Table of Contents

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

Chapter 1. Productivity Hacks

Hacks 1-11

Everyone wants to be more productive. That's probably why you use Perl: to get more work done in less time with less work.

Productivity isn't all about saving time, though. Saving effort is even more important, whether you mean finding the information you want, automating away repeated tasks, or finding ways not to have to think about things that you do all the time. In some ways, this is the notion of *relentless automation*—finding every little niggling task that always interrupts your current project by being so annoying, difficult, cumbersome, or different and then hiding it behind an alias, a shell script, a process, or whatever.

Here are a few ideas for ways to make your programming life easier and more productive. Try them, enjoy your new sense of free time, and let yourself notice the new points of friction in your life. Then solve them, too!

Hack 1. Add CPAN Shortcuts to Firefox

Keep module documentation and distributions mere keystrokes away.

If Perl has only one advantage over other programming languages, it's the number of modules on the CPAN (http://www.cpan.org/) that solve so many problems effectively. That brings up a smaller problem, though—choosing an appropriate module for the job.

http://search.cpan.org/ helps, but if you visit the site many times a day, the steps to start a search through the web interface can become annoying. Fortunately, the Mozilla family of web browsers, including Mozilla Firefox, let you set up shortcuts that make browsing much easier. These shortcuts are just bookmarked URLs with substitutable sections and keywords, but they're very powerful and useful—almost command-line aliases ("Make the Most of Shell Aliases" [Hack #4]) for your browser.

Here are three of the most useful.

Search for a Module

The first technique is to find the module you want. Normally, you could visit the CPAN search site, type the appropriate words in the box, submit the form, and browse through the results. That's too much work though!

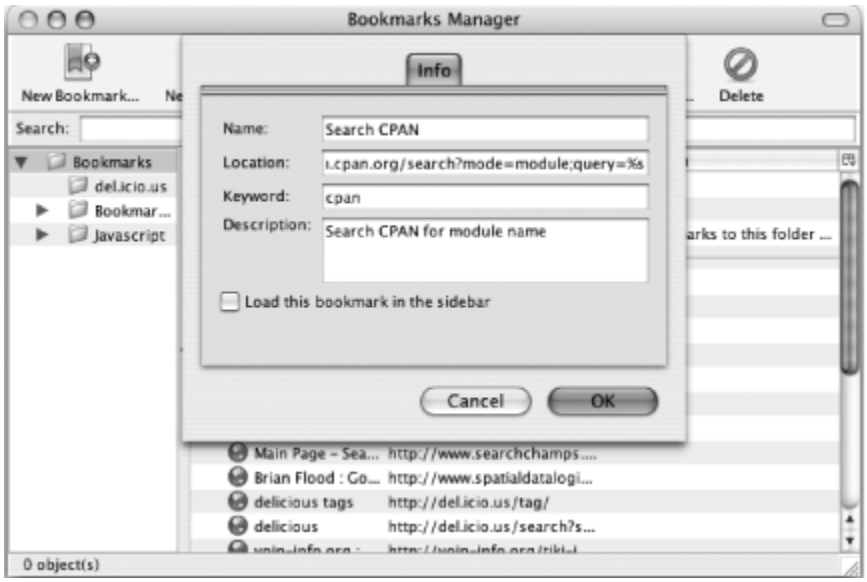
Open the bookmark menu in your browser; this is Bookmarks→Manage Bookmarks in Mozilla Firefox. Create a new bookmark. For name, put **`Search CPAN`** and for Keyword enter **`cpan`**. In the Location box, type:

```
http://search.cpan.org/search?mode=module;query=%s
```


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

Figure 1-1 shows the completed dialog box. Press OK, then go back to the browser. Clear the location bar, then type **cpanAcme** and hit Enter. This will take you immediately to the first page of search results for modules with `Acme` in their names.

Figure 1-1. Creating a new keyword bookmark search



Read Module Documentation

If you know exactly the name of the module you want, it's more convenient to jump straight to information about that module. Create a new bookmark named **Show Module Documentation**, with the keyword of **cpod** and the location:

```
http://search.cpan.org/perldoc/%s
```

Press OK, then type **cpod Test::Builder** and press Enter. You'll see the latest version of the `Test::Builder` documentation.

This doesn't seem to work for POD-only modules, such as `Test::Tutorial`. Also, beware that the case must match exactly.

Find Module Comments

Sometimes it's more valuable to find advice from other people about a module, especially when you may have uncovered a bug or something inexplicable in the documentation. The AnnoCPAN project (http://www.annocpan.org/) is a public site that allows users to annotate the documentation of any CPAN module. This is a good way to share your hard-won knowledge about a module with the world.

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

Create a new bookmark yet again, with a name of **`AnnoCPAN Module Documentation`** and a keyword of **`apod`**. Set the location to:

```
http://www.annocpan.org/?mode=search;field=Module;latest=1;name=%s
```

Save the bookmark, then type **`apod GraphViz`** in the browser's location bar and press Enter. Scroll down a few pages and you should see notes on various paragraphs of the documentation.

Hacking the Hack

The keyword search feature of Firefox turns your browser's address bar into a command line. It's simple to write your own CGI script or mod_perl handler to add a new command to the browser—all it has to do is take a query string and return information. You could easily write code to implement a single command that aggregates different documentation sources (for example, you can search JavaScript and HTML and Perl documentation with a single query).

The URL of the bookmark can be a `javascript:` URL that runs code in the browser. In essence you're creating a bookmarklet that you trigger on the command line. You could use JavaScript to open the search results in a new window or tab or search for the currently selected text.

Hack 2. Put Perldoc to Work

Do more than just read the documentation.

Perl has a huge amount of documentation available through the `perldoc` utility—and not just from the command line. These docs cover everything from the core language and tutorials through the standard library and any additional modules you install or even write. `perldoc` can do more, though.

Here are a few switches and options to increase your productivity.

Find Operator Documentation

The `perlfunc` document lists every built-in operator in the language in alphabetical order. If you need to know the order of arguments to `substr( )`, you *could* type **`perldoc perlfunc`**, and then search for the correct occurrence of `substr`.

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

In a decent pager, such as `less` on a Unix-like system, use the forward slash (`/`) to begin a search. Type the rest of the name and hit Enter to begin searching. Press `n` to find the next occurrence and `N` to find the previous one.

Why search yourself, though? `perldoc`'s `-f` switch searches `perlfunc` for you, presenting only the documentation for the named operator. Type instead:

```
$ perldoc -f substr
```

The program will launch your favorite pager, showing only the documentation for `substr`. Handy.

Answer a FAQ

The Perl FAQ is a very useful piece of the core documentation, with a table of contents in `perlfaq` and nine other documents (`perlfaq1` through `perlfaq9`) full of frequently asked questions and their answers.

Searching every document for your question, however, is more tedious than searching `perlfunc`. (Do skim `perlfaq` once in a while to see what questions there are, though.) Fortunately, the `-q` switch allows you to specify a search pattern for FAQ keywords.

If you remember that somewhere the FAQ explains how to shuffle an array, but you can't remember where, try:

```
$ perldoc -q shuffle
```

As with the `-f` switch, this will launch your favorite pager to view every question with the term `shuffle` in the title.

`-q` also handles regular expressions, so if you want to search for every mention of Perl 6, with or without that blessed space, try:

```
$ perldoc -q "Perl ?6"
```

The quotes prevent the shell from interpreting the space as an argument separator.

Webify It

Maybe the command line isn't your thing. Maybe you work in a group of programmers who won't leave their comfortable IDEs long enough to type a few commands—and who certainly won't read documentation from anywhere but the IDE or a web page.

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

That's okay. `perldoc` can produce HTML (or any other type of output for which you have a POD translator installed), too. Use the `-o` switch with your preferred output format. To turn `perltoc` into HTML, use the command:

```
$ perldoc -oHTML -dperltoc.html perltoc
```

The `-d` switch specifies the destination filename.

Valid HTML formatters include any of `Pod::Perldoc::HTML`, `Pod::Simple::HTML`, and `Pod::HTML`. If you have another formatter of the appropriate name installed, you can use it.

If you have multiple potential formatters for a type installed, use -M*`full_module_name`* instead of `-o` to disambiguate.

Find that Module!

Maybe you already know how to find, slice, and dice the documentation. Have you ever run a program that picked up the wrong version of a module? Sure, you can modify the program to print `%INC` and `@INC` and crawl through the output to see what went wrong—but `perldoc` has to be able to figure out where the module lives to show its documentation. Exploit it!

The `-l` switch tells `perldoc` to find the named module (or document) and print its location instead of formatting and displaying the text. Here's where `Test::Tutorial` and `perlunintro` live on my system:

```
$ perldoc -l Test::Tutorial
/usr/lib/perl5/vendor_perl/5.8.7/Test/Tutorial.pod

$ perldoc -l perluniintro
/usr/lib/perl5/5.8.7/pod/perluniintro.pod
```

If you have multiple versions of Perl installed, be sure you use the *correct* version of `perldoc`; it uses the `@INC` path in its own version of Perl.

This can be much faster than doing a `locate` or `find` and `grep` from the command line.

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

Browse the Code

`perldoc -l` is pretty useful, especially if you want to know where a module is, so that you can look inside it. One more switch makes that even more useful, however. The `-m` option shows the plain, unrendered text of the named module or document in your favorite pager.

If you suspect that the author of `Test::MockObject` has hidden some useful methods from you,[1] browse the source of the module with:

[1] He hasn't.

```
$ perldoc -m Test::MockObject
```

You can't *edit* the text of the module from here, but being able to read it—or being able to read the raw POD of a module with POD errors that cause its formatting to fail—can be very helpful.

Likewise, the `-u` option shows only the unformatted POD source, without the code.

Hack 3. Browse Perl Docs Online

Host your own HTML documentation.

`perldoc` is a fine way to view the documentation for Perl and all your installed modules and to output them in the file format of your choice ("Put Perldoc to Work" [Hack #2]). `perldoc`'s little brother, `podwebserver`, is an even handier way to browse documentation—and bookmark it, and search it, and sometimes even hardcopy it, all through whatever web browser you're using this week.

The Hack

`podwebserver` provides basically `perldoc`-as-HTML over HTTP. Sure, you could always just browse the documentation at http://search.cpan.org/—but using `podwebserver` means that you'll be seeing the documentation for exactly *your* system's Perl version and module versions.

`podwebserver`'s HTML is compatible with fancy browsers as well as with more lightweight tools such as `lynx`, `elinks`, or even the `w3m` browser in Emacs. In fact, there have been persistent rumors of some users adventurously accessing `podwebserver` via cell phones, or even using something called "the Micro-Soft Internet Explorer." O'Reilly Media, Inc. can neither confirm nor deny these rumors.

If `podwebserver` isn't on your system, install the `Pod::Webserver` module from CPAN.

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

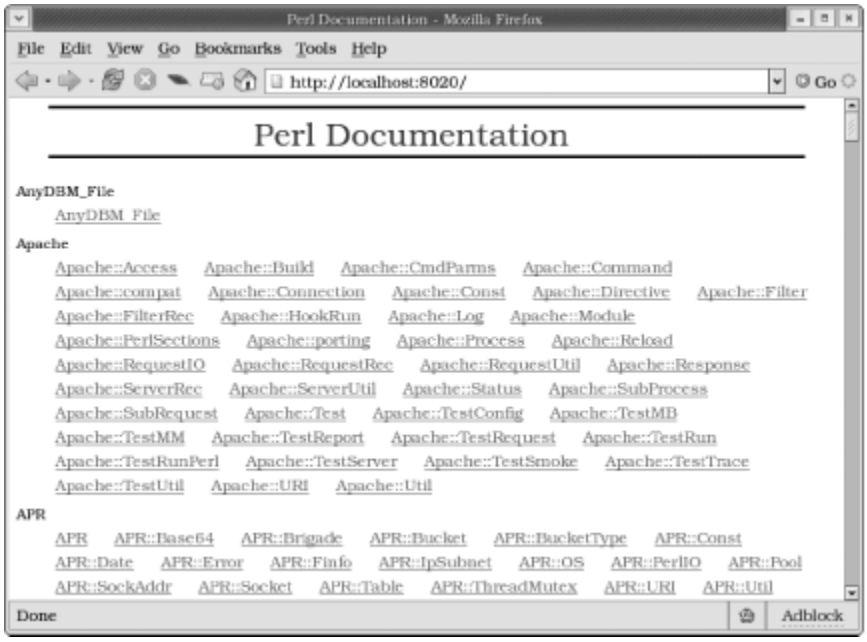
Running the Hack

To run `podwebserver`, just start it from the command line. You don't need `root` access:

```
$ podwebserver
```

Then start a web browser and browse to http://localhost:8020/. You'll see the index of the installed documentation (Figure 1-2).

Figure 1-2. An index of your Perl documentation



If you don't want to bind the web server to `localhost`, or if you have something already running on port 8020, use the `-H` and `-p` arguments to change the host and port.

```
$ podwebserver -H windwheel -p 8080
```

Hacking the Hack

Running a program and switching to your web browser to view a bookmark is too much work when you just want to check some documentation. Make your life easier with a little shell script ("Make the Most of Shell Aliases" [Hack #4]):

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

```
#!/bin/sh

podwebserver &
sleep 2
firefox -remote 'openurl( http://localhost:8020/, new-tab)'
```

Save the program as *~/bin/podweb*, make it executable (`chmod +x ~/bin/podweb`), make sure *~/bin/podweb* is in your `$PATH`, then run it:

```
$ podweb
```

If you have Mozilla Firefox open, this will pop up the index page in a new tab. Other web browsers have similar invocation schemes.

Hack 4. Make the Most of Shell Aliases

Make programming easier by programming your shell.

Perl is a language for people who type. It grew up from the shell to write all kinds of programs, but it still rewards people who don't mind launching programs from the command line.

If you spend your time writing Perl from the command line (whether you write short scripts or full-blown programs), spending a few minutes automating common tasks can save you lots of development time—and even more trouble.

Configuring Your Shell

The single most useful shell trick is the `realias` command. Normally creating a persistent alias means adding something to your *.bashrc* (or equivalent) file, starting a new shell, testing it, and then repeating the process until you get it right. Wouldn't it be nice to be able to edit and test a new alias in a single process?

Edit your *.bashrc* file and add a single line:

```
source ~/.aliases
```

Then create the file *~/.aliases*, containing:

```
alias realias='$EDITOR ~/.aliases; source ~/.aliases'
```


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

If you prefer `tcsh`, edit your *.cshrc* file. Then replace the = sign with a single space in all of the alias declarations.

Launch a new shell. Type the command `realias` and your favorite editor (assuming you have the `EDITOR` environment variable set, and if you don't something is *weird*) will open with your *~/.aliases* file. Add a line and save and quit:

```
alias reperl='perl -de0'
```

Now type `reperl` [2] at the command line:

[2] A pronounceable version of REPL—Read, Evaluate, Print, and Loop.

```
$ reperl

Loading DB routines from perl5db.pl version 1.28
Editor support available.

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

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

Within a single shell session you've identified a useful command that may be difficult to remember, automated it, and have started to use it productively. Nifty.

Useful Shell Aliases

What makes a good shell alias for Perl programming? Obviously a command that's difficult to remember, such as the one to put the Perl debugger into pseudo-interactive mode. Another good approach is to alias commands that are lengthy or otherwise difficult to type. One final category is a series of chained commands you find yourself typing often.

Here are a few examples. Change the paths as necessary, of course, but have fun removing a little more of the tedium from your life every time you notice yourself repeating something you could automate away. That's the Perl way.

Juggle multiple Perl versions

Suppose you're in the midst of upgrading Perl versions while you still have to maintain an older installation. You might have multiple versions of Perl installed. Instead of typing different paths all the time and instead of relying on tab completion to differentiate between `perl5.8.8` and `perl5.6.2` and so on, make the names different at the start:

```
alias newperl='/usr/local/bin/perl5.8.8'
alias oldperl='/usr/local/bin/perl5.6.2'
```


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

This is especially handy if you have a system Perl installed and don't want to break things by overwriting it.

Juggle multiple module versions

Suppose that you also must test your code against multiple versions of a module or library. For example, you need to know if your code works against version 4.x *and* 5.x of a database. Alias away the different library paths:

```
alias newdbperl='perl -M/home/dev/newlib/'
alias olddbperl='perl -M/home/dev/oldlib/'
```

Don't forget multipart commands

If you're a rigorous tester, you've likely encountered `Devel::Cover`. Though it's easy to use, it takes multiple steps to write a new report. Alias that away!

```
alias testcover='cover -delete; ./Build testcover; cover'
```

Remember configuration options

Suppose that you decide to test the Pugs project (http://www.pugscode.org/) and want to embed both Perl 5 and Parrot. Because Pugs undergoes such rapid development, you might have to run its *Makefile.PL* several times a week. Why make yourself remember how to configure it with the correct options every time? Alias it!

```
alias makepugs='PARROT_PATH="/home/chromatic/dev/parrot" \\
    PUGS_EMBED="perl 5 parrot" \\
    perl Makefile.PL && make'
```

Find a module's version

Sometimes you really need to know the version of a module—especially when you're tracking down a bug across multiple machines or pondering an upgrade. Typing out:

```
$ perl -MCGI::Application -le 'print CGI::Application->VERSION'
4.03
```

every time is too much work. Stick a function instead in your *.bashrc*:

```
function pmver ( ) { perl -M$1 -le "print $1->VERSION"; }
```

You can also add more error checking and turn it into an alias:

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

```
alias pmver="perl -le '\\$m = shift; eval qq(require \\$m)
   or die qq(module \\"\\$m\\" is not installed\\\\n); print \\$m->VERSION'"
```

Either way, run it as `pmver`:

```
$ pmver CGI::Application
4.03
```

Change Unix paths to Windows paths and back

These aliases work on Windows under Cygwin too. Even though it's still Windows on one side and Unix on the other, there's no reason you can't make it work correctly. Here's an alias that translates a Unix path to a Windows path and executes the Windows version of `gvim` on the file:

```
alias gvim='perl -we "exec q{/cygdrive/c/Progra~1/Vim/vim63/gvim.exe},
  map { s/^(.*)$/(-f \\$1)?qx{cygpath -aw \\"\\$1\\"}:\\$1/e; chomp; \\$_; }
  (@ARGV); "'
```

Launching general Windows programs from `bash` requires a similar hack:

```
alias winrun='exec 'cmd', "/c", ((split '/',$0)[-1], map {
  s/^(.*)$/(-f $1)?qx{cygpath -w "$1"}:$1/e;chomp;$_; } (@ARGV));'
```

Now you can launch non-Cygwin programs with arguments.

Hack 5. Autocomplete Perl Identifiers in Vim

Why type a full identifier if your editor can do it for you?

Good variable and function names are a great boon to productivity and maintainability, but brevity and clarity are often at odds. Instead of wearing out your keys, fingertips, and memory, consider making your text editor do the typing for you.

The Hack

If you use Vim, you have access to a handy autocompletion mechanism. In insert mode, type one or more letters of an identifier, then hit CTRL-N. The editor will complete your identifier using the first identifier in the same file that starts with the same letter(s). Hitting CTRL-N again gives you the second matching identifier, and so on.

This can be a real timesaver if you use long variable or subroutine names. As long as you've already typed an identifier once in a file, you can autocomplete it ever after, just by typing the first few letters and then CTRL-Ning to the right name:

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

```
sub find_the_factorial_of
{
    my ($the_number_whose_factorial_I_want) = @_;

    return 1 if $the_n<CTRL-N> <= 1;

    return $the_n<CTRL-N> * find<CTRL-N>($the_n<CTRL-N> - 1);
}
```

Unfortunately, Vim's idea of an identifier (in Vim-speak, a "keyword") isn't as broad as Perl's. Specifically, the editor doesn't recognize the colon character as a valid part of an identifier, which is annoying if you happen to like multipart class names, or qualified package variables.

However, it's easy to teach Vim that those intervening double-colons are valid parts of the identifiers. Add them to the editor's list of keyword characters by adding the line to your *.vimrc* file:

```
set iskeyword+=:
```

Then the following works too:

```
use Sub::Normal;

my $sub = Sub<CTRL-N>->new( );  # Expands to: Sub::Normal->new( )
```

Finding Identifiers Automatically

Of course, you still have to type the full name of `Sub::Normal` once, as part of the initial `use` statement. That really isn't as Lazy as it could be. It would be much better if Vim just magically knew about all the Perl modules you have installed and could cleverly autocomplete their names from the very first time you used them.

As it happens, that's easy to arrange as well. You just need a file that lists every module you have installed. Then tell Vim (in *.vimrc* again) to use all the identifiers in that file as a second source of keyword completions:

```
set complete+=k~/.vim_extras/file_that_lists_every_installed_Perl_module
```

The `complete+=k` tells Vim you're adding to the existing sources of completions for keywords. The path name that follows specifies the file containing the additional completions.

All you need is a simple Perl script to generate that file for you:

```
use File::Find 'find';

# Where to create this list...
my $LIST_DIR  = "$ENV{HOME}/.vim_extras/"
my $LIST_FILE = "file_that_lists_every_installed_Perl_module";

# Make sure the directory is available...
unless (-e $LIST_DIR )
{
    mkdir $LIST_DIR
        or die "Couldn't create directory $LIST_DIR ($!)\\n";
}
```


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

```
# (Re)create the file...
open my $fh, '>', "$LIST_DIR$LIST_FILE"
    or die "Couldn't create file '$LIST_FILE' ($!)\\n";

# Only report each module once (the first time it's seen)...
my %already_seen;

# Walk through the module include directories, finding .pm files...
for my $incl_dir (@INC)
{
    find
    {
        wanted => sub
        {
            my $file = $_;

            # They have to end in .pm...
            return unless $file =~ /\\.pm\\z/;

            # Convert the path name to a module name...
            $file =~ s{^\\Q$incl_dir/\\E}{ };
            $file =~ s{/}{::}g;
            $file =~ s{\\.pm\\z}{ };

            # Handle standard subdirectories (like site_perl/ or 5.8.6/)...
            $file =~ s{^.*\\b[a-z_0-9]+::}{ };
            $file =~ s{^\\d+\\.\\d+\\.\\d+::(?:[a-z_][a-z_0-9]*::)?}{ };
            return if $file =~ m{^::};

            # Print the module's name (once)...
            print {$fh} $file, "\\n" unless $already_seen{$file}++;
        },
        no_chdir => 1,
    }, $incl_dir;
}
```

Of course, you don't have to call the file `.vim_extras/file_that_lists_every_installed_Perl_module`. Just change the `$LIST_DIR` and `$LIST_FILE` variables to something saner.

Hacking the Hack

It's a natural next step to automate the generation of this file via `cron`. Beyond that, though, consider using Vim auto-commands to update the module list when you load and save files. To get information on auto-commands, type `:help autocmd-intro` within Vim. You could also check in and check out these module lists from a central repository to ensure that your editor knows about the class your coworker just added.

For a final coup-de-grace, consider extracting variable and subroutine names from the files as well. This will let you complete method names and exported variables. You could do this with regular expressions as heuristics, or through modules such as `Parse::Perl`.

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

Emacs users take heart. You can usually find equivalents by searching the web for `taskname` cperl-mode. Here's an autocompletion minor mode to add to your *~/.emacs* file:

```
(defadvice cperl-indent-command
    (around cperl-indent-or-complete)
    "Changes \\\\[cperl-indent-command] so it autocompletes when at the end of a word."
    (if (looking-at "\\>")
        (dabbrev-expand nil)
      ad-do-it))
  (eval-after-load "cperl-mode"
    '(progn (require 'dabbrev) (ad-activate 'cperl-indent-command)))
```

Hack 6. Use the Best Emacs Mode for Perl

Configure Emacs for easy Perl coding.

While perl-mode is the classic Perl-editing mode that Emacs uses for Perl files by default, most Perl programmers prefer the newer cperl-mode. (The "c" in the name is because its early versions borrowed code from c-mode. It's not actually written in C, nor meant for C.) Enabling it is easy.

The Hack

cperl-mode is probably already included in your version of Emacs, but you can get an up-to-date version from http://math.berkeley.edu/~ilya/software/emacs/. Save it to an Emacs library directory. Then enable it for *.pl* and *.pm* files by adding nine lines to your *~/.emacs* file:

```
(load-library "cperl-mode")
  (add-to-list 'auto-mode-alist '("\\\\.[Pp][LlMm][Cc]?$" . cperl-mode))
  (while (let ((orig (rassoc 'perl-mode auto-mode-alist)))
             (if orig (setcdr orig 'cperl-mode))))
  (while (let ((orig (rassoc 'perl-mode interpreter-mode-alist)))
           (if orig (setcdr orig 'cperl-mode))))
  (dolist (interpreter '("perl" "perl5" "miniperl" "pugs"))
    (unless (assoc interpreter interpreter-mode-alist)
      (add-to-list 'interpreter-mode-alist (cons interpreter 'cperl-mode))))
```

What can you do with it?

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

Put Perldoc at your fingertips

cperl-mode provides a handy function for calling `perldoc`, but does not associate it with any key by default. To put it at your fingertips, add one line to your *.emacs* file:

```
(global-set-key "\\M-p" 'cperl-perldoc) ; alt-p
```

If you want to use Pod::Webserver [Hack #3], use one of the various in-Emacs web browsers:

```
(global-set-key "\\M-p" '(lambda ( ) (interactive)
  (require 'w3m)
  (w3m-goto-url "http://localhost:8020/")
))
```

If you prefer your normal web browser, just set some particular key to start it up on the `Pod::Webserver` page:

```
(global-set-key "\\M-p"
  '(lambda ( ) (interactive) (start-process "" nil
  "firefox" "http://localhost:8020/"
  ; Or however you launch your favorite browser, like:
  ;   "gnome-terminal" "-e" "lynx http://localhost:8020/"
  ;   "xterm" "-e" "elinks http://localhost:8020/"
)))
```

Use a special mode just for Pod

One problem with both cperl-mode and perl-mode is that they both treat Pod the same: they just ignore it. To get better syntax highlighting for Pod, switch to the pod-mode. It probably isn't part of your Emacs distribution, so you download the latest version from http://www.cpan.org/authors/id/S/SC/SCHWIGON/pod-mode/.

Once installed, enable it in your *.emacs* file with:

```
(require 'pod-mode)
(add-to-list 'auto-mode-alist
  '("\\.pod$" . pod-mode))

; You might appreciate turning on these
;   features by default for Pod:

(add-hook 'pod-mode-hook '(lambda ( ) (progn
 (font-lock-mode)   ; =syntax highlighting
 (auto-fill-mode 1) ; =wordwrap
 (flyspell-mode 1)  ; =spellchecking

)))
```


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

Hack 7. Enforce Local Style

Keep your code clean without editing it by hand.

One of the first barriers to understanding code written by others is that their formatting style may not match yours. This is especially true if you find yourself maintaining code that, at best, has grown with little direction over the years. Whether you work with other developers and want to maintain a consistent set of coding guidelines, or you want to find *some* structure in a big ball of mud, `perltidy` can help untangle and bring consistency to even the scariest code.

The Hack

Install the CPAN module `Perl::Tidy`. This will also install the `perltidy` utility. Now you can use it!

From the command line

Run `perltidy` on a Perl program or module and it will write out a tidied version of that file with a *.tdy* suffix. For example, given *poorly_written_script.pl*, `perltidy` will, if possible, reformat the code and write the new version to *poorly_written_script.pl.tdy*. You can then run tests against the new code to verify that it performs just as did the previous version (even if it is much easier to read).

This command reformats the contents of *some_ugly_code.pl* so that it's no longer, well, *ugly*. How effective is it? The Perltidy docs offer an example. Before:

```
$_= <<'EOL';
   $url = URI::URL->new( "http://www/" );   die if $url eq "xXx";
EOL
LOOP:{print(" digits"),redo LOOP if/\\G\\d+\\b[,.;]?\\s*/gc;print(" lowercase"),
redo LOOP if/\\G[a-z]+\\b[,.;]?\\s*/gc;print(" UPPERCASE"),redo LOOP
if/\\G[A-Z]+\\b[,.;]?\\s*/gc;print(" Capitalized"),
redo LOOP if/\\G[A-Z][a-z]+\\b[,.;]?\\s*/gc;
print(" MiXeD"),redo LOOP if/\\G[A-Za-z]+\\b[,.;]?\\s*/gc;print(
" alphanumeric"),redo LOOP if/\\G[A-Za-z0-9]+\\b[,.;]?\\s*/gc;print(" line-noise"
),redo LOOP if/\\G[^A-Za-z0-9]+/gc;print". That's all!\\n";}
```

After:

```
$_ = <<'EOL';
   $url = URI::URL->new( "http://www/" );   die if $url eq "xXx";
EOL
LOOP: {
    print(" digits"),       redo LOOP if /\\G\\d+\\b[,.;]?\\s*/gc;
    print(" lowercase"),    redo LOOP if /\\G[a-z]+\\b[,.;]?\\s*/gc;
    print(" UPPERCASE"),    redo LOOP if /\\G[A-Z]+\\b[,.;]?\\s*/gc;
    print(" Capitalized"),  redo LOOP if /\\G[A-Z][a-z]+\\b[,.;]?\\s*/gc;
    print(" MiXeD"),        redo LOOP if /\\G[A-Za-z]+\\b[,.;]?\\s*/gc;
```


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

```
    print(" alphanumeric"), redo LOOP if /\\G[A-Za-z0-9]+\\b[,.;]?\\s*/gc;
    print(" line-noise"),   redo LOOP if /\\G[^A-Za-z0-9]+/gc;
    print ". That's all!\\n";
}
```

Big difference!

Perltidy is of course great for enforcing a particular coding style as you work, but it's also a lifesaver when the task of maintaining someone else's spaghetti code suddenly falls on you.

The default is good for the paranoid. For the adventurous, use the `-b` flag, which modifies the files in place and writes the originals to backup files. For example running `perltidy -b scary_script.pl` will produce a tidied *scary_script.pl*, if possible, and a *scary_script.pl.bak*.

This operation is not idempotent—`perltidy` *will* overwrite an existing backup file of the same name, if it exists.

The default formatting options may be inappropriate for your use. `Perl::Tidy` looks for a *.perltidyrc* file, first in your current directory, next in your home directory, and then in system-wide directories. The contents of this file are simple; they're the same command line switches that `perltidy` uses. For example, the author's preferred *.perltidyrc* file contains:

```
-ci=4 # indent 4 spaces when breaking a long line
-et=4 # replace 4 leading spaces with a tab
-bl   # place opening braces on newlines
```

See `man perltidy` for a complete list of formatting options.

Within Vim

The `perltidy` program is also useful from within text editors that can call external programs. This makes it possible to tidy code within an editor, without saving and opening external files—it's great for figuring out what poorly indented code does. From Vim, run it on the entirety of the current buffer with the ex command `%! perltidy`. It also makes a great Vim map—add to your *.vimrc* file something like:

```
map ,pt  <Esc>:%! perltidy<CR>
map ,ptv <Esc>:'<,'>! perltidy<CR>
```

Then in edit mode, type **`,pt`** and `perltidy` will reformat the contents of the current buffer. Select a region and **`,ptv`** will format its contents.

If you have a coding style that differs from the default values, add the command-line options to the maps.

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

Within Emacs

If you use Emacs to edit your Perl code, you can be virtuously lazy when it comes to reformatting your code. Just drop a bit of code into your *~/.emacs* file and restart Emacs:

```
(defmacro mark-active ()
    "Xemacs/emacs compatibility macro"
    (if (boundp 'mark-active)
        'mark-active
      '(mark)))
(defun perltidy ( )
  "Run perltidy on the current region or buffer."
  (interactive)
  ; Inexplicably, save-excursion doesn't work here.
  (let ((orig-point (point)))
    (unless (mark-active) (mark-defun))
    (shell-command-on-region (point) (mark) "perltidy -q" nil t)
    (goto-char orig-point)))
(global-set-key "\\C-ct" 'perltidy)
```

Then the next time you open up a file full of spaghetti Perl, just hit **`C-c t`** and watch as the "paragraph" of nearby code magically becomes legible! Better yet, if you want to reformat the entire file, hit **`M-x mark-whole-buffer`** and then **`C-c t`**.

To make Emacs tidy your code automatically when you save it, add this snippet of code:

```
(defvar perltidy-mode nil
    "Automatically 'perltidy' when saving.")
  (make-variable-buffer-local 'perltidy-mode)
  (defun perltidy-write-hook ()
    "Perltidys a buffer during 'write-file-hooks' for 'perltidy-mode'"
    (if perltidy-mode
        (save-excursion
          (widen)
          (mark-whole-buffer)
          (not (perltidy)))
      nil))
  (defun perltidy-mode (&optional arg)
    "Perltidy minor mode."
    (interactive "P")
    (setq perltidy-mode
          (if (null arg)
              (not perltidy-mode)
            (> (prefix-numeric-value arg) 0)))
    (make-local-hook 'write-file-hooks)
    (if perltidy-mode
        (add-hook 'write-file-hooks 'perltidy-write-hook)
      (remove-hook 'write-file-hooks 'perltidy-write-hook)))
  (if (not (assq 'perltidy-mode minor-mode-alist))
      (setq minor-mode-alist
            (cons '(perltidy-mode " Perltidy")
                  minor-mode-alist)))
  (eval-after-load "cperl-mode"
    '(add-hook 'cperl-mode-hook 'perltidy-mode))
```

Run `M-x perltidy-mode` to disable or re-enable the automatic code tidying.

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

Hack 8. Don't Save Bad Perl

Don't even write out your file if the Perl isn't valid!

Perl tests tend to start by checking that your code compiles. Even if the tests don't check, you'll know it pretty quickly as all your code collapses in a string of compiler errors. Then you have to fire up your editor again and track down the problem. It's simple, though, to tell Vim that if your Perl code won't compile, it shouldn't even write it to disk.

Even better, you can load Perl's error messages back into Vim to jump right to the problem spots.

The Hack

Vim supports filetype plug-ins that alter its behavior based on the type of file being edited. Enable these by adding a line to your *.vimrc*:

```
filetype plugin on
```

Now you can put files in *~/.vim/ftplugin* (*My Documents_vimfiles\\ftplugin* on Windows) and Vim will load them when it needs them. Perl plug-ins start with *perl_*, so save the following file as *perl_synwrite.vim*:

```
" perl_synwrite.vim: check syntax of Perl before writing
" latest version at: http://www.vim.org/scripts/script.php?script_id=896

"" abort if b:did_perl_synwrite is true: already loaded or user pref
if exists("b:did_perl_synwrite")
  finish
endif
let b:did_perl_synwrite = 1

"" set buffer :au pref: if defined globally, inherit; otherwise, false
if (exists("perl_synwrite_au") && !exists("b:perl_synwrite_au"))
  let b:perl_synwrite_au = perl_synwrite_au
elseif !exists("b:perl_synwrite_au")
  let b:perl_synwrite_au = 0
endif

"" set buffer quickfix pref: if defined globally, inherit; otherwise, false
if (exists("perl_synwrite_qf") && !exists("b:perl_synwrite_qf"))
  let b:perl_synwrite_qf = perl_synwrite_qf
elseif !exists("b:perl_synwrite_qf")
  let b:perl_synwrite_qf = 0
endif

"" execute the given do_command if the buffer is syntactically correct perl
"" -- or if do_anyway is true
function! s:PerlSynDo(do_anyway,do_command)
  let command = "!perl -c"

  if (b:perl_synwrite_qf)
```


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

```
    " this env var tells Vi::QuickFix to replace "-" with actual filename
    let $VI_QUICKFIX_SOURCEFILE=expand("%")
    let command = command . " -MVi::QuickFix"
  endif

  " respect taint checking
  if (match(getline(1), "^#!.\\\\+perl.\\\\+-T") = = 0)
    let command = command . " -T"
  endif

  exec "write" command

  silent! cgetfile " try to read the error file
  if !v:shell_error || a:do_anyway
    exec a:do_command
    set nomod
  endif
endfunction

"" set up the autocommand, if b:perl_synwrite_au is true
if (b:perl_synwrite_au > 0)
  let b:undo_ftplugin = "au! perl_synwrite * " . expand("%")

  augroup perl_synwrite
    exec "au BufWriteCmd,FileWriteCmd " . expand("%") .
         " call s:PerlSynDo(0,\\"write <afile>\\")"
  augroup END
endif

"" the :Write command
command -buffer -nargs=* -complete=file -range=% -bang Write call \\
    s:PerlSynDo("<bang>"= ="!","<line1>,<line2>write<bang> <args>")
```

Running the Hack

When you edit a Perl file, use `:W` instead of `:w` to write the file. If the file fails to compile with `perl -c`, Vim will refuse to write the file to disk. You can always fall back to `:w`, or use `:W!` to check, but write out the file even if it has bad syntax.

Hacking the Hack

The plug-in has two configurable options that you can set in your *.vimrc*. The first is `perl_synwrite_au`, which hooks the `:W` command's logic onto an autocommand that fires when you use `:w`. This will let you use `:w` for any sort of file, but still enjoy the syntax error catching of the plug-in. It's a little twitchy, though, when you start passing arguments to `:w`, so you're probably best off just using `:W`.

The second is `perl_synwrite_qf`, which lets the plug-in use the `Vi::QuickFix` module to parse `perl`'s errors into a format that Vim can use to jump to problems. With this option set, `perl` will write errors to *error.err*, which Vim will read when you use its QuickFix commands. `:help quickfix` lists all of the commands, but the two most useful are `:cf` to jump to the first syntax error and `:copen` to open a new window listing all your errors. In that new window, you can move to the error that interests you, hit Enter, and jump to the error in your buffer.

`Vi::QuickFix` relies on tying the standard error stream, which isn't possible in Perl 5.6, so if you use *perl_synwrite.vim* in more than one development environment, you might want to set the `perl_synwrite_qf` option dynamically:

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

```
silent call system("perl -e0 -MVi::QuickFix")
let perl_synwrite_qf = ! v:shell_error
```

In other words, if Perl can't `use` the `Vi::QuickFix` module, don't try using it for the plug-in.

By default, Vim thinks that **.t* files are Tads, or maybe Nroff, files. It's easy to fix; create a file in *~/.vim/ftdetect* containing:

```
au BufRead,BufNewFile *.t
                set ft=perl
```

Now when you edit `00-load.t`, Vim will know it's Perl and not your latest interactive fiction masterpiece.

Emacs users, you can use a minor mode to run a Perl syntax check before saving the file. Whenever `perl -c` fails, Emacs will not save your file. To save files anyway, toggle the mode off with `M-x perl-syntax-mode`. See "Enforce Local Style" [Hack #7] for a related tip on automatically tidying your code when saving.

```
(defvar perl-syntax-bin "perl"
    "The perl binary used to check syntax.")
  (defun perl-syntax-check-only ()
    "Returns a either nil or t depending on whether \\
     the current buffer passes perl's syntax check."
    (interactive)
    (let ((buf (get-buffer-create "*Perl syntax check*")))
      (let ((syntax-ok (= 0 (save-excursion
                              (widen)
                              (call-process-region
                               (point-min) (point-max) perl-syntax-bin nil buf nil "-c"))) ))
        (if syntax-ok (kill-buffer buf)
          (display-buffer buf))
        syntax-ok)))
  (defvar perl-syntax-mode nil
    "Check perl syntax before saving.")
  (make-variable-buffer-local 'perl-syntax-mode)
  (defun perl-syntax-write-hook ()
    "Check perl syntax during 'write-file-hooks' for 'perl-syntax-mode'"
    (if perl-syntax-mode
        (save-excursion
          (widen)
          (mark-whole-buffer)
          (not (perl-syntax-check-only)))
      nil))
  (defun perl-syntax-mode (&optional arg)
    "Perl syntax checking minor mode."
    (interactive "P")
    (setq perl-syntax-mode
          (if (null arg)
              (not perl-syntax-mode)
            (> (prefix-numeric-value arg) 0)))
    (make-local-hook 'write-file-hooks)
    (if perl-syntax-mode
        (add-hook 'write-file-hooks 'perl-syntax-write-hook)
```


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

```
    (remove-hook 'write-file-hooks 'perl-syntax-write-hook)))
(if (not (assq 'perl-syntax-mode minor-mode-alist))
    (setq minor-mode-alist
          (cons '(perl-syntax-mode " Perl Syntax")
                minor-mode-alist)))
(eval-after-load "cperl-mode"
  '(add-hook 'cperl-mode-hook 'perl-syntax-mode))
```

Hack 9. Automate Checkin Code Reviews

Let `Perl::Tidy` be your first code review—on every Subversion checkin!

In a multideveloper project, relying on developers to follow the coding standards without fail and to run `perltidy` against all of their code ("Enforce Local Style" [Hack #7]) before every checkin is unrealistic, especially because this is tedious work. Fortunately, this is an automatable process. If you use Subversion (or Svk), it's easy to write a hook that checks code for tidiness, however you define it.

The Hack

For various reasons, it's not possible to manipulate the committed files with a pre-commit hook in Subversion. That's why this is a hack.

Within your Subversion repository, copy the *hooks/post-commit.tmpl* file to *hooks/post-commit*—unless you already have the file. Remove all code that runs other commands (again, unless you're already using it). Add a single line:

```
perl /usr/local/bin/check_tidy_file.pl "$REPOS" "$REV"
```

Adjust the file path appropriately. Make the *hooks/post-commit* file executable with `chmod +x` on Unix.

Finally, save the *check_tidy_file.pl* program to the path you used in the file. The program is:

```
#!/usr/bin/perl

use strict;
use warnings;

use Perl::Tidy;

use File::Temp;
use File::Spec::Functions;
```


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

```
my $svnlook      = '/usr/bin/svnlook';
my $diff         = '/usr/bin/diff -u';

# eat the arguments so as not to confuse Perl::Tidy
my ($repo, $rev) = @ARGV;
@ARGV            = ( );

my @diffs;

for my $changed_file (get_changed_perl_files( $repo, $rev ))
{
    my $source = get_revision( $repo, $rev, $changed_file );
    Perl::Tidy::perltidy( source => \\$source, destination => \\(my $dest) );
    push @diffs, get_diff( $changed_file, $source, $dest );
}

report_diffs( @diffs );

sub get_changed_perl_files
{
    my ($repo, $rev) = @_;

    my @files;

    for my $change (\Q$svnlook changed $repo -r $rev\Q)
    {
        my ($status, $file) =  split( /\\s+/, $change );
        next unless $file   =~ /\\.p[lm]\\z/;
        push @files, $file;
    }

    return @files;
}

sub get_revision
{
    my ($repo, $rev, $file) = @_;
    return scalar \Q$svnlook cat $repo -r $rev $file\Q;
}

sub get_diff
{
    my $filename        = shift;
    return if $_[0] eq $_[1];

    my $dir   = File::Temp::tempdir( );
    my @files = map { catdir( $dir, $filename . $_ ) } qw( .orig .tidy );

    for my $file (@files)
    {
        open( my $out, '>', $file ) or die "Couldn't write $file: $!\\n";
        print $out shift;
        close $out;
    }

    return scalar \Q$diff @files\Q;
}

sub report_diffs
{
```


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

```
    for my $diff (@_)
    {
        warn "Error:\\n$diff\\n";
    }
}
```

When Subversion finishes committing a checkin to the repository, it calls the *hooks/post-commit* script, which itself launches other programs, passing the repository path and the number of the just-committed revision. This program uses the `svnlook` command to find the modified files, skipping everything that's not a Perl program or module (files ending in *.pl* or *.pm*).

For each of these files, it grabs the entire contents from the just-completed revision and runs it through `Perl::Tidy` (the actual engine of the `perltidy` utility). If the resulting file is the same as the revision, everything is fine. Otherwise, it runs a `diff` utility to see the changes necessary to make the file tidy. From there, `report_diffs( )` receives a list of these differences.

Hacking the Hack

As it is now, the program is only useful when run directly with the path to the repository and a revision number. It *could* instead write the differences to a file, automatically check in the revised versions in a new checkin, or e-mail the diffs to a list of programmers.

To use a *.perltidyrc* file with the tidier program, add the `perltidy => $rcfile_path` arguments to the `perltidy( )` call, where `$rcfile_path` contains the path to the *.perltidyrc* file to use.

Hack 10. Run Tests from Within Vim

Run your tests from your editor.

One of the nice things about Perl is the "tweak, run, tweak, run" development cycle. There's no separate compile phase to slow you down. However, you likely find yourself frequently writing tests and madly switching back and forth between the tests and the code. When you run the tests, you may exit the editor or type something like `!perl -Ilib/ t/test_program.t` in vi's command mode. This breaks the "tweak, run" rhythm.

The Hack

Perl programmers don't like to slow things down. Instead, consider binding keys in your editor to the chicken-bone voodoo you use to run your test suite.

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

Binding keys

By running the tests from within the editor, you no longer have to remember how to execute the tests or edit the editor. Just tweak and run. Add the following line to your *.vimrc* file to run the currently edited test file by typing `,t` (comma, tee):

```
map ,t  <Esc>:!prove -vl %<CR>
```

This technique uses the `prove` program to run your tests. `prove` is a handy little program distributed with and designed to run your tests through `Test::Harness`. The switches are `v` (vee), which tells `prove` to run in "verbose" mode and show all of the test output, and `l` (ell), which tells `prove` to add *lib/* to `@INC`.

If *lib/* is not where you typically do your development, use the `I` switch to add a different path to `@INC`.

```
map ,t  <Esc>:!prove -Iwork/ -v %<CR>
```

Seeing failures

If it's a long test and you get a few failures, it can be difficult to see where the failures were. If that's the case, use `,T` (comma capital tee) to pipe the results through your favorite pager.

```
map ,T  <Esc>:!prove -lv % \\| less<CR>
```

Of course, make sure your editor does not have those keys already mapped to something else. This hack does not recommend breaking existing mappings in your editor.

Managing paths

These techniques do tend to require that you edit your tests in the "standard" way. If you have your tests organized in subdirectories, switching to the *t/customer/* directory and editing *save.t* may cause problems when trying to tell `prove` which directories to use. If you habitually do this, don't tell `prove` which paths to add to `@INC`.

```
map ,t  <Esc>:!prove -v %<CR>
```

Instead, have your *tests* add paths to `@INC`:

```
use lib '../../lib';
```

That can get a bit clumsy and it can make it rather tough to reorganize your tests, but it works.

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

Here's a little alisp for Emacs users to put into your *~/.emacs* file to get the same thing. It binds to `C-c t`, but you can change to whatever you prefer:

```
(eval-after-load "cperl-mode"
    '(add-hook 'cperl-mode-hook
       (lambda () (local-set-key "\\C-ct" 'cperl-prove))))
  (defun cperl-prove ()
    "Run the current test."
    (interactive)
    (shell-command (concat "prove -v "
      (shell-quote-argument (buffer-file-name)))))
```

Hack 11. Run Perl from Emacs

Make Perl and Elisp play nicely together.

Emacs's long and varied history happens to embody much of Perl's "There's More Than One Way To Do It" approach to things. This is especially evident when you run a small bit of Perl code from within Emacs. Here's how to do just that.

The Hack

Suppose you really need to know the higher bits of the current value of `time( )`. In Perl, that's `print time() 8>>;`. You could use the `shell-command` command (normally on Control-Alt-One), and enter:

```
perl -e 'print time( ) >> 8;'
```

Emacs will dutifully run that command line and then show the output. Note though that you have to remember to quote and/or backslash-escape the Perl expression according to the rules of your default shell. This quickly becomes maddening if the expression itself contains quotes and/or backslashes or even is several lines long.

An alternative is to start an "Emacs shell" in an Emacs subwindow, then start the Perl debugger in that shell. That is, type **`alt-x " shell "`** Enter, and then **`perl -de1`** Enter, and then enter the expression just as if you were running the debugger in a normal terminal window:

```
% perl -de1

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):    1
  DB<1> p time( ) >> 8
4448317
  DB<2>
```

This means you don't have to escape the Perl expression as you would if you were sending it through a command line, but it does require you to know at least a bit about the Perl debugger and the Emacs shell. It also becomes troublesome in its own way when your expression is several lines long.

A simpler alternative is to save your snippet to a file named *delme123.pl* and to run that via a command line, but this is a very effective way to fill every directory in reach with files named with the same variant of *delme*.

I prefer defining a new function just for running Perl code in the Region (what you have selected in Emacs, between the Point and the Mark):

```
(defun perl-eval (beg end)
  "Run selected region as Perl code"
  (interactive "r")
  (shell-command-on-region beg end "perl")
   ; feeds the region to perl on STDIN
)
```

I bind it to my CTRL-Alt-p key:

```
(global-set-key "\\M-\\C-p" 'perl-eval)
```

Then when I want to run some Perl expression in whatever buffer I happen to be in, I just set the mark, type the expression, and hit CTRL-Alt-p. It requires no special escaping, nor are there any problems when the Perl code spans several lines.

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