NAME

perlhack - How to hack at the Perl internals

DESCRIPTION

This document attempts to explain how Perl development takes place, and ends with some suggestions for people wanting to become bona fide porters.

The perl5-porters mailing list is where the Perl standard distribution is maintained and developed. The list can get anywhere from 10 to 150 messages a day, depending on the heatedness of the debate. Most days there are two or three patches, extensions, features, or bugs being discussed at a time.

A searchable archive of the list is at either:

```
http://www.xray.mpe.mpg.de/mailing-lists/perl5-porters/
```

or

```
http://archive.develooper.com/perl5-porters@perl.org/
```

List subscribers (the porters themselves) come in several flavours. Some are quiet curious lurkers, who rarely pitch in and instead watch the ongoing development to ensure they're forewarned of new changes or features in Perl. Some are representatives of vendors, who are there to make sure that Perl continues to compile and work on their platforms. Some patch any reported bug that they know how to fix, some are actively patching their pet area (threads, Win32, the regexp engine), while others seem to do nothing but complain. In other words, it's your usual mix of technical people.

Over this group of porters presides Larry Wall. He has the final word in what does and does not change in the Perl language. Various releases of Perl are shepherded by a "pumpking", a porter responsible for gathering patches, deciding on a patch-by-patch, feature-by-feature basis what will and will not go into the release. For instance, Gurusamy Sarathy was the pumpking for the 5.6 release of Perl, and Jarkko Hietaniemi was the pumpking for the 5.8 release, and Rafael Garcia-Suarez holds the pumpking crown for the 5.10 release.

In addition, various people are pumpkings for different things. For instance, Andy Dougherty and Jarkko Hietaniemi did a grand job as the *Configure* pumpkin up till the 5.8 release. For the 5.10 release H.Merijn Brand took over.

Larry sees Perl development along the lines of the US government: there's the Legislature (the porters), the Executive branch (the pumpkings), and the Supreme Court (Larry). The legislature can discuss and submit patches to the executive branch all they like, but the executive branch is free to veto them. Rarely, the Supreme Court will side with the executive branch over the legislature, or the legislature over the executive branch. Mostly, however, the legislature and the executive branch are supposed to get along and work out their differences without impeachment or court cases.

You might sometimes see reference to Rule 1 and Rule 2. Larry's power as Supreme Court is expressed in The Rules:

1 Larry is always by definition right about how Perl should behave. This means he has final veto power on the core functionality.

2 Larry is allowed to change his mind about any matter at a later date, regardless of whether he previously invoked Rule 1.

Got that? Larry is always right, even when he was wrong. It's rare to see either Rule exercised, but they are often alluded to.

New features and extensions to the language are contentious, because the criteria used by the pumpkings, Larry, and other porters to decide which features should be implemented and incorporated are not codified in a few small design goals as with some other languages. Instead, the

heuristics are flexible and often difficult to fathom. Here is one person's list, roughly in decreasing order of importance, of heuristics that new features have to be weighed against:

Does concept match the general goals of Perl?

These haven't been written anywhere in stone, but one approximation is:

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 are 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.

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. If it can be done in a module instead of in the core, it's highly unlikely to be added.

Is the feature generic enough?

Is this something that only the submitter wants added to the language, or would it be 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. For instance, instead of implementing a "delayed evaluation" feature, the porters are waiting for a macro system that would permit delayed evaluation and much more.

Does it potentially introduce new bugs?

Radical rewrites of large chunks of the Perl interpreter have the potential to introduce new bugs. The smaller and more localized the change, the better.

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 a system-specific features. It's highly unlikely that nonportable 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. Nothing can be added without documentation, so submitting a patch for the appropriate manpages as well as the source code is always a good idea.

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.

If you're on the list, you might hear the word "core" bandied around. It refers to the standard distribution. "Hacking on the core" means you're changing the C source code to the Perl interpreter. "A core module" is one that ships with Perl.

Keeping in sync

The source code to the Perl interpreter, in its different versions, is kept in a repository managed by a revision control system (which is currently the Perforce program, see http://perforce.com/). The pumpkings and a few others have access to the repository to check in changes. Periodically the pumpking for the development version of Perl will release a new version, so the rest of the porters can see what's changed. The current state of the main trunk of repository, and patches that describe the individual changes that have happened since the last public release are available at this location:

```
http://public.activestate.com/pub/apc/
ftp://public.activestate.com/pub/apc/
```

If you're looking for a particular change, or a change that affected a particular set of files, you may find the **Perl Repository Browser** useful:

```
http://public.activestate.com/cgi-bin/perlbrowse
```

You may also want to subscribe to the perl5-changes mailing list to receive a copy of each patch that gets submitted to the maintenance and development "branches" of the perl repository. See http://lists.perl.org/ for subscription information.

If you are a member of the perl5-porters mailing list, it is a good thing to keep in touch with the most recent changes. If not only to verify if what you would have posted as a bug report isn't already solved in the most recent available perl development branch, also known as perl-current, bleading edge perl, bleedperl or bleadperl.

Needless to say, the source code in perl-current is usually in a perpetual state of evolution. You should expect it to be very buggy. Do **not** use it for any purpose other than testing and development.

Keeping in sync with the most recent branch can be done in several ways, but the most convenient and reliable way is using **rsync**, available at ftp://rsync.samba.org/pub/rsync/ . (You can also get the most recent branch by FTP.)

If you choose to keep in sync using rsync, there are two approaches to doing so:

rsync'ing the source tree

Presuming you are in the directory where your perl source resides and you have rsync installed and available, you can "upgrade" to the bleadperl using:

```
 # rsync -avz rsync://public.activestate.com/perl-current/ .
```

This takes care of updating every single item in the source tree to the latest applied patch level, creating files that are new (to your distribution) and setting date/time stamps of existing files to reflect the bleadperl status.

Note that this will not delete any files that were in '.' before the rsync. Once you are sure that the rsync is running correctly, run it with the --delete and the --dry-run options like this:

```
 # rsync -avz --delete --dry-run
rsync://public.activestate.com/perl-current/ .
```

This will *simulate* an rsync run that also deletes files not present in the bleadperl master copy. Observe the results from this run closely. If you are sure that the actual run would delete no files precious to you, you could remove the '--dry-run' option.

You can than check what patch was the latest that was applied by looking in the file **.patch**, which will show the number of the latest patch.

If you have more than one machine to keep in sync, and not all of them have access to the WAN (so you are not able to rsync all the source trees to the real source), there are some ways to get around this problem.

Using rsync over the LAN

Set up a local rsync server which makes the rsynced source tree available to the LAN and sync the other machines against this directory.

From http://rsync.samba.org/README.html :

```
   "Rsync uses rsh or ssh for communication. It does not need to
 be
    setuid and requires no special privileges for installation.
  It
    does not require an inetd entry or a daemon.  You must,
however,
    have a working rsh or ssh system.  Using ssh is recommended
for
    its security features."
```

Using pushing over the NFS

Having the other systems mounted over the NFS, you can take an active pushing approach by checking the just updated tree against the other not-yet synced trees. An example would be

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

  use strict;
  use File::Copy;
```

```
  my %MF = map {
      m/(\S+)/;
      $1 => [ (stat $1)[2, 7, 9] ]; # mode, size, mtime
      } `cat MANIFEST`;

  my %remote = map { $_ => "/$_/pro/3gl/CPAN/perl-5.7.1" }
qw(host1 host2);

  foreach my $host (keys %remote) {
      unless (-d $remote{$host}) {
   print STDERR "Cannot Xsync for host $host\n";
   next;
   }
      foreach my $file (keys %MF) {
   my $rfile = "$remote{$host}/$file";
   my ($mode, $size, $mtime) = (stat $rfile)[2, 7, 9];
   defined $size or ($mode, $size, $mtime) = (0, 0, 0);
   $size == $MF{$file}[1] && $mtime == $MF{$file}[2] and next;
   printf "%4s %-34s %8d %9d  %8d %9d\n",
       $host, $file, $MF{$file}[1], $MF{$file}[2], $size,
$mtime;
   unlink $rfile;
   copy ($file, $rfile);
   utime time, $MF{$file}[2], $rfile;
   chmod $MF{$file}[0], $rfile;
   }
      }
```

though this is not perfect. It could be improved with checking file checksums before updating. Not all NFS systems support reliable utime support (when used over the NFS).

rsync'ing the patches

The source tree is maintained by the pumpking who applies patches to the files in the tree. These patches are either created by the pumpking himself using `diff -c` after updating the file manually or by applying patches sent in by posters on the perl5-porters list. These patches are also saved and rsync'able, so you can apply them yourself to the source files.

Presuming you are in a directory where your patches reside, you can get them in sync with

```
# rsync -avz rsync://public.activestate.com/perl-current-diffs/ .
```

This makes sure the latest available patch is downloaded to your patch directory.

It's then up to you to apply these patches, using something like

```
# last=`ls -t *.gz | sed q`
# rsync -avz rsync://public.activestate.com/perl-current-diffs/ .
# find . -name '*.gz' -newer $last -exec gzcat {} \; >blead.patch
# cd ../perl-current
# patch -p1 -N <../perl-current-diffs/blead.patch
```

or, since this is only a hint towards how it works, use CPAN-patchaperl from Andreas König to have better control over the patching process.

Why rsync the source tree

It's easier to rsync the source tree

Since you don't have to apply the patches yourself, you are sure all files in the source tree are

in the right state.

It's more reliable

While both the rsync-able source and patch areas are automatically updated every few minutes, keep in mind that applying patches may sometimes mean careful hand-holding, especially if your version of the `patch` program does not understand how to deal with new files, files with 8-bit characters, or files without trailing newlines.

Why rsync the patches

It's easier to rsync the patches

If you have more than one machine that you want to keep in track with bleadperl, it's easier to rsync the patches only once and then apply them to all the source trees on the different machines.

In case you try to keep in pace on 5 different machines, for which only one of them has access to the WAN, rsync'ing all the source trees should than be done 5 times over the NFS. Having rsync'ed the patches only once, I can apply them to all the source trees automatically. Need you say more ;-)

It's a good reference

If you do not only like to have the most recent development branch, but also like to **fix** bugs, or extend features, you want to dive into the sources. If you are a seasoned perl core diver, you don't need no manuals, tips, roadmaps, perlguts.pod or other aids to find your way around. But if you are a starter, the patches may help you in finding where you should start and how to change the bits that bug you.

The file **Changes** is updated on occasions the pumpking sees as his own little sync points. On those occasions, he releases a tar-ball of the current source tree (i.e. perl@7582.tar.gz), which will be an excellent point to start with when choosing to use the 'rsync the patches' scheme. Starting with perl@7582, which means a set of source files on which the latest applied patch is number 7582, you apply all succeeding patches available from then on (7583, 7584, ...).

You can use the patches later as a kind of search archive.

Finding a start point

If you want to fix/change the behaviour of function/feature Foo, just scan the patches for patches that mention Foo either in the subject, the comments, or the body of the fix. A good chance the patch shows you the files that are affected by that patch which are very likely to be the starting point of your journey into the guts of perl.

Finding how to fix a bug

If you've found *where* the function/feature Foo misbehaves, but you don't know how to fix it (but you do know the change you want to make), you can, again, peruse the patches for similar changes and look how others apply the fix.

Finding the source of misbehaviour

When you keep in sync with bleadperl, the pumpking would love to *see* that the community efforts really work. So after each of his sync points, you are to 'make test' to check if everything is still in working order. If it is, you do 'make ok', which will send an OK report to perlbug@perl.org. (If you do not have access to a mailer from the system you just finished successfully 'make test', you can do 'make okfile', which creates the file `perl.ok`, which you can than take to your favourite mailer and mail yourself).

But of course, as always, things will not always lead to a success path, and one or more test do not pass the 'make test'. Before sending in a bug report (using 'make nok' or 'make nokfile'), check the mailing list if someone else has reported the bug already and if so, confirm it by replying to that message. If not, you might want to trace the

source of that misbehaviour **before** sending in the bug, which will help all the other porters in finding the solution.

Here the saved patches come in very handy. You can check the list of patches to see which patch changed what file and what change caused the misbehaviour. If you note that in the bug report, it saves the one trying to solve it, looking for that point.

If searching the patches is too bothersome, you might consider using perl's bugtron to find more information about discussions and ramblings on posted bugs.

If you want to get the best of both worlds, rsync both the source tree for convenience, reliability and ease and rsync the patches for reference.

Working with the source

Because you cannot use the Perforce client, you cannot easily generate diffs against the repository, nor will merges occur when you update via rsync. If you edit a file locally and then rsync against the latest source, changes made in the remote copy will *overwrite* your local versions!

The best way to deal with this is to maintain a tree of symlinks to the rsync'd source. Then, when you want to edit a file, you remove the symlink, copy the real file into the other tree, and edit it. You can then diff your edited file against the original to generate a patch, and you can safely update the original tree.

Perl's *Configure* script can generate this tree of symlinks for you. The following example assumes that you have used rsync to pull a copy of the Perl source into the *perl-rsync* directory. In the directory above that one, you can execute the following commands:

```
mkdir perl-dev
cd perl-dev
../perl-rsync/Configure -Dmksymlinks -Dusedevel -D"optimize=-g"
```

This will start the Perl configuration process. After a few prompts, you should see something like this:

```
Symbolic links are supported.

Checking how to test for symbolic links...
Your builtin 'test -h' may be broken.
Trying external '/usr/bin/test -h'.
You can test for symbolic links with '/usr/bin/test -h'.

Creating the symbolic links...
(First creating the subdirectories...)
(Then creating the symlinks...)
```

The specifics may vary based on your operating system, of course. After you see this, you can abort the *Configure* script, and you will see that the directory you are in has a tree of symlinks to the *perl-rsync* directories and files.

If you plan to do a lot of work with the Perl source, here are some Bourne shell script functions that can make your life easier:

```
    function edit {
 if [ -L $1 ]; then
     mv $1 $1.orig
  cp $1.orig $1
  vi $1
 else
     /bin/vi $1
```

```
 fi
   }

   function unedit {
if [ -L $1.orig ]; then
    rm $1
 mv $1.orig $1
 fi
   }
```

Replace "vi" with your favorite flavor of editor.

Here is another function which will quickly generate a patch for the files which have been edited in your symlink tree:

```
   mkpatchorig() {
local diffopts
    for f in `find . -name '*.orig' | sed s,^\./,,`
 do
    case `echo $f | sed 's,.orig$,,;s,.*\.,,'` in
  c)   diffopts=-p ;;
 pod) diffopts='-F^=' ;;
 *)   diffopts= ;;
 esac
    diff -du $diffopts $f `echo $f | sed 's,.orig$,,'`
    done
   }
```

This function produces patches which include enough context to make your changes obvious. This makes it easier for the Perl pumpking(s) to review them when you send them to the perl5-porters list, and that means they're more likely to get applied.

This function assumed a GNU diff, and may require some tweaking for other diff variants.

Perlbug administration

There is a single remote administrative interface for modifying bug status, category, open issues etc. using the **RT** *bugtracker* system, maintained by *Robert Spier*. Become an administrator, and close any bugs you can get your sticky mitts on:

```
http://rt.perl.org
```

The bugtracker mechanism for **perl5** bugs in particular is at:

```
http://bugs6.perl.org/perlbug
```

To email the bug system administrators:

```
"perlbug-admin" <perlbug-admin@perl.org>
```

Submitting patches

Always submit patches to *perl5-porters@perl.org*. If you're patching a core module and there's an author listed, send the author a copy (see *Patching a core module*). This lets other porters review your patch, which catches a surprising number of errors in patches. Either use the diff program (available in source code form from ftp://ftp.gnu.org/pub/gnu/ , or use Johan Vromans' *makepatch* (available from *CPAN/authors/id/JV/*). Unified diffs are preferred, but context diffs are accepted. Do not send RCS-style diffs or diffs without context lines. More information is given in the

Porting/patching.pod file in the Perl source distribution. Please patch against the latest **development** version (e.g., if you're fixing a bug in the 5.005 track, patch against the latest 5.005_5x version). Only patches that survive the heat of the development branch get applied to maintenance versions.

Your patch should update the documentation and test suite. See *Writing a test*.

To report a bug in Perl, use the program *perlbug* which comes with Perl (if you can't get Perl to work, send mail to the address *perlbug@perl.org* or *perlbug@perl.com*). Reporting bugs through *perlbug* feeds into the automated bug-tracking system, access to which is provided through the web at http://bugs.perl.org/ . It often pays to check the archives of the perl5-porters mailing list to see whether the bug you're reporting has been reported before, and if so whether it was considered a bug. See above for the location of the searchable archives.

The CPAN testers (http://testers.cpan.org/) are a group of volunteers who test CPAN modules on a variety of platforms. Perl Smokers (http://archives.develooper.com/daily-build@perl.org/) automatically tests Perl source releases on platforms with various configurations. Both efforts welcome volunteers.

It's a good idea to read and lurk for a while before chipping in. That way you'll get to see the dynamic of the conversations, learn the personalities of the players, and hopefully be better prepared to make a useful contribution when do you speak up.

If after all this you still think you want to join the perl5-porters mailing list, send mail to *perl5-porters-subscribe@perl.org*. To unsubscribe, send mail to *perl5-porters-unsubscribe@perl.org*.

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

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.
>
> You might also want to look at Gisle Aas's illustrated perlguts - there's no guarantee that this will be absolutely up-to-date with the latest documentation in the Perl core, but the fundamentals will be right. (http://gisle.aas.no/perl/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.

The perl5-porters FAQ

> This should be available from http://simon-cozens.org/writings/p5p-faq ; alternatively, you can get the FAQ emailed to you by sending mail to `perl5-porters-faq@perl.org`. It contains hints on reading perl5-porters, information on how perl5-porters works and how Perl development in general works.

Finding Your Way Around

Perl maintenance can be split into a number of areas, and certain people (pumpkins) will have responsibility for each area. These areas sometimes correspond to files or directories in the source kit. Among the areas are:

Core modules

Modules shipped as part of the Perl core live in the *lib/* and *ext/* subdirectories: *lib/* is for the pure-Perl modules, and *ext/* contains the core XS modules.

Tests

There are tests for nearly all the modules, built-ins and major bits of functionality. Test files all have a .t suffix. Module tests live in the *lib/* and *ext/* directories next to the module being tested. Others live in *t/*. See *Writing a test*

Documentation

Documentation maintenance includes looking after everything in the *pod/* directory, (as well as contributing new documentation) and the documentation to the modules in core.

Configure

The configure process is the way we make Perl portable across the myriad of operating systems it supports. Responsibility for the configure, build and installation process, as well as the overall portability of the core code rests with the configure pumpkin - others help out with individual operating systems.

The files involved are the operating system directories, (*win32/*, *os2/*, *vms/* and so on) the shell scripts which generate *config.h* and *Makefile*, as well as the metaconfig files which generate *Configure*. (metaconfig isn't included in the core distribution.)

Interpreter

And of course, there's the core of the Perl interpreter itself. Let's have a look at that in a little more detail.

Before we leave looking at the layout, though, don't forget that *MANIFEST* contains not only the file names in the Perl distribution, but short descriptions of what's in them, too. For an overview of the important files, try this:

```
perl -lne 'print if /^[^\/]+\.[ch]\s+/' MANIFEST
```

Elements of the interpreter

The work of the interpreter has two main stages: compiling the code into the internal representation, or bytecode, and then executing it. *"Compiled code" in perlguts* explains exactly how the compilation stage happens.

Here is a short breakdown of perl's operation:

Startup

The action begins in *perlmain.c*. (or *miniperlmain.c* for miniperl) This is very high-level code, enough to fit on a single screen, and it resembles the code found in *perlembed*; most of the real action takes place in *perl.c*

First, *perlmain.c* allocates some memory and constructs a Perl interpreter:

```
1 PERL_SYS_INIT3(&argc,&argv,&env);
2
3 if (!PL_do_undump) {
4     my_perl = perl_alloc();
5     if (!my_perl)
6         exit(1);
```

```
7     perl_construct(my_perl);
8     PL_perl_destruct_level = 0;
9 }
```

Line 1 is a macro, and its definition is dependent on your operating system. Line 3 references `PL_do_undump`, a global variable - all global variables in Perl start with `PL_`. This tells you whether the current running program was created with the `-u` flag to perl and then *undump*, which means it's going to be false in any sane context.

Line 4 calls a function in *perl.c* to allocate memory for a Perl interpreter. It's quite a simple function, and the guts of it looks like this:

```
    my_perl =
(PerlInterpreter*)PerlMem_malloc(sizeof(PerlInterpreter));
```

Here you see an example of Perl's system abstraction, which we'll see later: `PerlMem_malloc` is either your system's `malloc`, or Perl's own `malloc` as defined in *malloc.c* if you selected that option at configure time.

Next, in line 7, we construct the interpreter; this sets up all the special variables that Perl needs, the stacks, and so on.

Now we pass Perl the command line options, and tell it to go:

```
    exitstatus = perl_parse(my_perl, xs_init, argc, argv, (char
**)NULL);
    if (!exitstatus) {
        exitstatus = perl_run(my_perl);
    }
```

`perl_parse` is actually a wrapper around `S_parse_body`, as defined in *perl.c*, which processes the command line options, sets up any statically linked XS modules, opens the program and calls `yyparse` to parse it.

Parsing

The aim of this stage is to take the Perl source, and turn it into an op tree. We'll see what one of those looks like later. Strictly speaking, there's three things going on here.

`yyparse`, the parser, lives in *perly.c*, although you're better off reading the original YACC input in *perly.y*. (Yes, Virginia, there **is** a YACC grammar for Perl!) The job of the parser is to take your code and "understand" it, splitting it into sentences, deciding which operands go with which operators and so on.

The parser is nobly assisted by the lexer, which chunks up your input into tokens, and decides what type of thing each token is: a variable name, an operator, a bareword, a subroutine, a core function, and so on. The main point of entry to the lexer is `yylex`, and that and its associated routines can be found in *toke.c*. Perl isn't much like other computer languages; it's highly context sensitive at times, it can be tricky to work out what sort of token something is, or where a token ends. As such, there's a lot of interplay between the tokeniser and the parser, which can get pretty frightening if you're not used to it.

As the parser understands a Perl program, it builds up a tree of operations for the interpreter to perform during execution. The routines which construct and link together the various operations are to be found in *op.c*, and will be examined later.

Optimization

Now the parsing stage is complete, and the finished tree represents the operations that the Perl interpreter needs to perform to execute our program. Next, Perl does a dry run over the tree looking for optimisations: constant expressions such as `3 + 4` will be computed now, and the optimizer will also see if any multiple operations can be replaced with a single one. For instance, to fetch the variable `$foo`, instead of grabbing the glob `*foo` and looking at the scalar

component, the optimizer fiddles the op tree to use a function which directly looks up the scalar in question. The main optimizer is `peep` in *op.c*, and many ops have their own optimizing functions.

Running

Now we're finally ready to go: we have compiled Perl byte code, and all that's left to do is run it. The actual execution is done by the `runops_standard` function in *run.c*; more specifically, it's done by these three innocent looking lines:

```
while ((PL_op = CALL_FPTR(PL_op->op_ppaddr)(aTHX))) {
    PERL_ASYNC_CHECK();
}
```

You may be more comfortable with the Perl version of that:

```
PERL_ASYNC_CHECK() while $Perl::op = &{$Perl::op->{function}};
```

Well, maybe not. Anyway, each op contains a function pointer, which stipulates the function which will actually carry out the operation. This function will return the next op in the sequence - this allows for things like `if` which choose the next op dynamically at run time. The `PERL_ASYNC_CHECK` makes sure that things like signals interrupt execution if required.

The actual functions called are known as PP code, and they're spread between four files: *pp_hot.c* contains the "hot" code, which is most often used and highly optimized, *pp_sys.c* contains all the system-specific functions, *pp_ctl.c* contains the functions which implement control structures (`if`, `while` and the like) and *pp.c* contains everything else. These are, if you like, the C code for Perl's built-in functions and operators.

Note that each `pp_` function is expected to return a pointer to the next op. Calls to perl subs (and eval blocks) are handled within the same runops loop, and do not consume extra space on the C stack. For example, `pp_entersub` and `pp_entertry` just push a `CxSUB` or `CxEVAL` block struct onto the context stack which contain the address of the op following the sub call or eval. They then return the first op of that sub or eval block, and so execution continues of that sub or block. Later, a `pp_leavesub` or `pp_leavetry` op pops the `CxSUB` or `CxEVAL`, retrieves the return op from it, and returns it.

Exception handing

Perl's exception handing (i.e. `die` etc) is built on top of the low-level `setjmp()`/`longjmp()` C-library functions. These basically provide a way to capture the current PC and SP registers and later restore them; i.e. a `longjmp()` continues at the point in code where a previous `setjmp()` was done, with anything further up on the C stack being lost. This is why code should always save values using `SAVE_FOO` rather than in auto variables.

The perl core wraps `setjmp()` etc in the macros `JMPENV_PUSH` and `JMPENV_JUMP`. The basic rule of perl exceptions is that `exit`, and `die` (in the absence of `eval`) perform a `JMPENV_JUMP(2)`, while `die` within `eval` does a `JMPENV_JUMP(3)`.

At entry points to perl, such as `perl_parse()`, `perl_run()` and `call_sv(cv, G_EVAL)` each does a `JMPENV_PUSH`, then enter a runops loop or whatever, and handle possible exception returns. For a 2 return, final cleanup is performed, such as popping stacks and calling `CHECK` or `END` blocks. Amongst other things, this is how scope cleanup still occurs during an `exit`.

If a `die` can find a `CxEVAL` block on the context stack, then the stack is popped to that level and the return op in that block is assigned to `PL_restartop`; then a `JMPENV_JUMP(3)` is performed. This normally passes control back to the guard. In the case of `perl_run` and `call_sv`, a non-null `PL_restartop` triggers re-entry to the runops loop. The is the normal way that `die` or `croak` is handled within an `eval`.

Sometimes ops are executed within an inner runops loop, such as tie, sort or overload code. In this case, something like

```
sub FETCH { eval { die } }
```

would cause a longjmp right back to the guard in `perl_run`, popping both runops loops, which is clearly incorrect. One way to avoid this is for the tie code to do a `JMPENV_PUSH` before executing `FETCH` in the inner runops loop, but for efficiency reasons, perl in fact just sets a flag, using `CATCH_SET(TRUE)`. The `pp_require`, `pp_entereval` and `pp_entertry` ops check this flag, and if true, they call `docatch`, which does a `JMPENV_PUSH` and starts a new runops level to execute the code, rather than doing it on the current loop.

As a further optimisation, on exit from the eval block in the `FETCH`, execution of the code following the block is still carried on in the inner loop. When an exception is raised, `docatch` compares the `JMPENV` level of the `CxEVAL` with `PL_top_env` and if they differ, just re-throws the exception. In this way any inner loops get popped.

Here's an example.

```
1: eval { tie @a, 'A' };
2: sub A::TIEARRAY {
3:     eval { die };
4:     die;
5: }
```

To run this code, `perl_run` is called, which does a `JMPENV_PUSH` then enters a runops loop. This loop executes the eval and tie ops on line 1, with the eval pushing a `CxEVAL` onto the context stack.

The `pp_tie` does a `CATCH_SET(TRUE)`, then starts a second runops loop to execute the body of `TIEARRAY`. When it executes the entertry op on line 3, `CATCH_GET` is true, so `pp_entertry` calls `docatch` which does a `JMPENV_PUSH` and starts a third runops loop, which then executes the die op. At this point the C call stack looks like this:

```
Perl_pp_die
Perl_runops      # third loop
S_docatch_body
S_docatch
Perl_pp_entertry
Perl_runops      # second loop
S_call_body
Perl_call_sv
Perl_pp_tie
Perl_runops      # first loop
S_run_body
perl_run
main
```

and the context and data stacks, as shown by `-Dstv`, look like:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
  CX 1: EVAL   => AV()  PV("A"\0)
  retop=leave
STACK 1: MAGIC
  CX 0: SUB    =>
  retop=(null)
  CX 1: EVAL   => *
retop=nextstate
```

The die pops the first `CxEVAL` off the context stack, sets `PL_restartop` from it, does a `JMPENV_JUMP(3)`, and control returns to the top `docatch`. This then starts another third-level runops level, which executes the nextstate, pushmark and die ops on line 4. At the point that the

second pp_die is called, the C call stack looks exactly like that above, even though we are no longer within an inner eval; this is because of the optimization mentioned earlier. However, the context stack now looks like this, ie with the top CxEVAL popped:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
  CX 1: EVAL   => AV()  PV("A"\0)
  retop=leave
STACK 1: MAGIC
  CX 0: SUB    =>
  retop=(null)
```

The die on line 4 pops the context stack back down to the CxEVAL, leaving it as:

```
STACK 0: MAIN
  CX 0: BLOCK  =>
```

As usual, PL_restartop is extracted from the CxEVAL, and a JMPENV_JUMP(3) done, which pops the C stack back to the docatch:

```
S_docatch
Perl_pp_entertry
Perl_runops      # second loop
S_call_body
Perl_call_sv
Perl_pp_tie
Perl_runops      # first loop
S_run_body
perl_run
main
```

In this case, because the JMPENV level recorded in the CxEVAL differs from the current one, docatch just does a JMPENV_JUMP(3) and the C stack unwinds to:

```
perl_run
main
```

Because PL_restartop is non-null, run_body starts a new runops loop and execution continues.

Internal Variable Types

You should by now have had a look at *perlguts*, which tells you about Perl's internal variable types: SVs, HVs, AVs and the rest. If not, do that now.

These variables are used not only to represent Perl-space variables, but also any constants in the code, as well as some structures completely internal to Perl. The symbol table, for instance, is an ordinary Perl hash. Your code is represented by an SV as it's read into the parser; any program files you call are opened via ordinary Perl filehandles, and so on.

The core *Devel::Peek* module lets us examine SVs from a Perl program. Let's see, for instance, how Perl treats the constant "hello".

```
  % perl -MDevel::Peek -e 'Dump("hello")'
1 SV = PV(0xa041450) at 0xa04ecbc
2   REFCNT = 1
3   FLAGS = (POK,READONLY,pPOK)
4   PV = 0xa0484e0 "hello"\0
5   CUR = 5
6   LEN = 6
```

Reading `Devel::Peek` output takes a bit of practise, so let's go through it line by line.

Line 1 tells us we're looking at an SV which lives at `0xa04ecbc` in memory. SVs themselves are very simple structures, but they contain a pointer to a more complex structure. In this case, it's a PV, a structure which holds a string value, at location `0xa041450`. Line 2 is the reference count; there are no other references to this data, so it's 1.

Line 3 are the flags for this SV - it's OK to use it as a PV, it's a read-only SV (because it's a constant) and the data is a PV internally. Next we've got the contents of the string, starting at location `0xa0484e0`.

Line 5 gives us the current length of the string - note that this does **not** include the null terminator. Line 6 is not the length of the string, but the length of the currently allocated buffer; as the string grows, Perl automatically extends the available storage via a routine called `SvGROW`.

You can get at any of these quantities from C very easily; just add `Sv` to the name of the field shown in the snippet, and you've got a macro which will return the value: `SvCUR(sv)` returns the current length of the string, `SvREFCOUNT(sv)` returns the reference count, `SvPV(sv, len)` returns the string itself with its length, and so on. More macros to manipulate these properties can be found in *perlguts*.

Let's take an example of manipulating a PV, from `sv_catpvn`, in *sv.c*

```
     1  void
     2  Perl_sv_catpvn(pTHX_ register SV *sv, register const char *ptr,
register STRLEN len)
     3  {
     4      STRLEN tlen;
     5      char *junk;

     6      junk = SvPV_force(sv, tlen);
     7      SvGROW(sv, tlen + len + 1);
     8      if (ptr == junk)
     9          ptr = SvPVX(sv);
    10      Move(ptr,SvPVX(sv)+tlen,len,char);
    11      SvCUR(sv) += len;
    12      *SvEND(sv) = '\0';
    13      (void)SvPOK_only_UTF8(sv);          /* validate pointer */
    14      SvTAINT(sv);
    15  }
```

This is a function which adds a string, `ptr`, of length `len` onto the end of the PV stored in `sv`. The first thing we do in line 6 is make sure that the SV **has** a valid PV, by calling the `SvPV_force` macro to force a PV. As a side effect, `tlen` gets set to the current value of the PV, and the PV itself is returned to `junk`.

In line 7, we make sure that the SV will have enough room to accommodate the old string, the new string and the null terminator. If `LEN` isn't big enough, `SvGROW` will reallocate space for us.

Now, if `junk` is the same as the string we're trying to add, we can grab the string directly from the SV; `SvPVX` is the address of the PV in the SV.

Line 10 does the actual catenation: the `Move` macro moves a chunk of memory around: we move the string `ptr` to the end of the PV - that's the start of the PV plus its current length. We're moving `len` bytes of type `char`. After doing so, we need to tell Perl we've extended the string, by altering `CUR` to reflect the new length. `SvEND` is a macro which gives us the end of the string, so that needs to be a `"\0"`.

Line 13 manipulates the flags; since we've changed the PV, any IV or NV values will no longer be

valid: if we have `$a=10; $a.="6";` we don't want to use the old IV of 10. `SvPOK_only_utf8` is a special UTF-8-aware version of `SvPOK_only`, a macro which turns off the IOK and NOK flags and turns on POK. The final `SvTAINT` is a macro which launders tainted data if taint mode is turned on.

AVs and HVs are more complicated, but SVs are by far the most common variable type being thrown around. Having seen something of how we manipulate these, let's go on and look at how the op tree is constructed.

Op Trees

First, what is the op tree, anyway? The op tree is the parsed representation of your program, as we saw in our section on parsing, and it's the sequence of operations that Perl goes through to execute your program, as we saw in *Running*.

An op is a fundamental operation that Perl can perform: all the built-in functions and operators are ops, and there are a series of ops which deal with concepts the interpreter needs internally - entering and leaving a block, ending a statement, fetching a variable, and so on.

The op tree is connected in two ways: you can imagine that there are two "routes" through it, two orders in which you can traverse the tree. First, parse order reflects how the parser understood the code, and secondly, execution order tells perl what order to perform the operations in.

The easiest way to examine the op tree is to stop Perl after it has finished parsing, and get it to dump out the tree. This is exactly what the compiler backends *B::Terse*, *B::Concise* and *B::Debug* do.

Let's have a look at how Perl sees `$a = $b + $c`:

```
 % perl -MO=Terse -e '$a=$b+$c'
 1  LISTOP (0x8179888) leave
 2      OP (0x81798b0) enter
 3      COP (0x8179850) nextstate
 4      BINOP (0x8179828) sassign
 5          BINOP (0x8179800) add [1]
 6              UNOP (0x81796e0) null [15]
 7                  SVOP (0x80fafe0) gvsv  GV (0x80fa4cc) *b
 8              UNOP (0x81797e0) null [15]
 9                  SVOP (0x8179700) gvsv  GV (0x80efeb0) *c
10          UNOP (0x816b4f0) null [15]
11              SVOP (0x816dcf0) gvsv  GV (0x80fa460) *a
```

Let's start in the middle, at line 4. This is a BINOP, a binary operator, which is at location `0x8179828`. The specific operator in question is `sassign` - scalar assignment - and you can find the code which implements it in the function `pp_sassign` in *pp_hot.c*. As a binary operator, it has two children: the add operator, providing the result of `$b+$c`, is uppermost on line 5, and the left hand side is on line 10.

Line 10 is the null op: this does exactly nothing. What is that doing there? If you see the null op, it's a sign that something has been optimized away after parsing. As we mentioned in *Optimization*, the optimization stage sometimes converts two operations into one, for example when fetching a scalar variable. When this happens, instead of rewriting the op tree and cleaning up the dangling pointers, it's easier just to replace the redundant operation with the null op. Originally, the tree would have looked like this:

```
10          SVOP (0x816b4f0) rv2sv [15]
11              SVOP (0x816dcf0) gv  GV (0x80fa460) *a
```

That is, fetch the a entry from the main symbol table, and then look at the scalar component of it: `gvsv` (`pp_gvsv` into *pp_hot.c*) happens to do both these things.

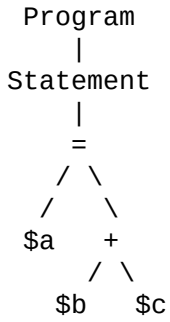
The right hand side, starting at line 5 is similar to what we've just seen: we have the `add` op (`pp_add` also in *pp_hot.c*) add together two `gvsv`s.

Now, what's this about?

```
1  LISTOP (0x8179888) leave
2      OP (0x81798b0) enter
3      COP (0x8179850) nextstate
```

`enter` and `leave` are scoping ops, and their job is to perform any housekeeping every time you enter and leave a block: lexical variables are tidied up, unreferenced variables are destroyed, and so on. Every program will have those first three lines: `leave` is a list, and its children are all the statements in the block. Statements are delimited by `nextstate`, so a block is a collection of `nextstate` ops, with the ops to be performed for each statement being the children of `nextstate`. `enter` is a single op which functions as a marker.

That's how Perl parsed the program, from top to bottom:



However, it's impossible to **perform** the operations in this order: you have to find the values of `$b` and `$c` before you add them together, for instance. So, the other thread that runs through the op tree is the execution order: each op has a field `op_next` which points to the next op to be run, so following these pointers tells us how perl executes the code. We can traverse the tree in this order using the `exec` option to `B::Terse`:

```
% perl -MO=Terse,exec -e '$a=$b+$c'
1  OP (0x8179928) enter
2  COP (0x81798c8) nextstate
3  SVOP (0x81796c8) gvsv  GV (0x80fa4d4) *b
4  SVOP (0x8179798) gvsv  GV (0x80efeb0) *c
5  BINOP (0x8179878) add [1]
6  SVOP (0x816dd38) gvsv  GV (0x80fa468) *a
7  BINOP (0x81798a0) sassign
8  LISTOP (0x8179900) leave
```

This probably makes more sense for a human: enter a block, start a statement. Get the values of `$b` and `$c`, and add them together. Find `$a`, and assign one to the other. Then leave.

The way Perl builds up these op trees in the parsing process can be unravelled by examining *perly.y*, the YACC grammar. Let's take the piece we need to construct the tree for `$a = $b + $c`

```
1 term    :   term ASSIGNOP term
2                { $$ = newASSIGNOP(OPf_STACKED, $1, $2, $3); }
3         |   term ADDOP term
4                { $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

If you're not used to reading BNF grammars, this is how it works: You're fed certain things by the

tokeniser, which generally end up in upper case. Here, ADDOP, is provided when the tokeniser sees + in your code. ASSIGNOP is provided when = is used for assigning. These are "terminal symbols", because you can't get any simpler than them.

The grammar, lines one and three of the snippet above, tells you how to build up more complex forms. These complex forms, "non-terminal symbols" are generally placed in lower case. `term` here is a non-terminal symbol, representing a single expression.

The grammar gives you the following rule: you can make the thing on the left of the colon if you see all the things on the right in sequence. This is called a "reduction", and the aim of parsing is to completely reduce the input. There are several different ways you can perform a reduction, separated by vertical bars: so, `term` followed by = followed by `term` makes a `term`, and `term` followed by + followed by `term` can also make a `term`.

So, if you see two terms with an = or +, between them, you can turn them into a single expression. When you do this, you execute the code in the block on the next line: if you see =, you'll do the code in line 2. If you see +, you'll do the code in line 4. It's this code which contributes to the op tree.

```
|   term ADDOP term
{ $$ = newBINOP($2, 0, scalar($1), scalar($3)); }
```

What this does is creates a new binary op, and feeds it a number of variables. The variables refer to the tokens: $1 is the first token in the input, $2 the second, and so on - think regular expression backreferences. $$ is the op returned from this reduction. So, we call newBINOP to create a new binary operator. The first parameter to newBINOP, a function in *op.c*, is the op type. It's an addition operator, so we want the type to be ADDOP. We could specify this directly, but it's right there as the second token in the input, so we use $2. The second parameter is the op's flags: 0 means "nothing special". Then the things to add: the left and right hand side of our expression, in scalar context.

Stacks

When perl executes something like `addop`, how does it pass on its results to the next op? The answer is, through the use of stacks. Perl has a number of stacks to store things it's currently working on, and we'll look at the three most important ones here.

Argument stack

Arguments are passed to PP code and returned from PP code using the argument stack, ST. The typical way to handle arguments is to pop them off the stack, deal with them how you wish, and then push the result back onto the stack. This is how, for instance, the cosine operator works:

```
NV value;
value = POPn;
value = Perl_cos(value);
XPUSHn(value);
```

We'll see a more tricky example of this when we consider Perl's macros below. POPn gives you the NV (floating point value) of the top SV on the stack: the $x in `cos($x)`. Then we compute the cosine, and push the result back as an NV. The X in XPUSHn means that the stack should be extended if necessary - it can't be necessary here, because we know there's room for one more item on the stack, since we've just removed one! The XPUSH* macros at least guarantee safety.

Alternatively, you can fiddle with the stack directly: SP gives you the first element in your portion of the stack, and TOP* gives you the top SV/IV/NV/etc. on the stack. So, for instance, to do unary negation of an integer:

```
SETi(-TOPi);
```

Just set the integer value of the top stack entry to its negation.

Argument stack manipulation in the core is exactly the same as it is in XSUBs - see *perlxstut*, *perlxs* and *perlguts* for a longer description of the macros used in stack manipulation.

Mark stack

I say "your portion of the stack" above because PP code doesn't necessarily get the whole stack to itself: if your function calls another function, you'll only want to expose the arguments aimed for the called function, and not (necessarily) let it get at your own data. The way we do this is to have a "virtual" bottom-of-stack, exposed to each function. The mark stack keeps bookmarks to locations in the argument stack usable by each function. For instance, when dealing with a tied variable, (internally, something with "P" magic) Perl has to call methods for accesses to the tied variables. However, we need to separate the arguments exposed to the method to the argument exposed to the original function - the store or fetch or whatever it may be. Here's how the tied `push` is implemented; see `av_push` in *av.c*:

```
1 PUSHMARK(SP);
2 EXTEND(SP,2);
3 PUSHs(SvTIED_obj((SV*)av, mg));
4 PUSHs(val);
5 PUTBACK;
6 ENTER;
7 call_method("PUSH", G_SCALAR|G_DISCARD);
8 LEAVE;
9 POPSTACK;
```

The lines which concern the mark stack are the first, fifth and last lines: they save away, restore and remove the current position of the argument stack.

Let's examine the whole implementation, for practice:

```
1 PUSHMARK(SP);
```

Push the current state of the stack pointer onto the mark stack. This is so that when we've finished adding items to the argument stack, Perl knows how many things we've added recently.

```
2 EXTEND(SP,2);
3 PUSHs(SvTIED_obj((SV*)av, mg));
4 PUSHs(val);
```

We're going to add two more items onto the argument stack: when you have a tied array, the `PUSH` subroutine receives the object and the value to be pushed, and that's exactly what we have here - the tied object, retrieved with `SvTIED_obj`, and the value, the SV `val`.

```
5 PUTBACK;
```

Next we tell Perl to make the change to the global stack pointer: `dSP` only gave us a local copy, not a reference to the global.

```
6 ENTER;
7 call_method("PUSH", G_SCALAR|G_DISCARD);
8 LEAVE;
```

`ENTER` and `LEAVE` localise a block of code - they make sure that all variables are tidied up, everything that has been localised gets its previous value returned, and so on. Think of them as the `{` and `}` of a Perl block.

To actually do the magic method call, we have to call a subroutine in Perl space: `call_method` takes care of that, and it's described in *perlcall*. We call the `PUSH` method in scalar context, and we're going to discard its return value.

```
9 POPSTACK;
```

Finally, we remove the value we placed on the mark stack, since we don't need it any more.

Save stack

C doesn't have a concept of local scope, so perl provides one. We've seen that ENTER and LEAVE are used as scoping braces; the save stack implements the C equivalent of, for example:

```
{
    local $foo = 42;
    ...
}
```

See *"Localising Changes" in perlguts* for how to use the save stack.

Millions of Macros

One thing you'll notice about the Perl source is that it's full of macros. Some have called the pervasive use of macros the hardest thing to understand, others find it adds to clarity. Let's take an example, the code which implements the addition operator:

```
1  PP(pp_add)
2  {
3      dSP; dATARGET; tryAMAGICbin(add,opASSIGN);
4      {
5        dPOPTOPnnrl_ul;
6        SETn( left + right );
7        RETURN;
8      }
9  }
```

Every line here (apart from the braces, of course) contains a macro. The first line sets up the function declaration as Perl expects for PP code; line 3 sets up variable declarations for the argument stack and the target, the return value of the operation. Finally, it tries to see if the addition operation is overloaded; if so, the appropriate subroutine is called.

Line 5 is another variable declaration - all variable declarations start with d - which pops from the top of the argument stack two NVs (hence nn) and puts them into the variables `right` and `left`, hence the `rl`. These are the two operands to the addition operator. Next, we call SETn to set the NV of the return value to the result of adding the two values. This done, we return - the RETURN macro makes sure that our return value is properly handled, and we pass the next operator to run back to the main run loop.

Most of these macros are explained in *perlapi*, and some of the more important ones are explained in *perlxs* as well. Pay special attention to *"Background and PERL_IMPLICIT_CONTEXT" in perlguts* for information on the `[pad]THX_?` macros.

The .i Targets

You can expand the macros in a *foo.c* file by saying

```
make foo.i
```

which will expand the macros using cpp. Don't be scared by the results.

Poking at Perl

To really poke around with Perl, you'll probably want to build Perl for debugging, like this:

```
./Configure -d -D optimize=-g
make
```

`-g` is a flag to the C compiler to have it produce debugging information which will allow us to step through a running program. *Configure* will also turn on the `DEBUGGING` compilation symbol which enables all the internal debugging code in Perl. There are a whole bunch of things you can debug with this: *perlrun* lists them all, and the best way to find out about them is to play about with them. The most useful options are probably

```
l  Context (loop) stack processing
t  Trace execution
o  Method and overloading resolution
c  String/numeric conversions
```

Some of the functionality of the debugging code can be achieved using XS modules.

```
-Dr => use re 'debug'
-Dx => use O 'Debug'
```

Using a source-level debugger

If the debugging output of `-D` doesn't help you, it's time to step through perl's execution with a source-level debugger.

- We'll use `gdb` for our examples here; the principles will apply to any debugger, but check the manual of the one you're using.

To fire up the debugger, type

```
gdb ./perl
```

You'll want to do that in your Perl source tree so the debugger can read the source code. You should see the copyright message, followed by the prompt.

```
(gdb)
```

`help` will get you into the documentation, but here are the most useful commands:

run [args]

Run the program with the given arguments.

break function_name

break source.c:xxx

Tells the debugger that we'll want to pause execution when we reach either the named function (but see *"Internal Functions" in perlguts*!) or the given line in the named source file.

step

Steps through the program a line at a time.

next

Steps through the program a line at a time, without descending into functions.

continue

Run until the next breakpoint.

finish

Run until the end of the current function, then stop again.

'enter'

Just pressing Enter will do the most recent operation again - it's a blessing when stepping

through miles of source code.

print

Execute the given C code and print its results. **WARNING**: Perl makes heavy use of macros, and *gdb* does not necessarily support macros (see later *gdb macro support*). You'll have to substitute them yourself, or to invoke cpp on the source code files (see *The .i Targets*) So, for instance, you can't say

```
print SvPV_nolen(sv)
```

but you have to say

```
print Perl_sv_2pv_nolen(sv)
```

You may find it helpful to have a "macro dictionary", which you can produce by saying `cpp -dM perl.c | sort`. Even then, *cpp* won't recursively apply those macros for you.

gdb macro support

Recent versions of *gdb* have fairly good macro support, but in order to use it you'll need to compile perl with macro definitions included in the debugging information. Using *gcc* version 3.1, this means configuring with `-Doptimize=-g3`. Other compilers might use a different switch (if they support debugging macros at all).

Dumping Perl Data Structures

One way to get around this macro hell is to use the dumping functions in *dump.c*; these work a little like an internal *Devel::Peek*, but they also cover OPs and other structures that you can't get at from Perl. Let's take an example. We'll use the `$a = $b + $c` we used before, but give it a bit of context: `$b = "6XXXX"; $c = 2.3;`. Where's a good place to stop and poke around?

What about `pp_add`, the function we examined earlier to implement the + operator:

```
(gdb) break Perl_pp_add
Breakpoint 1 at 0x46249f: file pp_hot.c, line 309.
```

Notice we use `Perl_pp_add` and not `pp_add` - see *"Internal Functions" in perlguts*. With the breakpoint in place, we can run our program:

```
(gdb) run -e '$b = "6XXXX"; $c = 2.3; $a = $b + $c'
```

Lots of junk will go past as gdb reads in the relevant source files and libraries, and then:

```
Breakpoint 1, Perl_pp_add () at pp_hot.c:309
309         dSP; dATARGET; tryAMAGICbin(add,opASSIGN);
(gdb) step
311           dPOPTOPnnrl_ul;
(gdb)
```

We looked at this bit of code before, and we said that `dPOPTOPnnrl_ul` arranges for two `NV`s to be placed into `left` and `right` - let's slightly expand it:

```
#define dPOPTOPnnrl_ul  NV right = POPn; \
                        SV *leftsv = TOPs; \
                        NV left = USE_LEFT(leftsv) ? SvNV(leftsv) : 0.0
```

`POPn` takes the SV from the top of the stack and obtains its NV either directly (if `SvNOK` is set) or by calling the `sv_2nv` function. `TOPs` takes the next SV from the top of the stack - yes, `POPn` uses `TOPs` - but doesn't remove it. We then use `SvNV` to get the NV from `leftsv` in the same way as before -

yes, POPn uses SvNV.

Since we don't have an NV for $b, we'll have to use sv_2nv to convert it. If we step again, we'll find ourselves there:

```
Perl_sv_2nv (sv=0xa0675d0) at sv.c:1669
1669        if (!sv)
(gdb)
```

We can now use Perl_sv_dump to investigate the SV:

```
SV = PV(0xa057cc0) at 0xa0675d0
REFCNT = 1
FLAGS = (POK,pPOK)
PV = 0xa06a510 "6XXXX"\0
CUR = 5
LEN = 6
$1 = void
```

We know we're going to get 6 from this, so let's finish the subroutine:

```
(gdb) finish
Run till exit from #0  Perl_sv_2nv (sv=0xa0675d0) at sv.c:1671
0x462669 in Perl_pp_add () at pp_hot.c:311
311           dPOPTOPnnrl_ul;
```

We can also dump out this op: the current op is always stored in PL_op, and we can dump it with Perl_op_dump. This'll give us similar output to *B::Debug*.

```
{
13  TYPE = add  ===> 14
    TARG = 1
    FLAGS = (SCALAR,KIDS)
    {
        TYPE = null  ===> (12)
          (was rv2sv)
        FLAGS = (SCALAR,KIDS)
        {
11          TYPE = gvsv  ===> 12
            FLAGS = (SCALAR)
            GV = main::b
        }
    }
```

finish this later

Patching

All right, we've now had a look at how to navigate the Perl sources and some things you'll need to know when fiddling with them. Let's now get on and create a simple patch. Here's something Larry suggested: if a U is the first active format during a pack, (for example, `pack "U3C8", @stuff`) then the resulting string should be treated as UTF-8 encoded.

How do we prepare to fix this up? First we locate the code in question - the pack happens at runtime, so it's going to be in one of the *pp* files. Sure enough, pp_pack is in *pp.c*. Since we're going to be altering this file, let's copy it to *pp.c~*.

[Well, it was in *pp.c* when this tutorial was written. It has now been split off with pp_unpack to its own

file, *pp_pack.c*]

Now let's look over `pp_pack`: we take a pattern into `pat`, and then loop over the pattern, taking each format character in turn into `datum_type`. Then for each possible format character, we swallow up the other arguments in the pattern (a field width, an asterisk, and so on) and convert the next chunk input into the specified format, adding it onto the output SV `cat`.

How do we know if the `U` is the first format in the `pat`? Well, if we have a pointer to the start of `pat` then, if we see a `U` we can test whether we're still at the start of the string. So, here's where `pat` is set up:

```
    STRLEN fromlen;
    register char *pat = SvPVx(*++MARK, fromlen);
    register char *patend = pat + fromlen;
    register I32 len;
    I32 datumtype;
    SV *fromstr;
```

We'll have another string pointer in there:

```
    STRLEN fromlen;
    register char *pat = SvPVx(*++MARK, fromlen);
    register char *patend = pat + fromlen;
 +  char *patcopy;
    register I32 len;
    I32 datumtype;
    SV *fromstr;
```

And just before we start the loop, we'll set `patcopy` to be the start of `pat`:

```
    items = SP - MARK;
    MARK++;
    sv_setpvn(cat, "", 0);
 +  patcopy = pat;
    while (pat < patend) {
```

Now if we see a `U` which was at the start of the string, we turn on the `UTF8` flag for the output SV, `cat`:

```
 +  if (datumtype == 'U' && pat==patcopy+1)
 +      SvUTF8_on(cat);
    if (datumtype == '#') {
        while (pat < patend && *pat != '\n')
            pat++;
```

Remember that it has to be `patcopy+1` because the first character of the string is the `U` which has been swallowed into `datumtype`!

Oops, we forgot one thing: what if there are spaces at the start of the pattern? `pack("  U*", @stuff)` will have `U` as the first active character, even though it's not the first thing in the pattern. In this case, we have to advance `patcopy` along with `pat` when we see spaces:

```
    if (isSPACE(datumtype))
        continue;
```

needs to become

```
    if (isSPACE(datumtype)) {
        patcopy++;
        continue;
    }
```

OK. That's the C part done. Now we must do two additional things before this patch is ready to go: we've changed the behaviour of Perl, and so we must document that change. We must also provide some more regression tests to make sure our patch works and doesn't create a bug somewhere else along the line.

The regression tests for each operator live in *t/op/*, and so we make a copy of *t/op/pack.t* to *t/op/pack.t~*. Now we can add our tests to the end. First, we'll test that the `U` does indeed create Unicode strings.

t/op/pack.t has a sensible ok() function, but if it didn't we could use the one from t/test.pl.

```
 require './test.pl';
 plan( tests => 159 );
```

so instead of this:

```
 print 'not ' unless "1.20.300.4000" eq sprintf "%vd",
pack("U*",1,20,300,4000);
 print "ok $test\n"; $test++;
```

we can write the more sensible (see *Test::More* for a full explanation of is() and other testing functions).

```
 is( "1.20.300.4000", sprintf "%vd", pack("U*",1,20,300,4000),
                                     "U* produces unicode" );
```

Now we'll test that we got that space-at-the-beginning business right:

```
 is( "1.20.300.4000", sprintf "%vd", pack("  U*",1,20,300,4000),
                                     "  with spaces at the beginning" );
```

And finally we'll test that we don't make Unicode strings if `U` is **not** the first active format:

```
 isnt( v1.20.300.4000, sprintf "%vd", pack("C0U*",1,20,300,4000),
                                      "U* not first isn't unicode" );
```

Mustn't forget to change the number of tests which appears at the top, or else the automated tester will get confused. This will either look like this:

```
 print "1..156\n";
```

or this:

```
 plan( tests => 156 );
```

We now compile up Perl, and run it through the test suite. Our new tests pass, hooray!

Finally, the documentation. The job is never done until the paperwork is over, so let's describe the change we've just made. The relevant place is *pod/perlfunc.pod*; again, we make a copy, and then we'll insert this text in the description of `pack`:

```
 =item *
```

```
 If the pattern begins with a C<U>, the resulting string will be treated
 as UTF-8-encoded Unicode. You can force UTF-8 encoding on in a string
 with an initial C<U0>, and the bytes that follow will be interpreted as
 Unicode characters. If you don't want this to happen, you can begin your
 pattern with C<C0> (or anything else) to force Perl not to UTF-8 encode
your
 string, and then follow this with a C<U*> somewhere in your pattern.
```

All done. Now let's create the patch. *Porting/patching.pod* tells us that if we're making major changes, we should copy the entire directory to somewhere safe before we begin fiddling, and then do

```
diff -ruN old new > patch
```

However, we know which files we've changed, and we can simply do this:

```
diff -u pp.c~             pp.c             >  patch
diff -u t/op/pack.t~      t/op/pack.t      >> patch
diff -u pod/perlfunc.pod~ pod/perlfunc.pod >> patch
```

We end up with a patch looking a little like this:

```
--- pp.c~       Fri Jun 02 04:34:10 2000
+++ pp.c        Fri Jun 16 11:37:25 2000
@@ -4375,6 +4375,7 @@
     register I32 items;
     STRLEN fromlen;
     register char *pat = SvPVx(*++MARK, fromlen);
+    char *patcopy;
     register char *patend = pat + fromlen;
     register I32 len;
     I32 datumtype;
@@ -4405,6 +4406,7 @@
...
```

And finally, we submit it, with our rationale, to perl5-porters. Job done!

Patching a core module

This works just like patching anything else, with an extra consideration. Many core modules also live on CPAN. If this is so, patch the CPAN version instead of the core and send the patch off to the module maintainer (with a copy to p5p). This will help the module maintainer keep the CPAN version in sync with the core version without constantly scanning p5p.

The list of maintainers of core modules is usefully documented in *Porting/Maintainers.pl*.

Adding a new function to the core

If, as part of a patch to fix a bug, or just because you have an especially good idea, you decide to add a new function to the core, discuss your ideas on p5p well before you start work. It may be that someone else has already attempted to do what you are considering and can give lots of good advice or even provide you with bits of code that they already started (but never finished).

You have to follow all of the advice given above for patching. It is extremely important to test any addition thoroughly and add new tests to explore all boundary conditions that your new function is expected to handle. If your new function is used only by one module (e.g. toke), then it should probably be named S_your_function (for static); on the other hand, if you expect it to accessible from other functions in Perl, you should name it Perl_your_function. See *"Internal Functions" in perlguts* for more details.

The location of any new code is also an important consideration. Don't just create a new top level .c file and put your code there; you would have to make changes to Configure (so the Makefile is created properly), as well as possibly lots of include files. This is strictly pumpking business.

It is better to add your function to one of the existing top level source code files, but your choice is complicated by the nature of the Perl distribution. Only the files that are marked as compiled static are located in the perl executable. Everything else is located in the shared library (or DLL if you are running under WIN32). So, for example, if a function was only used by functions located in toke.c, then your code can go in toke.c. If, however, you want to call the function from universal.c, then you should put your code in another location, for example util.c.

In addition to writing your c-code, you will need to create an appropriate entry in embed.pl describing your function, then run 'make regen_headers' to create the entries in the numerous header files that perl needs to compile correctly. See *"Internal Functions" in perlguts* for information on the various options that you can set in embed.pl. You will forget to do this a few (or many) times and you will get warnings during the compilation phase. Make sure that you mention this when you post your patch to P5P; the pumpking needs to know this.

When you write your new code, please be conscious of existing code conventions used in the perl source files. See *perlstyle* for details. Although most of the guidelines discussed seem to focus on Perl code, rather than c, they all apply (except when they don't ;). See also *Porting/patching.pod* file in the Perl source distribution for lots of details about both formatting and submitting patches of your changes.

Lastly, TEST TEST TEST TEST TEST any code before posting to p5p. Test on as many platforms as you can find. Test as many perl Configure options as you can (e.g. MULTIPLICITY). If you have profiling or memory tools, see *EXTERNAL TOOLS FOR DEBUGGING PERL* below for how to use them to further test your code. Remember that most of the people on P5P are doing this on their own time and don't have the time to debug your code.

Writing a test

Every module and built-in function has an associated test file (or should...). If you add or change functionality, you have to write a test. If you fix a bug, you have to write a test so that bug never comes back. If you alter the docs, it would be nice to test what the new documentation says.

In short, if you submit a patch you probably also have to patch the tests.

For modules, the test file is right next to the module itself. *lib/strict.t* tests *lib/strict.pm*. This is a recent innovation, so there are some snags (and it would be wonderful for you to brush them out), but it basically works that way. Everything else lives in *t/*.

t/base/

Testing of the absolute basic functionality of Perl. Things like `if`, basic file reads and writes, simple regexes, etc. These are run first in the test suite and if any of them fail, something is *really* broken.

t/cmd/

These test the basic control structures, `if/else`, `while`, subroutines, etc.

t/comp/

Tests basic issues of how Perl parses and compiles itself.

t/io/

Tests for built-in IO functions, including command line arguments.

t/lib/

The old home for the module tests, you shouldn't put anything new in here. There are still some bits and pieces hanging around in here that need to be moved. Perhaps you could move them?

t/op/ Thanks!

Tests for perl's built in functions that don't fit into any of the other directories.

t/pod/

Tests for POD directives. There are still some tests for the Pod modules hanging around in here that need to be moved out into *lib/*.

t/run/

Testing features of how perl actually runs, including exit codes and handling of PERL* environment variables.

t/uni/

Tests for the core support of Unicode.

t/win32/

Windows-specific tests.

t/x2p

A test suite for the s2p converter.

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.

t/base t/comp

Since we don't know if require works, or even subroutines, use ad hoc tests for these two. Step carefully to avoid using the feature being tested.

t/cmd t/run t/io t/op

Now that basic require() and subroutines are tested, you can use the t/test.pl library which emulates the important features of Test::More while using a minimum of core features.

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

t/lib ext lib

Now that the core of Perl is tested, Test::More can 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 and avoiding things like `fork()` and `system()` unless absolutely necessary.

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.

coretest

Run *perl* on all core tests (*t/** and *lib/[a-z]** pragma tests).

(Not available on Win32)

test.deparse

Run all the tests through B::Deparse. Not all tests will succeed.

(Not available on Win32)

test.taintwarn

Run all tests with the **-t** command-line switch. Not all tests are expected to succeed (until they're specifically fixed, of course).

(Not available on Win32)

minitest

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

test.valgrind check.valgrind utest.valgrind ucheck.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.third check.third utest.third ucheck.third

(Only in Tru64) Run all the tests using the memory leak + naughty memory access tool "Third Degree". The log files will be named *perl.3log.testname*.

test.torture torturetest

Run all the usual tests and some extra tests. As of Perl 5.8.0 the only extra tests are Abigail's JAPHs, *t/japh/abigail.t*.

You can also run the torture test with *t/harness* by giving `-torture` argument to *t/harness*.

utest ucheck test.utf8 check.utf8

Run all the tests with -Mutf8. Not all tests will succeed.

(Not available on Win32)

minitest.utf16 test.utf16

Runs the tests with UTF-16 encoded scripts, encoded with different versions of this encoding.

`make utest.utf16` runs the test suite with a combination of `-utf8` and `-utf16` arguments to *t/TEST*.

(Not available on Win32)

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.

Running tests by hand

You can run part of the test suite by hand by using one 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 outbut.

-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 patho/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 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*)

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'.

EXTERNAL TOOLS FOR DEBUGGING PERL

Sometimes it helps to use external tools while debugging and testing Perl. This section tries to guide you through using some common testing and debugging tools with Perl. This is meant as a guide to interfacing these tools with Perl, not as any kind of guide to the use of the tools themselves.

NOTE 1: Running under memory debuggers such as Purify, valgrind, or Third Degree greatly slows down the execution: seconds become minutes, minutes become hours. For example as of Perl 5.8.1, the ext/Encode/t/Unicode.t takes extraordinarily long to complete under e.g. Purify, Third Degree, and valgrind. Under valgrind it takes more than six hours, even on a snappy computer-- the said test must be doing something that is quite unfriendly for memory debuggers. If you don't feel like waiting, that you can simply kill away the perl process.

NOTE 2: To minimize the number of memory leak false alarms (see *PERL_DESTRUCT_LEVEL* for more information), you have to have environment variable PERL_DESTRUCT_LEVEL set to 2. The *TEST* and harness scripts do that automatically. But if you are running some of the tests manually-- for csh-like shells:

```
setenv PERL_DESTRUCT_LEVEL 2
```

and for Bourne-type shells:

```
PERL_DESTRUCT_LEVEL=2
export PERL_DESTRUCT_LEVEL
```

or in UNIXy environments you can also use the `env` command:

```
env PERL_DESTRUCT_LEVEL=2 valgrind ./perl -Ilib ...
```

NOTE 3: There are known memory leaks when there are compile-time errors within eval or require, seeing `S_doeval` in the call stack is a good sign of these. Fixing these leaks is non-trivial, unfortunately, but they must be fixed eventually.

Rational Software's Purify

Purify is a commercial tool that is helpful in identifying memory overruns, wild pointers, memory leaks and other such badness. Perl must be compiled in a specific way for optimal testing with Purify. Purify is available under Windows NT, Solaris, HP-UX, SGI, and Siemens Unix.

Purify on Unix

On Unix, Purify creates a new Perl binary. To get the most benefit out of Purify, you should create the perl to Purify using:

```
sh Configure -Accflags=-DPURIFY -Doptimize='-g' \
 -Uusemymalloc -Dusemultiplicity
```

where these arguments mean:

-Accflags=-DPURIFY

Disables Perl's arena memory allocation functions, as well as forcing use of memory allocation functions derived from the system malloc.

-Doptimize='-g'

Adds debugging information so that you see the exact source statements where the problem occurs. Without this flag, all you will see is the source filename of where the error occurred.

-Uusemymalloc

Disable Perl's malloc so that Purify can more closely monitor allocations and leaks. Using Perl's malloc will make Purify report most leaks in the "potential" leaks category.

-Dusemultiplicity

Enabling the multiplicity option allows perl to clean up thoroughly when the interpreter shuts down, which reduces the number of bogus leak reports from Purify.

Once you've compiled a perl suitable for Purify'ing, then you can just:

```
make pureperl
```

which creates a binary named 'pureperl' that has been Purify'ed. This binary is used in place of the standard 'perl' binary when you want to debug Perl memory problems.

As an example, to show any memory leaks produced during the standard Perl testset you would create and run the Purify'ed perl as:

```
make pureperl
cd t
../pureperl -I../lib harness
```

which would run Perl on test.pl and report any memory problems.

Purify outputs messages in "Viewer" windows by default. If you don't have a windowing environment or if you simply want the Purify output to unobtrusively go to a log file instead of to the interactive window, use these following options to output to the log file "perl.log":

```
setenv PURIFYOPTIONS "-chain-length=25 -windows=no \
 -log-file=perl.log -append-logfile=yes"
```

If you plan to use the "Viewer" windows, then you only need this option:

```
setenv PURIFYOPTIONS "-chain-length=25"
```

In Bourne-type shells:

```
PURIFYOPTIONS="..."
export PURIFYOPTIONS
```

or if you have the "env" utility:

```
env PURIFYOPTIONS="..." ../pureperl ...
```

Purify on NT

Purify on Windows NT instruments the Perl binary 'perl.exe' on the fly. There are several options in the makefile you should change to get the most use out of Purify:

DEFINES

You should add -DPURIFY to the DEFINES line so the DEFINES line looks something like:

```
DEFINES = -DWIN32 -D_CONSOLE -DNO_STRICT $(CRYPT_FLAG) -DPURIFY=1
```

to disable Perl's arena memory allocation functions, as well as to force use of memory allocation functions derived from the system malloc.

USE_MULTI = define

Enabling the multiplicity option allows perl to clean up thoroughly when the interpreter shuts down, which reduces the number of bogus leak reports from Purify.

#PERL_MALLOC = define

Disable Perl's malloc so that Purify can more closely monitor allocations and leaks. Using Perl's malloc will make Purify report most leaks in the "potential" leaks category.

CFG = Debug

Adds debugging information so that you see the exact source statements where the problem occurs. Without this flag, all you will see is the source filename of where the error occurred.

As an example, to show any memory leaks produced during the standard Perl testset you would create and run Purify as:

```
cd win32
make
cd ../t
purify ../perl -I../lib harness
```

which would instrument Perl in memory, run Perl on test.pl, then finally report any memory problems.

valgrind

The excellent valgrind tool can be used to find out both memory leaks and illegal memory accesses. As of August 2003 it unfortunately works only on x86 (ELF) Linux. The special "test.valgrind" target can be used to run the tests under valgrind. Found errors and memory leaks are logged in files named *test.valgrind*.

As system libraries (most notably glibc) are also triggering errors, valgrind allows to suppress such errors using suppression files. The default suppression file that comes with valgrind already catches a lot of them. Some additional suppressions are defined in *t/perl.supp*.

To get valgrind and for more information see

```
http://developer.kde.org/~sewardj/
```

Compaq's/Digital's/HP's Third Degree

Third Degree is a tool for memory leak detection and memory access checks. It is one of the many tools in the ATOM toolkit. The toolkit is only available on Tru64 (formerly known as Digital UNIX formerly known as DEC OSF/1).

When building Perl, you must first run Configure with -Doptimize=-g and -Uusemymalloc flags, after that you can use the make targets "perl.third" and "test.third". (What is required is that Perl must be compiled using the `-g` flag, you may need to re-Configure.)

The short story is that with "atom" you can instrument the Perl executable to create a new executable called *perl.third*. When the instrumented executable is run, it creates a log of dubious memory traffic in file called *perl.3log*. See the manual pages of atom and third for more information. The most extensive Third Degree documentation is available in the Compaq "Tru64 UNIX Programmer's Guide", chapter "Debugging Programs with Third Degree".

The "test.third" leaves a lot of files named *foo_bar.3log* in the t/ subdirectory. There is a problem with these files: Third Degree is so effective that it finds problems also in the system libraries. Therefore you should used the Porting/thirdclean script to cleanup the **.3log* files.

There are also leaks that for given certain definition of a leak, aren't. See *PERL_DESTRUCT_LEVEL* for more information.

PERL_DESTRUCT_LEVEL

If you want to run any of the tests yourself manually using e.g. valgrind, or the pureperl or perl.third executables, please note that by default perl **does not** explicitly cleanup all the memory it has allocated (such as global memory arenas) but instead lets the exit() of the whole program "take care" of such allocations, also known as "global destruction of objects".

There is a way to tell perl to do complete cleanup: set the environment variable PERL_DESTRUCT_LEVEL to a non-zero value. The t/TEST wrapper does set this to 2, and this is what you need to do too, if you don't want to see the "global leaks": For example, for "third-degreed" Perl:

```
env PERL_DESTRUCT_LEVEL=2 ./perl.third -Ilib t/foo/bar.t
```

(Note: the mod_perl apache module uses also this environment variable for its own purposes and extended its semantics. Refer to the mod_perl documentation for more information. Also, spawned threads do the equivalent of setting this variable to the value 1.)

If, at the end of a run you get the message *N scalars leaked*, you can recompile with `-DDEBUG_LEAKING_SCALARS`, which will cause the addresses of all those leaked SVs to be dumped; it also converts `new_SV()` from a macro into a real function, so you can use your favourite debugger to discover where those pesky SVs were allocated.

Profiling

Depending on your platform there are various of profiling Perl.

There are two commonly used techniques of profiling executables: *statistical time-sampling* and *basic-block counting*.

The first method takes periodically samples of the CPU program counter, and since the program counter can be correlated with the code generated for functions, we get a statistical view of in which functions the program is spending its time. The caveats are that very small/fast functions have lower probability of showing up in the profile, and that periodically interrupting the program (this is usually done rather frequently, in the scale of milliseconds) imposes an additional overhead that may skew the results. The first problem can be alleviated by running the code for longer (in general this is a good idea for profiling), the second problem is usually kept in guard by the profiling tools themselves.

The second method divides up the generated code into *basic blocks*. Basic blocks are sections of code that are entered only in the beginning and exited only at the end. For example, a conditional jump starts a basic block. Basic block profiling usually works by *instrumenting* the code by adding *enter basic block #nnnn* book-keeping code to the generated code. During the execution of the code the basic block counters are then updated appropriately. The caveat is that the added extra code can skew the results: again, the profiling tools usually try to factor their own effects out of the results.

Gprof Profiling

gprof is a profiling tool available in many UNIX platforms, it uses *statistical time-sampling*.

You can build a profiled version of perl called "perl.gprof" by invoking the make target "perl.gprof" (What is required is that Perl must be compiled using the `-pg` flag, you may need to re-Configure). Running the profiled version of Perl will create an output file called *gmon.out* is created which contains the profiling data collected during the execution.

The gprof tool can then display the collected data in various ways. Usually gprof understands the following options:

-a

Suppress statically defined functions from the profile.

-b

Suppress the verbose descriptions in the profile.

-e routine

Exclude the given routine and its descendants from the profile.

-f routine

Display only the given routine and its descendants in the profile.

-s

Generate a summary file called *gmon.sum* which then may be given to subsequent gprof runs to accumulate data over several runs.

-z

Display routines that have zero usage.

For more detailed explanation of the available commands and output formats, see your own local documentation of gprof.

GCC gcov Profiling

Starting from GCC 3.0 *basic block profiling* is officially available for the GNU CC.

You can build a profiled version of perl called *perl.gcov* by invoking the make target "perl.gcov" (what is required that Perl must be compiled using gcc with the flags `-fprofile-arcs -ftest-coverage`, you may need to re-Configure).

Running the profiled version of Perl will cause profile output to be generated. For each source file an accompanying ".da" file will be created.

To display the results you use the "gcov" utility (which should be installed if you have gcc 3.0 or newer installed). *gcov* is run on source code files, like this

```
gcov sv.c
```

which will cause *sv.c.gcov* to be created. The *.gcov* files contain the source code annotated with relative frequencies of execution indicated by "#" markers.

Useful options of *gcov* include `-b` which will summarise the basic block, branch, and function call coverage, and `-c` which instead of relative frequencies will use the actual counts. For more information on the use of *gcov* and basic block profiling with gcc, see the latest GNU CC manual, as of GCC 3.0 see

```
http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc.html
```

and its section titled "8. gcov: a Test Coverage Program"

```
http://gcc.gnu.org/onlinedocs/gcc-3.0/gcc_8.html#SEC132
```

Pixie Profiling

Pixie is a profiling tool available on IRIX and Tru64 (aka Digital UNIX aka DEC OSF/1) platforms. Pixie does its profiling using *basic-block counting*.

You can build a profiled version of perl called *perl.pixie* by invoking the make target "perl.pixie" (what is required is that Perl must be compiled using the `-g` flag, you may need to re-Configure).

In Tru64 a file called *perl.Addrs* will also be silently created, this file contains the addresses of the basic blocks. Running the profiled version of Perl will create a new file called "perl.Counts" which contains the counts for the basic block for that particular program execution.

To display the results you use the *prof* utility. The exact incantation depends on your operating system, "prof perl.Counts" in IRIX, and "prof -pixie -all -L. perl" in Tru64.

In IRIX the following prof options are available:

-h

Reports the most heavily used lines in descending order of use. Useful for finding the hotspot lines.

-l

Groups lines by procedure, with procedures sorted in descending order of use. Within a procedure, lines are listed in source order. Useful for finding the hotspots of procedures.

In Tru64 the following options are available:

-p[rocedures]

Procedures sorted in descending order by the number of cycles executed in each procedure. Useful for finding the hotspot procedures. (This is the default option.)

-h[eavy]

Lines sorted in descending order by the number of cycles executed in each line. Useful for finding the hotspot lines.

-i[nvocations]

The called procedures are sorted in descending order by number of calls made to the procedures. Useful for finding the most used procedures.

-l[ines]

Grouped by procedure, sorted by cycles executed per procedure. Useful for finding the hotspots of procedures.

-testcoverage

The compiler emitted code for these lines, but the code was unexecuted.

-z[ero]

Unexecuted procedures.

For further information, see your system's manual pages for pixie and prof.

Miscellaneous tricks

- Those debugging perl with the DDD frontend over gdb may find the following useful:

  You can extend the data conversion shortcuts menu, so for example you can display an SV's IV value with one click, without doing any typing. To do that simply edit ~/.ddd/init file and add after:

```
! Display shortcuts.
Ddd*gdbDisplayShortcuts: \
/t ()   // Convert to Bin\n\
/d ()   // Convert to Dec\n\
/x ()   // Convert to Hex\n\
/o ()   // Convert to Oct(\n\
```

  the following two lines:

```
((XPV*) (())->sv_any )->xpv_pv  // 2pvx\n\
((XPVIV*) (())->sv_any )->xiv_iv // 2ivx
```

  so now you can do ivx and pvx lookups or you can plug there the sv_peek "conversion":

```
Perl_sv_peek(my_perl, (SV*)()) // sv_peek
```

(The my_perl is for threaded builds.) Just remember that every line, but the last one, should end with \n\

Alternatively edit the init file interactively via: 3rd mouse button -> New Display -> Edit Menu

Note: you can define up to 20 conversion shortcuts in the gdb section.

- If you see in a debugger a memory area mysteriously full of 0xabababab, you may be seeing the effect of the Poison() macro, see *perlclib*.

CONCLUSION

We've had a brief look around the Perl source, an overview of the stages *perl* goes through when it's running your code, and how to use a debugger to poke at the Perl guts. We took a very simple problem and demonstrated how to solve it fully - with documentation, regression tests, and finally a patch for submission to p5p. Finally, we talked about how to use external tools to debug and test Perl.

I'd now suggest you read over those references again, and then, as soon as possible, get your hands dirty. The best way to learn is by doing, so:

- 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...
- Keep up to date with the bleeding edge Perl distributions and get familiar with the changes. Try and get an idea of what areas people are working on and the changes they're making.
- 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!

AUTHOR

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