the version number of the package:

```
$ perl -MModule::CoreList -e 'print Module::CoreList->first_release(
            "Test::Simple", '0.30' ), "\\n"'
5.007003
```

Anything released after Perl 5.7.3 contains `Test::Builder`. Of course, note that this doesn't mean that releases of Perl with *lower* numbers don't contain `Test::Builder`—Perl 5.6.2, released *after* Perl 5.7.3, contains `Test::Simple` 0.47.

If you're really curious, and especially if you end up with information about development releases of Perl, browse through the data structures at the end of Module::CoreList by hand [Hack #2] for more detailed information.

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

When did `Module::CoreList` make it in the core? Perl 5.9.2.

Hack 74. Trace All Used Modules

See what modules your program uses—and what modules those modules use!

Perhaps the most useful feature of Perl 5 is module support, allowing the use of existing, pre-written code. With thousands of modules on the CPAN available for free, it's likely that any code you write will use at least a few other pieces of code.

Of course, all of the modules you use optionally use a few modules of their own, and so on. You could find yourself loading dozens of pieces of code for what looks like a simple program. Alternately, you may just be curious to see the relationships within your code.

Wouldn't it be nice to see which modules your code loaded from where? Now you can.

The Hack

The easiest way to gather the information on what Perl modules any piece of code loads is a little-known feature of `@INC`, the magic variable that governs where Perl looks to load modules. If `@INC` contains a code reference, it will execute that reference when attempting to load a module. This is a great place to store code to manage library paths, as `PAR` and `The::Net` (http://www.perlmonks.org/?node_id=92473, not on the CPAN) do. It also works well to collect interesting statistics:

```
package Devel::TraceUse;

use Time::HiRes qw( gettimeofday tv_interval );

BEGIN
{
    unshift @INC, \\&trace_use unless grep { "$_" eq \\&trace_use . '' } @INC;
}

sub trace_use
{
    my ($code, $module) = @_;
    (my $mod_name       = $module) =~ s{/}{::}g;
    $mod_name           =~ s/\\.pm$//;
    my ($package, $filename, $line) = caller( );
    my $elapsed         = 0;
```


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

```
    {
        local *INC      = [ @INC[1..$#INC] ];
        my $start_time  = [ gettimeofday( ) ];
        eval "package $package; require '$mod_name';";
        $elapsed        = tv_interval( $start_time );
    }
    $package            = $filename if $package eq 'main';
    push @used,
    {
        'file'   => $package,
        'line'   => $line,
        'time'   => $elapsed,
        'module' => $mod_name,
    };

    return;
}

END
{
    my $first = $used[0];
    my %seen  = ( $first->{file} => 1 );
    my $pos   = 1;

    warn "Modules used from $first->{file}:\\n";

    for my $mod (@used)
    {
        my $message = '';

        if (exists $seen{$mod->{file}})
        {
            $pos = $seen{$mod->{file}};
        }
        else
        {
            $seen{$mod->{file}} = ++$pos;
        }

        my $indent = '  ' x $pos;
        $message  .= "$indent$mod->{module}, line $mod->{line}";
        $message  .= " ($mod->{time})" if $mod->{time};
        warn "$message\\n";
    }
}

1;
```

The code begins by storing a reference to `trace_use( )` at the head of `@ISA`. Whenever Perl encounters a `use` or `require` statement for a module it hasn't previously loaded, it will loop through each entry in `@ISA`, trying to load the module from there. As the first entry is a subroutine reference, Perl will call the subroutine with the name of the module to load (at least, translated into a Unix-style file path).

`Devel::TraceUse` translates the path name back into a module name, looks up the call stack to find the name of the package and file containing the `use` or `require` statement as well as the line number of the statement, and then redispatches the lookup, taking itself temporarily out of `@INC`. This redispatch allows the module to collect information on how long it took to load the module.

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

This time isn't absolute; the string `eval` statement as well as the calls to `Time::HiRes` take up a near-constant amount of time. However, it's likely consistent, so comparing times to each other is sensible.

The code uses the filename of the caller if there's no explicit package given, stores all of the available information, and pushes that structure into an array of modules used.

At the end of the program, the module prints a report of the modules loaded in the order in which Perl encountered them.

Running the Hack

Perhaps the `prove` utility from `Test::Harness` has captured your attention and you want to know what modules it loads. With `Devel::TraceUse` in your path somewhere, run the command:

```
$ perl -MDevel::TraceUse /usr/bin/prove
Modules used from /usr/bin/prove:
  Test::Harness, line 8 (0.000544)
    Test::Harness::Straps, line 6 (0.000442)
      Test::Harness::Assert, line 9 (0.000464)
      Test::Harness::Iterator, line 10 (0.000581)
      Test::Harness::Point, line 11 (0.000437)
      POSIX, line 313 (0.000483)
        XSLoader, line 9 (0.000425)
    Benchmark, line 9 (0.000497)
      Exporter::Heavy, line 17 (0.000502)
  Getopt::Long, line 9 (0.000495)
    constant, line 221 (0.000475)
  Pod::Usage, line 10 (0.000486)
    File::Spec, line 405 (0.000464)
      File::Spec::Unix, line 21 (0.000432)
    Pod::Text, line 411 (0.000471)
      Pod::ParseLink, line 30 (0.000475)
      Pod::Select, line 31 (0.000447)
        Pod::Parser, line 242 (0.000461)
          Pod::InputObjects, line 205 (0.000444)
          Symbol, line 210 (0.000469)
  File::Glob, line 82 (0.000521)
```

Thus `prove` uses `Test::Harness`, `Getopt::Long`, `Pod::Usage`, and `File::Glob` directly, each of which uses several other modules. If you were adding features to `prove` and wanted to know if using `POSIX` would add significantly to the resource footprint, you would now know that you already pay the price for it, so you might as well use it.

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

Hacking the Hack

What could make this module more useful? Right now, it doesn't report the use of `Time::HiRes`, because it uses that internally. Making timing information optional would be nice. Furthermore, the report always goes to STDERR, which may mingle badly with other program output.

Perhaps you want to filter out certain packages selectively, or trace *all* of the uses of `require` and `use`. In lieu of reloading every module every time some piece of code wants to use it, Perl tracks loaded modules by caching their filenames in `%INC`. To have `Devel::TraceUse` track every attempt to load a module, whether Perl has loaded it, keep the delegation, but clear out `%INC`. (Be sure to keep your own cache, though, to prevent subroutine redefinitions and initialization code from running over and over again.)

Hack 75. Find All Symbols in a Package

Explore symbol tables without soft references.

One of the earliest temptations for novice programmers is to use the contents of one variable as part of the name of another variable. After making one too many costly mistakes or showing such code to a more experienced programmer, novices start to use the `strict` pragma to warn them about dubious constructs.

However, several advanced features of Perl, such as the implementation of the `Exporter` module, are only possible by reading from and writing to the symbol table at run time. Normally `strict` forbids this—but it's possible to access global symbols at run time with `strict` enabled.

This is an easy way to find out if a symbol—such as a scalar, array, hash, subroutine, or filehandle—exists.

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

The Hack

Suppose you want to check whether a specific type of variable is present in a given namespace. You need to know the name of the package, the name of the variable, and the type of the variable.

Defining the following subroutine in the `UNIVERSAL` package makes the class method `contains_symbol` available to any package:[4]

[4] Er...class.

```
my %types =
(
    '$' => 'SCALAR',
    '@' => 'ARRAY',
    '%' => 'HASH',
    '*' => 'IO',
    '&' => 'CODE',
);

sub UNIVERSAL::contains_symbol
{
    my ($namespace, $symbol) = @_;
    my @keys                 = split( /::/, $namespace );
    my $type                 = $types{ substr( $symbol, 0, 1, '' ) }
                            || 'SCALAR';

    my $table = \\%main::;

    for my $key (@keys)
    {
        $key .= '::';
        return 0 unless exists $table->{$key};
        $table = $table->{$key};
    }

    return 0 unless exists $table->{$symbol};
    return *{ $table->{$symbol} }{ $type } ? 1 : 0;
}
```

To see if a symbol exists, for example to test that `contains_symbol` exists in the `UNIVERSAL` package, call the method like:

```
print "Found it!\\n" if UNIVERSAL->contains_symbol( '&contains_symbol' );
```

How does it work?

Perl uses the same data structure for hashes as it does for symbol tables. The same operations —storing and retrieving values by key; iterating over keys, values, or both; and checking the existence of a key—work on both. The secret is knowing how to access the symbol table.

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

The main symbol table is always available as the hash named `%main::`. Every other symbol table has an entry starting there. For example, `strict`'s symbols are available in `$main::{'strict::'}`, while `CGI::Application`'s symbols are in `$main::{'CGI::'}{'Application::'}`. Each level is a new hash reference.

The quotes are important to identify the name with the colons appropriately.

Within a symbol table, all leaf entries (values that aren't themselves symbol tables) contain typeglobs. A typeglob is similar to a hash, but it cannot contain arbitrary keys—it has a fixed set of keys, as shown in the `%types` array.[5]

[5] There are other keys, but they're less common and rarely worth mentioning.

Because a typeglob *isn't* a hash, you can't access its members as you would a hash. Instead, you must dereference it with the leading `*` glob identifier, then subscript it with the name of the slot to check.

Running the Hack

Once you have the glob, you can assign references to it to fill in its slots. For example, to create a new subroutine `growl( )` in `Games::ScaryHouse::Monster`, find the symbol table for `Games::ScaryHouse::Monster` and the glob named `growl`. Then assign a reference to a subroutine to the glob and you will be able to call it as `Games::ScaryHouse::Monster::growl( )`. Similar techniques work for anything else to which you can take a reference.

This trick offers *some* benefits over other ways of querying for a symbol, such as using soft references, calling `can( )` on subroutines (which may run afoul of inheritance), and wrapping potentially harmful accesses in `eval` blocks. However do note that an optimization [6] automatically created a scalar entry in every new glob. Thus if you have a package global hash named `%compatriots`, `contains_symbol( )` will claim that `$compatriots` also exists.

[6] Removed as of Perl 5.9.3.

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

This technique does *not* work on lexical variables; they don't live in symbol tables!

Hack 76. Peek Inside Closures

Violate closure-based encapsulation when you really need to.

Very few rules in Perl are inviolate—not even the rule that lexicals are inaccessible outside their scopes. For closures to work (and even lexicals in general), Perl has to be able to access them *somehow*. If you could use the same mechanism, you could read from and write to these variables.

This is very useful for debugging closures and closure-based objects [Hack #43]. It's scary and wrong, but sometimes it's just what you need.

The Hack

Robin Houston's `PadWalker` module helpfully encapsulates the necessary dark magic in a single place that, most importantly, you don't have to understand to use. Suppose you have a misbehaving counter closure:[7]

[7] The erroneous operator is ==. There are actually *two* bugs, though.

```
sub make_counter
{
    my ($start, $end, $step) = @_;

    return sub
    {
        return if $start == $end;
        $start += $step;
    };
}
```


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

One way to debug this is to throw test case after test case at it [Hack #53] until it fails and you can deduce and reproduce why. An easier approach is to show all of the enclosed values when you have a misbehaving counter.

Once you have a counter, use `PadWalker`'s `closed_over( )` function to retrieve a hash of all closed-over variables, keyed on the name of the variable:

```
use Data::Dumper;
use PadWalker 'closed_over';

my $hundred_by_nines = make_counter( 0, 100, 9 );

while ( my $item = $hundred_by_nines->( ) )
{
    my $vars = closed_over( $hundred_by_nines );
    warn Dumper( $vars );
}
```

Running the Hack

Running this reveals that `$start`, the current value of the counter, quickly exceeds 100.

```
$VAR1 = {
          '$start' => \\9,
          '$step' => \\9,
          '$end' => \\100
        };
$VAR1 = {
          '$start' => \\18,
          '$step' => \\9,
          '$end' => \\100
        };

# ...

$VAR1 = {
          '$start' => \\6966,
          '$step' => \\9,
          '$end' => \\100
        };

# ...
```

`$step` and `$end` are okay, but because `$start` never actually *equals* `$end`, the closure never returns its end marker.

Changing the misbehaving operator to >= fixes this.[8]

[8] Consider if the `$step` is negative, however.

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

Hacking the Hack

One good turn of scary encapsulation-violation deserves another. The hash that `closed_over( )` returns actually contains *references* to the closed-over variables as its values. If you dereference them, you can assign to them. Here's one way to debug the idea that the comparision operator is incorrect:

```
while ( my $item = $hundred_by_nines->( ) )
{
    my $vars  = closed_over( $hundred_by_nines );
    my $start = $vars->{'$start'};
    my $end   = $vars->{'$start'};
    my $step  = $vars->{'$step'};

    if ( $$start > $$step )
    {
        $$start = $$end - $$step;
    }
}
```

`PadWalker` is good for accessing all sorts of lexicals. If you have a subroutine reference of any kind, you can see the names of the lexicals within that subroutine—not just any lexicals it closes over. You can't always get the values, though. They're only active if you're in something that that subroutine actually called somewhere.

Be careful, though; just because you can look in someone's closet doesn't mean that you should.

The CPAN module `Data::Dump::Streamer` can do similar magic, except that it also deparses the closure. This is useful in other circumstances. The code:

```
use Data::Dump::Streamer;
my $hundred_by_nines = make_counter( 0, 100, 9 );
1 while 100 > $hundred_by_nines->( );
Dump( $hundred_by_nines );
```

produces the result:

```
my ($end,$start,$step);
$end = 100;
$start = 108;
$step = 9;
$CODE1 = sub {
           return if $start = = $end;
           $start += $step;

         };
```


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

Hack 77. Find All Global Variables

Track down global variables so you can replace them.

Perl 5's roots in Perl 1 show through sometimes. This is especially evident in the fact that variables are global by default and lexical only by declaration. The `strict` pragma helps, but adding that to a large program that's only grown over time (in the sense that kudzu grows) can make programs difficult to manage.

One problem of refactoring such a program is that it's difficult to tell by reading whether a particular variable is global or lexical, especially when any declaration may have come hundreds or thousands of lines earlier. Your friends and co-workers may claim that you can't run a program to analyze your program and find these global variables, but *you can*!

The Hack

Perl 5 has several core modules in the `B::*` namespace referred to as the backend compiler collection. These modules let you work with the internal form of a program as Perl has compiled and is running it. To see a representation of a program as Perl sees it, use the `B::Concise` module. Here's a short program that uses both lexical and global variables:

```
use vars qw( $frog $toad );

sub wear_bunny_costume
{
    my $bunny = shift;
    $frog     = $bunny;
    print "\\$bunny is $bunny\\n\\$frog is $frog\\n\\$toad is $toad";
}
```

`$frog` and `$toad` are global variables.[9] `$bunny` is a lexical variable. Unless you notice the `my` or `use vars` lines, it's not obvious to the reader which is which. Perl knows, though:

[9] They're also friends.

```
$ perl -MO=Concise,wear_bunny_costume friendly_animals.pl
examples/friendly_animals.pl syntax OK
main::wear_bunny_costume:
n  <1> leavesub[1 ref] K/REFC,1 ->(end)
-     <@> lineseq KP ->n
```


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

```
1        <;> nextstate(main 35 friendly_animals.pl:5) v ->2
6        <2> sassign vKS/2 ->7
4           <1> shift sK/1 ->5
3              <1> rv2av[t2] sKRM/1 ->4
2                 <$> gv(*_) s ->3
5           <0> padsv[$bunny:35,36] sRM*/LVINTRO -6
7        <;> nextstate(main 36 friendly_animals.pl:6) v ->8
a        <2> sassign vKS/2 ->b
8           <0> padsv[$bunny:35,36] s ->9
-           <1> ex-rv2sv sKRM*/1 ->a
9              <$> gvsv(*frog) s -a
b        <;> nextstate(main 36 friendly_animals.pl:7) v ->c
m        <@> print sK ->n
c           <0> pushmark s ->d
-           <1> ex-stringify sK/1 ->m
-              <0> ex-pushmark s ->d
l              <2> concat[t6] sKS/2 ->m
j                 <2> concat[t5] sKS/2 ->k
h                    <2> concat[t4] sKS/2 ->i
f                       <2> concat[t3] sK/2 ->g
d                          <$> const(PV "$bunny is ") s ->e
e                          <0> padsv[$bunny:35,36] s -f
g                       <$> const(PV "\\n$frog is ") s ->h
-                    <1> ex-rv2sv sK/1 ->j
i                       <$> gvsv(*frog) s -j
k                 <$> const(PV "\\n") s ->l
```

That's a lot of potentially confusing output, but it's reasonably straightforward. This is a textual representation of the optree representing the `wear_bunny_costume( )` subroutine. The emboldened lines represent variable accesses. As you can see, there are two different opcodes used to fetch values from a variable. `padsv` fetches the value of a named lexical from a lexical pad, while `gvsv` fetches the value of a scalar from a typeglob.

Running the Hack

Knowing this, you can search for all `gvsv` ops within a compiled program and find the global variables! `B::XPath` is a backend module that allows you to search a given tree with XPath expressions. To look for a `gvsv` node in the optree, use the XPath expression `//gvsv`:

```
use B::XPath;

my $node = B::XPath->fetch_root( \\&wear_bunny_costume );

for my $global ( $node->match( '//gvsv' ) )
{
    my $location = $global->find_nextstate( );
    printf( "Global %s found at %s:%d\\n",
    $global->NAME( ), $location->file( ), $location->line( ) );
}
```

`fetch_root( )` gets the root opcode for a given subroutine. To search the entire program, use `B::XPath::fetch_main_root( ).match( )` applies an XPath expression to the optree starting at the given `$node`, returning a list of matching nodes.

As each node returned should be a `gvsv` op (blessed into `B::XPath::SVOP`), the `NAME ( )` method retrieves the name of the glob. The `find_nextstate( )` method finds the

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

nearest parent control op (or COP) which contains the name of the file and the line number on which the variable appeared.[10] The results are:

[10] It uses a heuristic, so it may not always be *exact*.

```
$ perl friendly_animals.pl
Global frog found at friendly_animals.pl:8
Global frog found at friendly_animals.pl:9
```

Hacking the Hack

If you want to find only globals named `$toad`, change the XPath expression and parameterize it by a node attribute:

```
$node->match( '//gvsv[@NAME="toad"]' ))
```

There's no limit to the types of opcodes you can search for in a program beyond what `B::XPath` supports and the XPath expressions you can write. As long as you can dump a snippet of code into an optree list, you can eventually turn that into an XPath expression. From there, just grab the node information you need and you're on your way.

See also the built-in `B::Xref` module. It produces a cross reference of variables and subroutines in your code.

Hack 78. Introspect Your Subroutines

Trace any subroutine to its source.

You can name anonymous subroutines [Hack #57] and deparse them [Hack #56]. You can even peek at their closed-over lexical variables [Hack #76]. There are still more wonders in the world.

Someday you'll have to debug a running program and figure out exactly where package *A* picked up subroutine *B*. One option is to trace all `import( )` calls, but that's even less fun than it sounds. Another option is to pull out the scariest and most powerful toolkit in the Perl hacker's toolbox: the `B::*` modules.

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

The Hack

Finding a misbehaving function means you need to know two of three things:

- The original package of the function
- The name of the file containing the function
- The line number in the file corresponding to the function

From there, your debugging should be somewhat easier. Perl stores all of this information for every CV[11] it compiles. You just need a way to get to it.

[11] The internal representation of all subroutines and methods.

The usual entry point is through the `B` module and its `svref_2object( )` function, which takes a normal Perl data structure, grabs the underlying C representation, and wraps it in hairy-scary objects that allow you to peek (though not usually poke) at its guts.

It's surprisingly easy to report a subroutine's vital information:

```
use B;

sub introspect_sub
{
    my $sub      = shift;
    my $cv       = B::svref_2object( $sub );

    return join( ':',
        $cv->STASH->NAME( ), $cv->FILE( ), $cv->GV->LINE( ) . "\\n"
    );
}
```

`introspect_sub( )` takes one argument, a reference to a subroutine. After passing it to `svref_2object( )`, it receives back a `B::CV` object. The `STASH( )` method returns the typeglob representing the package's namespace—calling `NAME( )` on this returns the package name. The `FILE( )` method returns the name of the file containing this subroutine. The `GV( )` method returns the particular symbol table entry for this subroutine, in which the `LINE( )` method returns the line of the file corresponding to the start of this subroutine.

Okay, using `Devel::Peek::CvGV` on a subroutine reference is easier.

```
use Devel::Peek 'CvGV';
sub Foo::bar { }
print CvGV( \\&Foo::bar );
```


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

Of course, that prints the *name* of the glob containing the subroutine...but it's a quick way to find even that much information. Now you know *two* ways to do it!

Running the Hack

Pass in any subroutine reference and print the result somehow to see all of this wonderful data:

```
use Data::Dumper;

package Foo;

sub foo { }

package Bar;

sub bar { }
*foo = \\&Foo::foo;

package main;

warn introspect_sub( \\&Foo::foo );
warn introspect_sub( \\&Bar::bar );
warn introspect_sub( \\&Bar::foo );
warn introspect_sub( \\&Dumper );

# introspect_sub( ) as before...
```

Run the file as normal:

```
$ perl introspect.pl
Foo:examples/introspect.pl:14
Bar:examples/introspect.pl:18
Foo:examples/introspect.pl:14
Data::Dumper:/usr/lib/perl5/site_perl/5.8.7/powerpc-linux/Data/Dumper.pm:495
```

As you can see, aliasing `Bar::foo( )` to `Foo::foo( )` didn't fool the introspector, nor did importing `Dumper( )` from `Data::Dumper`.

Hacking the Hack

That's not all though. You can also see any lexical variables declared within a subroutine. Every CV holds a special array[12] called a padlist. This padlist itself contains two arrays, one holding the name of lexical variables and the other containing an array of arrays holding the values for subsequent recursive invocations of the subroutine.[13]

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

[12] In an AV data structure that represents arrays.

[13] At least, it's something like that; it gets complex quickly.

Grabbing a list of all lexical variables declared in that scope is as simple as walking the appropriate array in the padlist:

```
sub introspect_sub
{
    my $sub      = shift;
    my $cv       = B::svref_2object( $sub );
    my ($names)  = $cv->PADLIST->ARRAY( );
            my $report   = join( ':',
        $cv->STASH->NAME( ), $cv->FILE( ), $cv->GV->LINE( ) . "\\n"
    );

    my @lexicals = map { $_->can( 'PV' ) ? $_->PV( ) : ( ) } $names->ARRAY( );
            return $report unless @lexicals;
            $report .= "\\t(" . join( ', ', @lexicals ) . ")\\n";
            return $report;
}
```

There's one trick and that's that the array containing the names of the lexicals doesn't *only* contain their names. However, knowing that the `B::OP`-derived objects holding the names will always have a `PV( )` method that returns a string representing the appropriate value of the scalar, the code filters out everything else. It works nicely, too:

```
use Data::Dumper;

package Foo;

sub foo
{
            my ($foo, $bar, $baz) = @_;
            }

package Bar;

sub bar { }
*foo = \\&Foo::foo;

package main;

warn introspect_sub( \\&Foo::foo );
warn introspect_sub( \\&Bar::bar );
warn introspect_sub( \\&Bar::foo );
warn introspect_sub( \\&Dumper );

# introspect_sub( ) as modified...
```

This outputs:

```
$ perl introspect_lexicals.pl
Foo:examples/introspect.pl:14
    ($foo, $bar, $baz)
Bar:examples/introspect.pl:18
Foo:examples/introspect.pl:14
    ($foo, $bar, $baz)
Data::Dumper:/usr/lib/perl5/site_perl/5.8.7/powerpc-linux/Data/Dumper.pm:495
```


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

Easy...at least once you've trawled through `perldoc B` and perhaps the Perl source code (*cv.h* and *pad.c*, if you really need details).

Hack 79. Find Imported Functions

Keep an eye on your namespace.

Importing functions is a mixed blessing. Having functions available from another namespace without having to type their full names is convenient. However, the chance for name collisions and confusion increases with the number of imported symbols.

There are multiple ways to tell the original package of a function, but many of them involve lots of deep magic and, in cases of generated functions, may not tell the whole story. If you really want to know what you've imported and when, the shortest and simplest approach is to use the `Devel::Symdump` module.

The Hack

To get a list of functions from a package, create a new `Devel::Symdump` object and use the `functions( )` method on it:

```
use Devel::Symdump;
my $symbols   = Devel::Symdump->new( 'main' );
my @functions = $symbols->functions( );
```

That gives you a list of fully-qualified function names as of the time of the call. Load and import from the other modules you need, and then create and query a *new* `Devel::Symdump` object to get a longer list of functions.

Running the Hack

Suppose you want to know what `File::Spec::Functions` imports.[14] If you can wedge the code to create and query the first Devel::Symdump object before the use line executes [Hack #70], all you have to do is perform an array intersection to remove duplicate elements.

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

[14] Sure, you could read the documentation, but your system administrator compressed the documentation and broke it.

```
use Devel::Symdump;

my %existing;

BEGIN
{
    my $symbols = Devel::Symdump->new( 'main' );
    @existing{ $symbols->functions( ) } = ( );
}

use File::Spec::Functions;

BEGIN
{
    my $symbols   = Devel::Symdump->new( 'main' );
    my @new_funcs =
        map { s/main::// ; $_ }
        grep { not exists $existing{ $_ } } $symbols->functions( );
    local $" = "\\n  ";
    warn qq|Imported:$"@new_funcs\\n|;
}
```

As of Perl 5.8.7, this prints:

```
$ perl show_fsf_symbols.pl
Imported:
  catfile
  curdir
  updir
  path
  file_name_is_absolute
  no_upwards
  canonpath
  catdir
  rootdir
$
```

Are you worried that this won't account for user-defined functions? Don't—by the time the `BEGIN` blocks run, Perl hasn't seen any yet. You're safe.

Hacking the Hack

`Devel::Symdump` works on more than just functions. You can find all exported scalars, arrays, hashes, and file and directory handles, as well as all other symbol tables beneath the named one. Beware, though, that all new symbols have a scalar created by default (at least in Perl prior to 5.9.3), so searching for those isn't as useful as you might think.

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

It would be easy to register a list of all exported functions with the using package, to allow more introspection and runtime. You could even write a module that does this and re-exports them to your package.

Hack 80. Profile Your Program Size

Find out how much memory your program takes, and then trim it!

The difference between a Perl program and a natively compiled binary is far more than just program convenience. Although the Perl program can do far more with less *source* code, in memory, Perl's data structures and bookkeeping can take up more space than you might think. Size matters sometimes—even if you have plenty of memory (if you're not trying to optimize for shared memory in a child-forking application, for example), a program with good algorithms and not tied to IO or incoming requests can still run faster if it has fewer operations to perform.

One of the best optimizations of Perl programs is trimming the number of operations it has to perform. The less work it has to do, the better.

This isn't an argument for obfuscated or golfed code—just good profiling to find and trim the few fat spots left in a program.

The Hack

When Perl compiles a program, it builds an internal representation called the optree. This represents every single discrete operation in a program. Thus knowing how many opcodes there are in a program (or module) and the size of each opcode is necessary to know where to start optimizing.

The `B::TerseSize` module is useful in this case.[15] It adds a `size( )` method to all ops. More importantly, it gives you detailed information about the size of all symbols in a package if you call `package_size( )`.

[15] It's *more* useful when used with mod_perl and `Apache::Status`; see http://modperlbook.org/html/ch09_04.html.

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

To find the largest subroutine in a package and report on its opcodes, use code something like:

```
use B::TerseSize;

sub report_largest_sub
{
    my $package                 = shift;
    my ($symbols, $count, $size) = B::TerseSize::package_size( $package );
    my ($largest)               =
        sort { $symbols->{$b}{size} <=> $symbols->{$a}{size} }
        grep { exists $symbols->{$_}{count} }
        keys %$symbols;

    print "Total size for $package is $size in $count ops.\\n";
    print "Reporting $largest.\\n";
    B::TerseSize::CV_walk( 'root', $package . '::' . $largest );
}
```

`package_size( )` returns three items: a reference to a hash where the key is the name of a symbol and the value is a hash reference with the count of opcodes for that symbol and the total size of the symbol, the total count of opcodes for the package, and the total size of the package.

`report_largest_sub( )` takes the name of a loaded package, finds the largest subroutine in that package (where the heuristic is that only subroutines have a `count` key in the second-level hash of the symbol information), prints some summary information about the package, and then calls `CV_walk( )` which prints a lot of information about the selected subroutine.

Running the Hack

The real meat of the hack is in interpreting the output. `B::TerseSize` displays statistics for every significant line of code in a subroutine. Thus, calling `report_largest_sub( )` on `Text::WikiFormat` will print pages of output for `find_list( )`:

```
Total size for Text::WikiFormat is 92078 in 1970 ops.
Reporting find_list.
UNOP   leavesub      0x10291e88 {28 bytes} [targ 1 - $line]
    LISTOP lineseq       0x10290050 {32 bytes}

------------------------------------------------------------
        COP    nextstate     0x10290010 {24 bytes}
        BINOP  aassign       0x1028ffe8 {32 bytes} [targ 6 - undef]
            UNOP   null          0x1028fd38 {28 bytes} [list]
                OP     pushmark      0x1028ffc8 {24 bytes}
                UNOP   rv2av         0x1028ffa8 {28 bytes} [targ 5 - undef]
                    SVOP   gv            0x1028ff88 {96 bytes}  GV *_
            UNOP   null          0x1028d660 {28 bytes} [list]
                OP     pushmark      0x1028fec0 {24 bytes}
                OP     padsv         0x1028fe68 {24 bytes} [targ 1 - $line]
                OP     padsv         0x1028fea0 {24 bytes} [targ 2 -
                                                            $list_types]
```


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

```
                OP      padsv         0x1028fee0 {24 bytes} [targ 3 - $tags]
                OP      padsv         0x1028ff10 {24 bytes} [targ 4 - $opts]

[line 317 size: 380 bytes]

-------------------------------------------------------------

(snip 234 more lines)
```

The final line gives the key to interpreting the output; it represents line 317 of the file defining this package:

```
315: sub find_list
316: {
317:     my ( $line, $list_types, $tags, $opts ) = @_;
318:
319:     for my $list (@$list_types)
```

This single line costs twelve opcodes and around 380 bytes[16] of memory. If this were worth optimizing, perhaps removing an unused variable would help.

[16] Give or take; `B::TerseSize` can only guess sometimes.

The previous lines list each op on this line in tree order. That is, the root of the branch is the `nextstate` control op. It has a sibling, the `leaveloop` binary op. You can ignore the memory address, but the size of the op in curly braces can be useful. Finally, some ops have additional information in square brackets—especially those referring to lexical variables.

The real use of this information is when you can compare two different implementations of an algorithm to each other to optimize for memory usage or number of ops. Sometimes the code with the fewest number of lines really isn't slimmer.

Hacking the Hack

Do you absolutely hate the output from `CV_walk( )`? Write your callback and use `B::walkoptree_slow( )` or `B::walkoptree_exec( )` to call it. Don't forget to use `B::TerseSize` to make the `size( )` method available on ops. You can get package and line number information from `nextstate` ops.

Unfortunately, doing this effectively probably means stealing the code from `B::TerseSize`. At least it's reasonably small and self-contained. Look for the methods declared in the `B::` namespace.

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

Hack 81. Reuse Perl Processes

Spend CPU time running programs, not recompiling them.

As nice as it is to be able to type and run a Perl program, compiled programs sometimes have a few advantages. Quickly running processes might spend most of their time launching, not running.

If you have a program you might want to run all the time, but it takes significant time to load the appropriate modules and get ready to run and you just can't spare that time on something that should execute immediately and get out of the way or that might have to run dozens of times per second under high load, trade a little memory for speed with `PPerl`.

The Hack

Matt Sergeant's `PPerl` module provides a mod_perl-like environment for normal Perl programs. A well-written program can run under `PPerl` with no modifications.

Suppose you use Chia-liang Kao's amazingly useful `SVK` distributed revision control system, written in Perl.[17] You're continually making lots of little checkins, and you've started to notice a bit of a lag as launching the program continually recompiles a handful of complex modules.

[17] If you don't use it already, try it.

Make a copy of the `svk` program in your path where your shell will find it before the system version. Edit the file and change the first line from:

```
#!/usr/bin/perl -w
```

... to:

```
#!/usr/bin/pperl -w
```

That's it!

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

Running the Hack

The first time you launch `svk`, it will take just about as long as normal. Subsequent launches will run much more quickly, as `PPerl` reuses the launched process—avoiding the repeated hit of compilation.

This works well for other processes too—mail filters written with `Mail::Audit` or `Mail::Filter`, `SpamAssassin`, and any Perl program that can run multiple times idempotently but usually takes little time to run.

Hacking the Hack

As an administrator, to make a persistent `svk` to share between every developer on the box, create an alias (or equivalent shell script) to launch `svk` with the `--anyuser` flag:

```
alias svk='/usr/bin/pperl -- --anyuser /usr/bin/svk'
```

Other useful flags include `--prefork` to tune the number of persistent processes to launch and `--maxclients` to set the maximum number of requests any child will serve before exiting. (This helps keep down memory usage, as multiple requests unshare more and more pages.)

One feature `PPerl` currently lacks is to shut down the persistent process after it goes unused for a period of time.

Hack 82. Trace Your Ops

Watch Perl execute individual instructions.

Unlike so-called natively compiled programs, Perl programs are instructions for the Perl virtual machine. When Perl compiles a program, it reduces it to a tree of opcodes, such as

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

"fetch this lexical variable" and "add the attached constant." If you want to see what your program is doing, the best[18] way is to examine each individual opcode.

[18] Though not necessarily the *easiest*.

The `B::*` modules—the compiler backend to Perl—give you some flexibility in examining compiled code from Perl. They don't give you many opportunities to play with ops as Perl runs them, however. Fortunately, `Runops::Trace` does.

The Hack

`Runops::Trace` replaces Perl's standard runloop with an alternate runloop that calls back to Perl code, passing the `B::*` object representing the next op that will run. This allows you to request and log any data from that op.

Perl's standard runloop executes the current op, fetches the next op after that, dispatches any signals that have arrived, and repeats.

For example, to count the number of accesses to global symbols within a program, write a callback logger:

```
package TraceGlobals;

use strict;
use warnings;

use Runops::Trace \\&trace_globals;

my %globals;

sub trace_globals
{
    return unless $_[0]->isa( 'B::SVOP' ) && $_[0]->name( ) eq 'gv';
    my $gv   = shift->gv( );
    my $data = $globals{ $gv->SAFENAME( ) } ||= { };
    my $key  = $gv->FILE( ) . ':' . $gv->LINE( );
    $data->{$key}++;
}

END
{
    Runops::Trace->unimport( );

    for my $gv ( sort keys %globals )
    {
        my $gv_data = $globals{ $gv };
        my @counts  = keys %$gv_data;

        for my $line ( sort { $gv_data->{$b} <=> $gv_data->{$a} } @counts)
```


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

```
        {
            printf "%04d %s %-> s\\n", $gv_data->{$line}, $gv, $line;

        }
    }
}

1;
```

The important work is in `trace_globals( )`. The subroutine first examines its only argument, throwing out all non-SV opcodes and all non-GV opcodes. (These are opcodes that access typeglobs, or GVs, as the Perl internals call them.) Then it fetches the GV object attached to the op, logging the name of the GV (`SAFENAME( )`) and the file (`FILE( )`) and line (`LINE( )`) where the symbol occurs.

The `END` block formats and reports this data nicely. The call to `Runops::Trace->unimport( )` at the start prevents the tracing module from accidentally trying to trace itself at the end of the program.

Running the Hack

Because of the way `Runops::Trace` installs its tracing runloop, you must load a tracing module *before* the code you want to trace. The easiest way to do this is from the command line, perhaps on the program from "Find All Symbols in a Package" [Hack #75]:

```
$ perl -MTraceGlobals find_package_symbols.pl
Foo:find_package_symbols.pl:14
        ($foo, $bar, $baz)
Bar:find_package_symbols.pl:18
Foo:find_package_symbols.pl:14
        ($foo, $bar, $baz)
Data::Dumper:/usr/lib/perl5/site_perl/5.8.7/powerpc-linux/Data/Dumper.pm:484
0001 AddrRef -> /usr/lib/perl5/5.8.7/overload.pm:94
0054 Bits -> /usr/lib/perl5/5.8.7/warnings.pm:189
0003 Cache -> /usr/lib/perl5/5.8.7/Exporter.pm:13
0002 DeadBits -> /usr/lib/perl5/5.8.7/warnings.pm:239
0001 Dumper -> /usr/lib/perl5/5.8.7/Exporter.pm:65
0001 EXPORT -> /usr/lib/perl5/site_perl/5.8.7/powerpc-linux/Data/Dumper.pm:24
0001 EXPORT_OK -> /usr/lib/perl5/site_perl/5.8.7/powerpc-linux/
        Data/Dumper.pm:25
0001 ISA -> /usr/lib/perl5/site_perl/5.8.7/powerpc-linux/Data/Dumper.pm:23
0002 Offsets -> /usr/lib/perl5/5.8.7/warnings.pm:136
0003 SIG -> /usr/lib/perl5/5.8.7/Exporter.pm:62
0001 StrVal -> /usr/lib/perl5/site_perl/5.8.7/powerpc-linux/
        Data/Dumper.pm:104
0037 _ -> :0
<...>
```

The first part of the output is the normal program output, as the program runs as normal. The second half of the output shows the number of accesses, the name of the symbol, and the file and line of the definition of the symbol. The final line is interesting—it shows the requests made for the glob named _, usually accessed as `@_` and not defined in a package or a file.

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

Hacking the Hack

Finding all of the global symbols is interesting, especially if you want to explore a certain code path where static analysis isn't helpful [Hack #77]. You can do much, much more with a tracing runloop. Consider that the callback function is basically the entry point into an event-driven state machine. Find the type of ops you want to query and perform your behavior based on that.

For example, to measure the amount of time you spend in one package over another, look for the `B::COP` objects that represent the `nextstate` op and keep timing information. To see when a variable changes, look for `B::SVOP` objects accessing that particular variable.

A future enhancement to `Runops::Trace` may allow you to *change* the next op, declining to handle dangerous or indelicate operations, or even redirecting to different ops. To learn more, read the documentation for `B` and become familiar with optrees with `B::Concise` and `B::Terse`.

Hack 83. Write Your Own Warnings

Improve static code checking.

You have `strict` under control. You know why you `use warnings`. Maybe you even use `B::Lint` to find problems. Are they truly enough for you? If you've ever wished that you could make `strict` stricter or make `warnings` preachier, you're in luck.

`Perl::Critic` is a similarly excellent tool that audits your code based on Damian Conway's Perl Best Practices (O'Reilly).

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

The Hack

It's impossible to override some built-in functions[19] [Hack #91] like `print( )` and `printf( )`. Usually `print( )` succeeds because it writes to an internal buffer—but occasionally Perl has to flush the buffer. `print( )` might fail if you write to a file on a full file system, to a closed handle, or for any of several other reasons. If you don't check `print( )` and `close( )` for success, you might lose data without knowing about it.

[19] Run `perl -MB::Keywords -le 'eval{ prototype $_ } or print for @B::Keywords::Functions'` after installing `B::Keywords` to see a complete list.

The best you can do for unoverridable functions is to create new warnings for unsafe code.

Here's *bad_style.pl*, a short program that opens a file and writes something to it. It has three misfeatures: ignoring the results of `print( )` and `close( )` and a terribly non-descriptive variable name:

```
open my $fh, '>>', 'bad_style.txt'
    or die "Can't open bad_style.txt for appending: $!\\n";
print {$fh} 'Hello!';
close $fh;
```

You *could* review every line of code in your system to find these errors. Better yet, teach `B::Lint` how to find them for you:

```
package B::Lint::VoidSyscalls;

use strict;
use warnings;

use B 'OPf_WANT_VOID';
use B::Lint;

# Make B::Lint accept plugins if it doesn't already.
use if ! B::Lint->can('register_plugin'),
    'B::Lint::Pluggable';


# Register this plugin.
B::Lint->register_plugin( __PACKAGE__, [ 'void_syscall' ] );

# Check these opcodes
my $SYSCALL = qr/ ^ (?: open | print | close ) $ /msx;

# Also look for things that are right at the end of a subroutine
# sub foo { return print( ) }
my $TERM = qr/ ^ (?: leavesub ) $/msx;

sub match
{
    my ( $op, $checks ) = @_;

    if (     $checks->{void_syscall}
         and $op->name( ) =~ m/$SYSCALL/msx )
    {
        if ( $op->flags() & OPf_WANT_VOID )
```


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

```
        {
            warn "Unchecked " .  $op->name( ) .  " system call "
                .  "at " .  B::Lint->file( ) .  " on line "
                .  B::Lint->line( ) .  "\\n";
        }
        elsif ( $op->next->name( ) =~ m/$TERM/msx )
        {
            warn "Potentially unchecked " .  $op->name( ) .  " system call "
                .  "at " .  B::Lint->file( ) .  " on line "
                .  B::Lint->line( ) .  "\\n";
        }
    }
}
```

As of Perl 5.9.3, `B::Lint` supports plugins. Earlier versions don't, so this code checks the version and loads a fallback if necessary.

This module also checks for system calls made in potentially void context at the end of functions—that is, where the next opcode is `leavesub`.

Running the Hack

Checking *bad_style.pl* with `B::Lint::VoidSyscalls` is easy:

```
$ perl -MB::Lint::VoidSyscalls -MO=Lint bad_style.pl
Unchecked print system call at bad_style.pl on line 3
Unchecked close system call at bad_style.pl on line 4
bad_style.pl syntax OK
```

Hacking the Hack

The idea is pretty general: find bad stuff in the optree ("Find All Global Variables" [Hack #77] shows how to mine the optree) and tell the user about it. There are plenty of possibilities to add more strictness to your OO code—checking that a class actually exists for class method calls, that the methods being called on those classes exist, and even that the methods being called are appropriate methods for certain classes. Here's an alternate `match( )` subroutine that does just that:

```
sub match
{
    my $op = shift;

    if ( $op->name() eq 'entersub' )
    {
        my $class  = eval { $op->first->sibling         ->sv->PV };
        my $method = eval { $op->first->sibling->sibling->sv->PV };
```


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

```
        if ( defined $class )
        {
            no strict 'refs';

            # check strict classes
            if ( not %{ $class . '::' } )
            {
                B::Lint::warning "Class $class doesn't exist";
            }
            # check strict class methods
            elsif ( defined $method and not $class->can($method) )
            {
                B::Lint::warning "Class $class can't do method $method";
            }
        }
        elsif (     defined $method
                and not grep { $_->can($method) } classes( B::Lint->file() ) )
        {
            B::Lint::warning "Object can't do method $method";
        }
    }
}

use File::Slurp 'read_file';

my %classes;
sub classes
{
    my $file = shift;
    $classes{$file} ||= scalar {
        map { $_ => 1 }
        grep { defined %{ $_ . '::' } }
        read_file($file) =~ m/( \\w+ (?: (?:::|')\\w+ )* )/msxg
    };
    return keys %{ $classes{$file} };
}
```


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