NAME

perlhack - How to hack on Perl

DESCRIPTION

This document explains how Perl development works. It includes details about the Perl 5 Porters email list, the Perl repository, the Perlbug bug tracker, patch guidelines, and commentary on Perl development philosophy.

SUPER QUICK PATCH GUIDE

If you just want to submit a single small patch like a pod fix, a test for a bug, comment fixes, etc., it's easy! Here's how:

* Check out the source repository

  The perl source is in a git repository. You can clone the repository with the following command:

  ```
  % git clone git://perl5.git.perl.org/perl.git perl
  ```

* Ensure you're following the latest advice

  In case the advice in this guide has been updated recently, read the latest version directly from the perl source:

  ```
  % perldoc pod/perlhack.pod
  ```

* Make your change

  Hack, hack, hack. Keep in mind that Perl runs on many different platforms, with different operating systems that have different capabilities, different filesystem organizations, and even different character sets. *perlhacktips* gives advice on this.

* Test your change

  You can run all the tests with the following commands:

  ```
  % ./Configure -des -Dusedevel
  % make test
  ```

  Keep hacking until the tests pass.

* Commit your change

  Committing your work will save the change *on your local system*:

  ```
  % git commit -a -m 'Commit message goes here'
  ```

  Make sure the commit message describes your change in a single sentence. For example, "Fixed spelling errors in perlhack.pod".

* Send your change to perlbug

  The next step is to submit your patch to the Perl core ticket system via email.

  If your changes are in a single git commit, run the following commands to generate the patch file and attach it to your bug report:

  ```
  % git format-patch -1
  % ./perl -Ilib utils/perlbug -p 0001-*.patch
  ```

  The perlbug program will ask you a few questions about your email address and the patch you're submitting. Once you've answered them it will submit your patch via email.

  If your changes are in multiple commits, generate a patch file for each one and provide them to perlbug's `-p` option separated by commas:

```
% git format-patch -3
% ./perl -Ilib utils/perlbug -p 0001-fix1.patch,0002-fix2.patch,\
> 0003-fix3.patch
```

When prompted, pick a subject that summarizes your changes.

* Thank you

The porters appreciate the time you spent helping to make Perl better. Thank you!

* Next time

The next time you wish to make a patch, you need to start from the latest perl in a pristine state. Check you don't have any local changes or added files in your perl check-out which you wish to keep, then run these commands:

```
% git pull
% git reset --hard origin/blead
% git clean -dxf
```

BUG REPORTING

If you want to report a bug in Perl, you must use the *perlbug* command line tool. This tool will ensure that your bug report includes all the relevant system and configuration information.

To browse existing Perl bugs and patches, you can use the web interface at *http://rt.perl.org/*.

Please check the archive of the perl5-porters list (see below) and/or the bug tracking system before submitting a bug report. Often, you'll find that the bug has been reported already.

You can log in to the bug tracking system and comment on existing bug reports. If you have additional information regarding an existing bug, please add it. This will help the porters fix the bug.

PERL 5 PORTERS

The perl5-porters (p5p) mailing list is where the Perl standard distribution is maintained and developed. The people who maintain Perl are also referred to as the "Perl 5 Porters", "p5p" or just the "porters".

A searchable archive of the list is available at *http://markmail.org/search/?q=perl5-porters*. There is also an archive at *http://archive.develooper.com/perl5-porters@perl.org/*.

perl-changes mailing list

The perl5-changes mailing list receives a copy of each patch that gets submitted to the maintenance and development branches of the perl repository. See *http://lists.perl.org/list/perl5-changes.html* for subscription and archive information.

#p5p on IRC

Many porters are also active on the *irc://irc.perl.org/#p5p* channel. Feel free to join the channel and ask questions about hacking on the Perl core.

GETTING THE PERL SOURCE

All of Perl's source code is kept centrally in a Git repository at *perl5.git.perl.org*. The repository contains many Perl revisions from Perl 1 onwards and all the revisions from Perforce, the previous version control system.

For much more detail on using git with the Perl repository, please see *perlgit*.

Read access via Git

You will need a copy of Git for your computer. You can fetch a copy of the repository using the git protocol:

```
% git clone git://perl5.git.perl.org/perl.git perl
```

This clones the repository and makes a local copy in the *perl* directory.

If you cannot use the git protocol for firewall reasons, you can also clone via http, though this is much slower:

```
% git clone http://perl5.git.perl.org/perl.git perl
```

Read access via the web

You may access the repository over the web. This allows you to browse the tree, see recent commits, subscribe to RSS feeds for the changes, search for particular commits and more. You may access it at *http://perl5.git.perl.org/perl.git*. A mirror of the repository is found at *https://github.com/Perl/perl5*.

Read access via rsync

You can also choose to use rsync to get a copy of the current source tree for the bleadperl branch and all maintenance branches:

```
% rsync -avz rsync://perl5.git.perl.org/perl-current .
% rsync -avz rsync://perl5.git.perl.org/perl-5.12.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.10.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.8.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.6.x .
% rsync -avz rsync://perl5.git.perl.org/perl-5.005xx .
```

(Add the `--delete` option to remove leftover files.)

To get a full list of the available sync points:

```
% rsync perl5.git.perl.org::
```

Write access via git

If you have a commit bit, please see *perlgit* for more details on using git.

PATCHING PERL

If you're planning to do more extensive work than a single small fix, we encourage you to read the documentation below. This will help you focus your work and make your patches easier to incorporate into the Perl source.

Submitting patches

If you have a small patch to submit, please submit it via perlbug. You can also send email directly to perlbug@perl.org. Please note that messages sent to perlbug may be held in a moderation queue, so you won't receive a response immediately.

You'll know your submission has been processed when you receive an email from our ticket tracking system. This email will give you a ticket number. Once your patch has made it to the ticket tracking system, it will also be sent to the perl5-porters@perl.org list.

Patches are reviewed and discussed on the p5p list. Simple, uncontroversial patches will usually be applied without any discussion. When the patch is applied, the ticket will be updated and you will receive email. In addition, an email will be sent to the p5p list.

In other cases, the patch will need more work or discussion. That will happen on the p5p list.

You are encouraged to participate in the discussion and advocate for your patch. Sometimes your patch may get lost in the shuffle. It's appropriate to send a reminder email to p5p if no action has been taken in a month. Please remember that the Perl 5 developers are all volunteers, and be polite.

Changes are always applied directly to the main development branch, called "blead". Some patches may be backported to a maintenance branch. If you think your patch is appropriate for the maintenance branch (see *"MAINTENANCE BRANCHES" in perlpolicy*), please explain why when you submit it.

Getting your patch accepted

If you are submitting a code patch there are several things that you can do to help the Perl 5 Porters accept your patch.

Patch style

If you used git to check out the Perl source, then using `git format-patch` will produce a patch in a style suitable for Perl. The `format-patch` command produces one patch file for each commit you made. If you prefer to send a single patch for all commits, you can use `git diff`.

```
% git checkout blead
% git pull
% git diff blead my-branch-name
```

This produces a patch based on the difference between blead and your current branch. It's important to make sure that blead is up to date before producing the diff, that's why we call `git pull` first.

We strongly recommend that you use git if possible. It will make your life easier, and ours as well.

However, if you're not using git, you can still produce a suitable patch. You'll need a pristine copy of the Perl source to diff against. The porters prefer unified diffs. Using GNU `diff`, you can produce a diff like this:

```
% diff -Npurd perl.pristine perl.mine
```

Make sure that you `make realclean` in your copy of Perl to remove any build artifacts, or you may get a confusing result.

Commit message

As you craft each patch you intend to submit to the Perl core, it's important to write a good commit message. This is especially important if your submission will consist of a series of commits.

The first line of the commit message should be a short description without a period. It should be no longer than the subject line of an email, 50 characters being a good rule of thumb.

A lot of Git tools (Gitweb, GitHub, git log --pretty=oneline, ...) will only display the first line (cut off at 50 characters) when presenting commit summaries.

The commit message should include a description of the problem that the patch corrects or new functionality that the patch adds.

As a general rule of thumb, your commit message should help a programmer who knows the Perl core quickly understand what you were trying to do, how you were trying to do it, and why the change matters to Perl.

* Why

  Your commit message should describe why the change you are making is important. When someone looks at your change in six months or six years, your intent should be clear.

  If you're deprecating a feature with the intent of later simplifying another bit of code, say so. If you're fixing a performance problem or adding a new feature to support some other bit of the core, mention that.

* What

  Your commit message should describe what part of the Perl core you're changing and what

you expect your patch to do.

* How

While it's not necessary for documentation changes, new tests or trivial patches, it's often worth explaining how your change works. Even if it's clear to you today, it may not be clear to a porter next month or next year.

A commit message isn't intended to take the place of comments in your code. Commit messages should describe the change you made, while code comments should describe the current state of the code.

If you've just implemented a new feature, complete with doc, tests and well-commented code, a brief commit message will often suffice. If, however, you've just changed a single character deep in the parser or lexer, you might need to write a small novel to ensure that future readers understand what you did and why you did it.

Comments, Comments, Comments

Be sure to adequately comment your code. While commenting every line is unnecessary, anything that takes advantage of side effects of operators, that creates changes that will be felt outside of the function being patched, or that others may find confusing should be documented. If you are going to err, it is better to err on the side of adding too many comments than too few.

The best comments explain *why* the code does what it does, not *what it does*.

Style

In general, please follow the particular style of the code you are patching.

In particular, follow these general guidelines for patching Perl sources:

- 4-wide indents for code, 2-wide indents for nested CPP `#defines`, with 8-wide tabstops.
- Use spaces for indentation, not tab characters.

  The codebase is a mixture of tabs and spaces for indentation, and we are moving to spaces only. Converting lines you're patching from 8-wide tabs to spaces will help this migration.
- Try hard not to exceed 79-columns
- ANSI C prototypes
- Uncuddled elses and "K&R" style for indenting control constructs
- No C++ style (//) comments
- Mark places that need to be revisited with XXX (and revisit often!)
- Opening brace lines up with "if" when conditional spans multiple lines; should be at end-of-line otherwise
- In function definitions, name starts in column 0 (return value-type is on previous line)
- Single space after keywords that are followed by parens, no space between function name and following paren
- Avoid assignments in conditionals, but if they're unavoidable, use extra paren, e.g. "if (a && (b = c)) ..."
- "return foo;" rather than "return(foo);"
- "if (!foo) ..." rather than "if (foo == FALSE) ..." etc.
- Do not declare variables using "register". It may be counterproductive with modern compilers, and is deprecated in C++, under which the Perl source is regularly compiled.

- In-line functions that are in headers that are accessible to XS code need to be able to compile without warnings with commonly used extra compilation flags, such as gcc's `-Wswitch-default` which warns whenever a switch statement does not have a "default" case. The use of these extra flags is to catch potential problems in legal C code, and is often used by Perl aggregators, such as Linux distributors.

Test suite

If your patch changes code (rather than just changing documentation), you should also include one or more test cases which illustrate the bug you're fixing or validate the new functionality you're adding. In general, you should update an existing test file rather than create a new one.

Your test suite additions should generally follow these guidelines (courtesy of Gurusamy Sarathy <gsar@activestate.com>):

- Know what you're testing. Read the docs, and the source.
- Tend to fail, not succeed.
- Interpret results strictly.
- Use unrelated features (this will flush out bizarre interactions).
- Use non-standard idioms (otherwise you are not testing TIMTOWTDI).
- Avoid using hardcoded test numbers whenever possible (the EXPECTED/GOT found in t/op/tie.t is much more maintainable, and gives better failure reports).
- Give meaningful error messages when a test fails.
- Avoid using qx// and system() unless you are testing for them. If you do use them, make sure that you cover _all_ perl platforms.
- Unlink any temporary files you create.
- Promote unforeseen warnings to errors with $SIG{__WARN__}.
- Be sure to use the libraries and modules shipped with the version being tested, not those that were already installed.
- Add comments to the code explaining what you are testing for.
- Make updating the '1..42' string unnecessary. Or make sure that you update it.
- Test _all_ behaviors of a given operator, library, or function.

  Test all optional arguments.

  Test return values in various contexts (boolean, scalar, list, lvalue).

  Use both global and lexical variables.

  Don't forget the exceptional, pathological cases.

Patching a core module

This works just like patching anything else, with one extra consideration.

Modules in the *cpan/* directory of the source tree are maintained outside of the Perl core. When the author updates the module, the updates are simply copied into the core. See that module's documentation or its listing on *http://search.cpan.org/* for more information on reporting bugs and submitting patches.

In most cases, patches to modules in *cpan/* should be sent upstream and should not be applied to the Perl core individually. If a patch to a file in *cpan/* absolutely cannot wait for the fix to be made upstream, released to CPAN and copied to blead, you must add (or update) a `CUSTOMIZED` entry in

the *"Porting/Maintainers.pl"* file to flag that a local modification has been made. See *"Porting/Maintainers.pl"* for more details.

In contrast, modules in the *dist/* directory are maintained in the core.

Updating perldelta

For changes significant enough to warrant a *pod/perldelta.pod* entry, the porters will greatly appreciate it if you submit a delta entry along with your actual change. Significant changes include, but are not limited to:

- Adding, deprecating, or removing core features
- Adding, deprecating, removing, or upgrading core or dual-life modules
- Adding new core tests
- Fixing security issues and user-visible bugs in the core
- Changes that might break existing code, either on the perl or C level
- Significant performance improvements
- Adding, removing, or significantly changing documentation in the *pod/* directory
- Important platform-specific changes

Please make sure you add the perldelta entry to the right section within *pod/perldelta.pod*. More information on how to write good perldelta entries is available in the `Style` section of *Porting/how_to_write_a_perldelta.pod*.

What makes for a good patch?

New features and extensions to the language can be contentious. There is no specific set of criteria which determine what features get added, but here are some questions to consider when developing a patch:

Does the concept match the general goals of Perl?

Our goals include, but are not limited to:

1. Keep it fast, simple, and useful.
2. Keep features/concepts as orthogonal as possible.
3. No arbitrary limits (platforms, data sizes, cultures).
4. Keep it open and exciting to use/patch/advocate Perl everywhere.
5. Either assimilate new technologies, or build bridges to them.

Where is the implementation?

All the talk in the world is useless without an implementation. In almost every case, the person or people who argue for a new feature will be expected to be the ones who implement it. Porters capable of coding new features have their own agendas, and are not available to implement your (possibly good) idea.

Backwards compatibility

It's a cardinal sin to break existing Perl programs. New warnings can be contentious--some say that a program that emits warnings is not broken, while others say it is. Adding keywords has the potential to break programs, changing the meaning of existing token sequences or functions might break programs.

The Perl 5 core includes mechanisms to help porters make backwards incompatible changes more

compatible such as the *feature* and *deprecate* modules. Please use them when appropriate.

Could it be a module instead?

Perl 5 has extension mechanisms, modules and XS, specifically to avoid the need to keep changing the Perl interpreter. You can write modules that export functions, you can give those functions prototypes so they can be called like built-in functions, you can even write XS code to mess with the runtime data structures of the Perl interpreter if you want to implement really complicated things.

Whenever possible, new features should be prototyped in a CPAN module before they will be considered for the core.

Is the feature generic enough?

Is this something that only the submitter wants added to the language, or is it broadly useful? Sometimes, instead of adding a feature with a tight focus, the porters might decide to wait until someone implements the more generalized feature.

Does it potentially introduce new bugs?

Radical rewrites of large chunks of the Perl interpreter have the potential to introduce new bugs.

How big is it?

The smaller and more localized the change, the better. Similarly, a series of small patches is greatly preferred over a single large patch.

Does it preclude other desirable features?

A patch is likely to be rejected if it closes off future avenues of development. For instance, a patch that placed a true and final interpretation on prototypes is likely to be rejected because there are still options for the future of prototypes that haven't been addressed.

Is the implementation robust?

Good patches (tight code, complete, correct) stand more chance of going in. Sloppy or incorrect patches might be placed on the back burner until the pumpking has time to fix, or might be discarded altogether without further notice.

Is the implementation generic enough to be portable?

The worst patches make use of system-specific features. It's highly unlikely that non-portable additions to the Perl language will be accepted.

Is the implementation tested?

Patches which change behaviour (fixing bugs or introducing new features) must include regression tests to verify that everything works as expected.

Without tests provided by the original author, how can anyone else changing perl in the future be sure that they haven't unwittingly broken the behaviour the patch implements? And without tests, how can the patch's author be confident that his/her hard work put into the patch won't be accidentally thrown away by someone in the future?

Is there enough documentation?

Patches without documentation are probably ill-thought out or incomplete. No features can be added or changed without documentation, so submitting a patch for the appropriate pod docs as well as the source code is important.

Is there another way to do it?

Larry said "Although the Perl Slogan is *There's More Than One Way to Do It*, I hesitate to make 10 ways to do something". This is a tricky heuristic to navigate, though--one man's essential addition is another man's pointless cruft.

Does it create too much work?

Work for the pumpking, work for Perl programmers, work for module authors, ... Perl is supposed to be easy.

Patches speak louder than words

Working code is always preferred to pie-in-the-sky ideas. A patch to add a feature stands a much higher chance of making it to the language than does a random feature request, no matter how fervently argued the request might be. This ties into "Will it be useful?", as the fact that someone took the time to make the patch demonstrates a strong desire for the feature.

TESTING

The core uses the same testing style as the rest of Perl, a simple "ok/not ok" run through Test::Harness, but there are a few special considerations.

There are three ways to write a test in the core: *Test::More*, *t/test.pl* and ad hoc `print $test ? "ok 42\n" : "not ok 42\n"`. The decision of which to use depends on what part of the test suite you're working on. This is a measure to prevent a high-level failure (such as Config.pm breaking) from causing basic functionality tests to fail.

The *t/test.pl* library provides some of the features of *Test::More*, but avoids loading most modules and uses as few core features as possible.

If you write your own test, use the *Test Anything Protocol*.

* *t/base*, *t/comp* and *t/opbasic*

  Since we don't know if `require` works, or even subroutines, use ad hoc tests for these three. Step carefully to avoid using the feature being tested. Tests in *t/opbasic*, for instance, have been placed there rather than in *t/op* because they test functionality which *t/test.pl* presumes has already been demonstrated to work.

* *t/cmd*, *t/run*, *t/io* and *t/op*

  Now that basic require() and subroutines are tested, you can use the *t/test.pl* library.

  You can also use certain libraries like Config conditionally, but be sure to skip the test gracefully if it's not there.

* Everything else

  Now that the core of Perl is tested, *Test::More* can and should be used. You can also use the full suite of core modules in the tests.

When you say "make test", Perl uses the *t/TEST* program to run the test suite (except under Win32 where it uses *t/harness* instead). All tests are run from the *t/* directory, **not** the directory which contains the test. This causes some problems with the tests in *lib/*, so here's some opportunity for some patching.

You must be triply conscious of cross-platform concerns. This usually boils down to using *File::Spec*, avoiding things like `fork()` and `system()` unless absolutely necessary, and not assuming that a given character has a particular ordinal value (code point) or that its UTF-8 representation is composed of particular bytes.

There are several functions available to specify characters and code points portably in tests. The always-preloaded functions `utf8::unicode_to_native()` and its inverse `utf8::native_to_unicode()` take code points and translate appropriately. The file *t/charset_tools.pl* has several functions that can be useful. It has versions of the previous two functions that take strings as inputs -- not single numeric code points: `uni_to_native()` and `native_to_uni()`. If you must look at the individual bytes comprising a UTF-8 encoded string, `byte_utf8a_to_utf8n()` takes as input a string of those bytes encoded for an ASCII platform, and returns the equivalent string in the native platform. For example,

`byte_utf8a_to_utf8n("\xC2\xA0")` returns the byte sequence on the current platform that form the UTF-8 for `U+00A0`, since `"\xC2\xA0"` are the UTF-8 bytes on an ASCII platform for that code point. This function returns `"\xC2\xA0"` on an ASCII platform, and `"\x80\x41"` on an EBCDIC 1047 one.

But easiest is, if the character is specifiable as a literal, like `"A"` or `"%"`, to use that; if not so specifiable, you can use use `\N{}` , if the side effects aren't troublesome. Simply specify all your characters in hex, using `\N{U+ZZ}` instead of `\xZZ`. `\N{}` is the Unicode name, and so it always gives you the Unicode character. `\N{U+41}` is the character whose Unicode code point is `0x41`, hence is `'A'` on all platforms. The side effects are:

- These select Unicode rules. That means that in double-quotish strings, the string is always converted to UTF-8 to force a Unicode interpretation (you can `utf8::downgrade()` afterwards to convert back to non-UTF8, if possible). In regular expression patterns, the conversion isn't done, but if the character set modifier would otherwise be `/d`, it is changed to `/u`.
- If you use the form `\N{`*`character name`*`}`, the *charnames* module gets automatically loaded. This may not be suitable for the test level you are doing.

If you are testing locales (see *perllocale*), there are helper functions in *t/loc_tools.pl* to enable you to see what locales there are on the current platform.

Special make test targets

There are various special make targets that can be used to test Perl slightly differently than the standard "test" target. Not all them are expected to give a 100% success rate. Many of them have several aliases, and many of them are not available on certain operating systems.

* test_porting

  This runs some basic sanity tests on the source tree and helps catch basic errors before you submit a patch.

* minitest

  Run *miniperl* on *t/base*, *t/comp*, *t/cmd*, *t/run*, *t/io*, *t/op*, *t/uni* and *t/mro* tests.

* test.valgrind check.valgrind

  (Only in Linux) Run all the tests using the memory leak + naughty memory access tool "valgrind". The log files will be named *testname.valgrind*.

* test_harness

  Run the test suite with the *t/harness* controlling program, instead of *t/TEST*. *t/harness* is more sophisticated, and uses the *Test::Harness* module, thus using this test target supposes that perl mostly works. The main advantage for our purposes is that it prints a detailed summary of failed tests at the end. Also, unlike *t/TEST*, it doesn't redirect stderr to stdout.

  Note that under Win32 *t/harness* is always used instead of *t/TEST*, so there is no special "test_harness" target.

  Under Win32's "test" target you may use the TEST_SWITCHES and TEST_FILES environment variables to control the behaviour of *t/harness*. This means you can say

```
nmake test TEST_FILES="op/*.t"
nmake test TEST_SWITCHES="-torture" TEST_FILES="op/*.t"
```

* test-notty test_notty

  Sets PERL_SKIP_TTY_TEST to true before running normal test.

Parallel tests

The core distribution can now run its regression tests in parallel on Unix-like platforms. Instead of running `make test`, set `TEST_JOBS` in your environment to the number of tests to run in parallel, and run `make test_harness`. On a Bourne-like shell, this can be done as

```
TEST_JOBS=3 make test_harness  # Run 3 tests in parallel
```

An environment variable is used, rather than parallel make itself, because *TAP::Harness* needs to be able to schedule individual non-conflicting test scripts itself, and there is no standard interface to `make` utilities to interact with their job schedulers.

Note that currently some test scripts may fail when run in parallel (most notably *dist/IO/t/io_dir.t*). If necessary, run just the failing scripts again sequentially and see if the failures go away.

Running tests by hand

You can run part of the test suite by hand by using one of the following commands from the *t/* directory:

```
./perl -I../lib TEST list-of-.t-files
```

or

```
./perl -I../lib harness list-of-.t-files
```

(If you don't specify test scripts, the whole test suite will be run.)

Using t/harness for testing

If you use `harness` for testing, you have several command line options available to you. The arguments are as follows, and are in the order that they must appear if used together.

```
harness -v -torture -re=pattern LIST OF FILES TO TEST
harness -v -torture -re LIST OF PATTERNS TO MATCH
```

If `LIST OF FILES TO TEST` is omitted, the file list is obtained from the manifest. The file list may include shell wildcards which will be expanded out.

* -v

  Run the tests under verbose mode so you can see what tests were run, and debug output.

* -torture

  Run the torture tests as well as the normal set.

* -re=PATTERN

  Filter the file list so that all the test files run match PATTERN. Note that this form is distinct from the **-re LIST OF PATTERNS** form below in that it allows the file list to be provided as well.

* -re LIST OF PATTERNS

  Filter the file list so that all the test files run match /(LIST|OF|PATTERNS)/. Note that with this form the patterns are joined by '|' and you cannot supply a list of files, instead the test files are obtained from the MANIFEST.

You can run an individual test by a command similar to

```
./perl -I../lib path/to/foo.t
```

except that the harnesses set up some environment variables that may affect the execution of the test:

* PERL_CORE=1

  indicates that we're running this test as part of the perl core test suite. This is useful for modules that have a dual life on CPAN.

* PERL_DESTRUCT_LEVEL=2

  is set to 2 if it isn't set already (see *"PERL_DESTRUCT_LEVEL" in perlhacktips*).

* PERL

  (used only by *t/TEST*) if set, overrides the path to the perl executable that should be used to run the tests (the default being *./perl*).

* PERL_SKIP_TTY_TEST

  if set, tells to skip the tests that need a terminal. It's actually set automatically by the Makefile, but can also be forced artificially by running 'make test_notty'.

Other environment variables that may influence tests

* PERL_TEST_Net_Ping

  Setting this variable runs all the Net::Ping modules tests, otherwise some tests that interact with the outside world are skipped. See *perl58delta*.

* PERL_TEST_NOVREXX

  Setting this variable skips the vrexx.t tests for OS2::REXX.

* PERL_TEST_NUMCONVERTS

  This sets a variable in op/numconvert.t.

* PERL_TEST_MEMORY

  Setting this variable includes the tests in *t/bigmem/*. This should be set to the number of gigabytes of memory available for testing, eg. `PERL_TEST_MEMORY=4` indicates that tests that require 4GiB of available memory can be run safely.

See also the documentation for the Test and Test::Harness modules, for more environment variables that affect testing.

Performance testing

The file *t/perf/benchmarks* contains snippets of perl code which are intended to be benchmarked across a range of perls by the *Porting/bench.pl* tool. If you fix or enhance a performance issue, you may want to add a representative code sample to the file, then run *bench.pl* against the previous and current perls to see what difference it has made, and whether anything else has slowed down as a consequence.

The file *t/perf/opcount.t* is designed to test whether a particular code snippet has been compiled into an optree containing specified numbers of particular op types. This is good for testing whether optimisations which alter ops, such as converting an `aelem` op into an `aelemfast` op, are really doing that.

The files *t/perf/speed.t* and *t/re/speed.t* are designed to test things that run thousands of times slower if a particular optimisation is broken (for example, the utf8 length cache on long utf8 strings). Add a test that will take a fraction of a second normally, and minutes otherwise, causing the test file to time out on failure.

MORE READING FOR GUTS HACKERS

To hack on the Perl guts, you'll need to read the following things:

* *perlsource*

  An overview of the Perl source tree. This will help you find the files you're looking for.

* *perlinterp*

  An overview of the Perl interpreter source code and some details on how Perl does what it does.

* *perlhacktut*

  This document walks through the creation of a small patch to Perl's C code. If you're just getting started with Perl core hacking, this will help you understand how it works.

* *perlhacktips*

  More details on hacking the Perl core. This document focuses on lower level details such as how to write tests, compilation issues, portability, debugging, etc.

  If you plan on doing serious C hacking, make sure to read this.

* *perlguts*

  This is of paramount importance, since it's the documentation of what goes where in the Perl source. Read it over a couple of times and it might start to make sense - don't worry if it doesn't yet, because the best way to study it is to read it in conjunction with poking at Perl source, and we'll do that later on.

  Gisle Aas's "illustrated perlguts", also known as *illguts*, has very helpful pictures:

  http://search.cpan.org/dist/illguts/

* *perlxstut* and *perlxs*

  A working knowledge of XSUB programming is incredibly useful for core hacking; XSUBs use techniques drawn from the PP code, the portion of the guts that actually executes a Perl program. It's a lot gentler to learn those techniques from simple examples and explanation than from the core itself.

* *perlapi*

  The documentation for the Perl API explains what some of the internal functions do, as well as the many macros used in the source.

* *Porting/pumpkin.pod*

  This is a collection of words of wisdom for a Perl porter; some of it is only useful to the pumpkin holder, but most of it applies to anyone wanting to go about Perl development.

CPAN TESTERS AND PERL SMOKERS

The CPAN testers (*http://testers.cpan.org/*) are a group of volunteers who test CPAN modules on a variety of platforms.

Perl Smokers (*http://www.nntp.perl.org/group/perl.daily-build/* and *http://www.nntp.perl.org/group/perl.daily-build.reports/*) automatically test Perl source releases on platforms with various configurations.

Both efforts welcome volunteers. In order to get involved in smoke testing of the perl itself visit *http://search.cpan.org/dist/Test-Smoke/*. In order to start smoke testing CPAN modules visit *http://search.cpan.org/dist/CPANPLUS-YACSmoke/* or *http://search.cpan.org/dist/minismokebox/* or *http://search.cpan.org/dist/CPAN-Reporter/*.

WHAT NEXT?

If you've read all the documentation in the document and the ones listed above, you're more than ready to hack on Perl.

Here's some more recommendations

- Subscribe to perl5-porters, follow the patches and try and understand them; don't be afraid to ask if there's a portion you're not clear on - who knows, you may unearth a bug in the patch...
- Do read the README associated with your operating system, e.g. README.aix on the IBM AIX OS. Don't hesitate to supply patches to that README if you find anything missing or changed over a new OS release.
- Find an area of Perl that seems interesting to you, and see if you can work out how it works. Scan through the source, and step over it in the debugger. Play, poke, investigate, fiddle! You'll probably get to understand not just your chosen area but a much wider range of *perl*'s activity as well, and probably sooner than you'd think.

"The Road goes ever on and on, down from the door where it began."

If you can do these things, you've started on the long road to Perl porting. Thanks for wanting to help make Perl better - and happy hacking!

Metaphoric Quotations

If you recognized the quote about the Road above, you're in luck.

Most software projects begin each file with a literal description of each file's purpose. Perl instead begins each with a literary allusion to that file's purpose.

Like chapters in many books, all top-level Perl source files (along with a few others here and there) begin with an epigrammatic inscription that alludes, indirectly and metaphorically, to the material you're about to read.

Quotations are taken from writings of J.R.R. Tolkien pertaining to his Legendarium, almost always from *The Lord of the Rings*. Chapters and page numbers are given using the following editions:

- *The Hobbit*, by J.R.R. Tolkien. The hardcover, 70th-anniversary edition of 2007 was used, published in the UK by Harper Collins Publishers and in the US by the Houghton Mifflin Company.
- *The Lord of the Rings*, by J.R.R. Tolkien. The hardcover, 50th-anniversary edition of 2004 was used, published in the UK by Harper Collins Publishers and in the US by the Houghton Mifflin Company.
- *The Lays of Beleriand*, by J.R.R. Tolkien and published posthumously by his son and literary executor, C.J.R. Tolkien, being the 3rd of the 12 volumes in Christopher's mammoth *History of Middle Earth*. Page numbers derive from the hardcover edition, first published in 1983 by George Allen & Unwin; no page numbers changed for the special 3-volume omnibus edition of 2002 or the various trade-paper editions, all again now by Harper Collins or Houghton Mifflin.

Other JRRT books fair game for quotes would thus include *The Adventures of Tom Bombadil*, *The Silmarillion*, *Unfinished Tales*, and *The Tale of the Children of Hurin*, all but the first posthumously assembled by CJRT. But *The Lord of the Rings* itself is perfectly fine and probably best to quote from, provided you can find a suitable quote there.

So if you were to supply a new, complete, top-level source file to add to Perl, you should conform to this peculiar practice by yourself selecting an appropriate quotation from Tolkien, retaining the original spelling and punctuation and using the same format the rest of the quotes are in. Indirect and oblique is just fine; remember, it's a metaphor, so being meta is, after all, what it's for.

AUTHOR

This document was originally written by Nathan Torkington, and is maintained by the perl5-porters mailing list.