NAME

perldebug - Perl debugging

DESCRIPTION

First of all, have you tried using the **-w** switch?

If you're new to the Perl debugger, you may prefer to read *perldebtut*, which is a tutorial introduction to the debugger .

The Perl Debugger

If you invoke Perl with the **-d** switch, your script runs under the Perl source debugger. This works like an interactive Perl environment, prompting for debugger commands that let you examine source code, set breakpoints, get stack backtraces, change the values of variables, etc. This is so convenient that you often fire up the debugger all by itself just to test out Perl constructs interactively to see what they do. For example:

```
$ perl -d -e 42
```

In Perl, the debugger is not a separate program the way it usually is in the typical compiled environment. Instead, the **-d** flag tells the compiler to insert source information into the parse trees it's about to hand off to the interpreter. That means your code must first compile correctly for the debugger to work on it. Then when the interpreter starts up, it preloads a special Perl library file containing the debugger.

The program will halt *right before* the first run-time executable statement (but see below regarding compile-time statements) and ask you to enter a debugger command. Contrary to popular expectations, whenever the debugger halts and shows you a line of code, it always displays the line it's *about* to execute, rather than the one it has just executed.

Any command not recognized by the debugger is directly executed (`eval`'d) as Perl code in the current package. (The debugger uses the DB package for keeping its own state information.)

Note that the said `eval` is bound by an implicit scope. As a result any newly introduced lexical variable or any modified capture buffer content is lost after the eval. The debugger is a nice environment to learn Perl, but if you interactively experiment using material which should be in the same scope, stuff it in one line.

For any text entered at the debugger prompt, leading and trailing whitespace is first stripped before further processing. If a debugger command coincides with some function in your own program, merely precede the function with something that doesn't look like a debugger command, such as a leading `;` or perhaps a +, or by wrapping it with parentheses or braces.

Debugger Commands

The debugger understands the following commands:

h

Prints out a summary help message

h [command]

Prints out a help message for the given debugger command.

h h

The special argument of h h produces the entire help page, which is quite long.

If the output of the h h command (or any command, for that matter) scrolls past your screen, precede the command with a leading pipe symbol so that it's run through your pager, as in

```
DB> |h h
```

You may change the pager which is used via `o pager=...` command.

p expr

Same as `print {$DB::OUT} expr` in the current package. In particular, because this is just Perl's own `print` function, this means that nested data structures and objects are not dumped, unlike with the `x` command.

The `DB::OUT` filehandle is opened to */dev/tty*, regardless of where STDOUT may be redirected to.

x [maxdepth] expr

Evaluates its expression in list context and dumps out the result in a pretty-printed fashion. Nested data structures are printed out recursively, unlike the real `print` function in Perl. When dumping hashes, you'll probably prefer 'x \%h' rather than 'x %h'. See *Dumpvalue* if you'd like to do this yourself.

The output format is governed by multiple options described under *Configurable Options*.

If the `maxdepth` is included, it must be a numeral *N*; the value is dumped only *N* levels deep, as if the `dumpDepth` option had been temporarily set to *N*.

V [pkg [vars]]

Display all (or some) variables in package (defaulting to `main`) using a data pretty-printer (hashes show their keys and values so you see what's what, control characters are made printable, etc.). Make sure you don't put the type specifier (like $) there, just the symbol names, like this:

```
V DB filename line
```

Use `~pattern` and `!pattern` for positive and negative regexes.

This is similar to calling the `x` command on each applicable var.

X [vars]

Same as `V currentpackage [vars]`.

y [level [vars]]

Display all (or some) lexical variables (mnemonic: `mY` variables) in the current scope or *level* scopes higher. You can limit the variables that you see with *vars* which works exactly as it does for the `V` and `X` commands. Requires the `PadWalker` module version 0.08 or higher; will warn if this isn't installed. Output is pretty-printed in the same style as for `V` and the format is controlled by the same options.

T

Produce a stack backtrace. See below for details on its output.

s [expr]

Single step. Executes until the beginning of another statement, descending into subroutine calls. If an expression is supplied that includes function calls, it too will be single-stepped.

n [expr]

Next. Executes over subroutine calls, until the beginning of the next statement. If an expression is supplied that includes function calls, those functions will be executed with stops before each statement.

r

Continue until the return from the current subroutine. Dump the return value if the `PrintRet` option is set (default).

<CR>

Repeat last `n` or `s` command.

c [line|sub]

Continue, optionally inserting a one-time-only breakpoint at the specified line or subroutine.

l

List next window of lines.

l min+incr

List `incr+1` lines starting at `min`.

l min-max

List lines `min` through `max`. `l -` is synonymous to `-`.

l line

List a single line.

l subname

List first window of lines from subroutine. *subname* may be a variable that contains a code reference.

-

List previous window of lines.

v [line]

View a few lines of code around the current line.

.

Return the internal debugger pointer to the line last executed, and print out that line.

f filename

Switch to viewing a different file or `eval` statement. If *filename* is not a full pathname found in the values of %INC, it is considered a regex.

`eval`ed strings (when accessible) are considered to be filenames: `f (eval 7)` and `f eval 7\b` access the body of the 7th `eval`ed string (in the order of execution). The bodies of the currently executed `eval` and of `eval`ed strings that define subroutines are saved and thus accessible.

/pattern/

Search forwards for pattern (a Perl regex); final / is optional. The search is case-insensitive by default.

?pattern?

Search backwards for pattern; final ? is optional. The search is case-insensitive by default.

L [abw]

List (default all) actions, breakpoints and watch expressions

S [[!]regex]

List subroutine names [not] matching the regex.

t

Toggle trace mode (see also the `AutoTrace` option).

t expr

Trace through execution of `expr`. See *"Frame Listing Output Examples" in perldebguts* for examples.

b

Sets breakpoint on current line

b [line] [condition]

Set a breakpoint before the given line. If a condition is specified, it's evaluated each time the statement is reached: a breakpoint is taken only if the condition is true. Breakpoints may only be set on lines that begin an executable statement. Conditions don't use `if`:

```
b 237 $x > 30
b 237 ++$count237 < 11
b 33 /pattern/i
```

b subname [condition]

Set a breakpoint before the first line of the named subroutine. *subname* may be a variable containing a code reference (in this case *condition* is not supported).

b postpone subname [condition]

Set a breakpoint at first line of subroutine after it is compiled.

b load filename

Set a breakpoint before the first executed line of the *filename*, which should be a full pathname found amongst the %INC values.

b compile subname

Sets a breakpoint before the first statement executed after the specified subroutine is compiled.

B line

Delete a breakpoint from the specified *line*.

B *

Delete all installed breakpoints.

a [line] command

Set an action to be done before the line is executed. If *line* is omitted, set an action on the line about to be executed. The sequence of steps taken by the debugger is

1. check for a breakpoint at this line
2. print the line if necessary (tracing)
3. do any actions associated with that line
4. prompt user if at a breakpoint or in single-step
5. evaluate line

For example, this will print out $foo every time line 53 is passed:

```
a 53 print "DB FOUND $foo\n"
```

A line

Delete an action from the specified line.

A *

Delete all installed actions.

w expr

Add a global watch-expression. We hope you know what one of these is, because they're supposed to be obvious.

W expr

Delete watch-expression

W *

Delete all watch-expressions.

o

Display all options

o booloption ...

Set each listed Boolean option to the value `1`.

o anyoption? ...

Print out the value of one or more options.

o option=value ...

Set the value of one or more options. If the value has internal whitespace, it should be quoted. For example, you could set `o pager="less -MQeicsNfr"` to call **less** with those specific options. You may use either single or double quotes, but if you do, you must escape any embedded instances of same sort of quote you began with, as well as any escaping any escapes that immediately precede that quote but which are not meant to escape the quote itself. In other words, you follow single-quoting rules irrespective of the quote; eg: `o option='this isn\'t bad'` or `o option="She said, \"Isn't it?\""`.

For historical reasons, the `=value` is optional, but defaults to 1 only where it is safe to do so--that is, mostly for Boolean options. It is always better to assign a specific value using `=`. The `option` can be abbreviated, but for clarity probably should not be. Several options can be set together. See *Configurable Options* for a list of these.

< ?

List out all pre-prompt Perl command actions.

< [command]

Set an action (Perl command) to happen before every debugger prompt. A multi-line command may be entered by backslashing the newlines.

< *

Delete all pre-prompt Perl command actions.

<< command

Add an action (Perl command) to happen before every debugger prompt. A

multi-line command may be entered by backwhacking the newlines.

> ?

List out post-prompt Perl command actions.

> command

Set an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines (we bet you couldn't've guessed this by now).

> *

Delete all post-prompt Perl command actions.

>> command

Adds an action (Perl command) to happen after the prompt when you've just given a command to return to executing the script. A multi-line command may be entered by backslashing the newlines.

{ ?

List out pre-prompt debugger commands.

{ [command]

Set an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered in the customary fashion.

Because this command is in some senses new, a warning is issued if you appear to have accidentally entered a block instead. If that's what you mean to do, write it as with `;{ ... }` or even `do { ... }`.

{ *

Delete all pre-prompt debugger commands.

{{ command

Add an action (debugger command) to happen before every debugger prompt. A multi-line command may be entered, if you can guess how: see above.

! number

Redo a previous command (defaults to the previous command).

! -number

Redo number'th previous command.

! pattern

Redo last command that started with pattern. See `o recallCommand`, too.

!! cmd

Run cmd in a subprocess (reads from DB::IN, writes to DB::OUT) See `o shellBang`, also. Note that the user's current shell (well, their `$ENV{SHELL}` variable) will be used, which can interfere with proper interpretation of exit status or signal and coredump information.

source file

Read and execute debugger commands from *file*. *file* may itself contain `source` commands.

H -number

Display last n commands. Only commands longer than one character are listed. If *number* is omitted, list them all.

q or ^D

Quit. ("quit" doesn't work for this, unless you've made an alias) This is the only supported way to exit the debugger, though typing `exit` twice might work.

Set the `inhibit_exit` option to 0 if you want to be able to step off the end the script. You may also need to set $finished to 0 if you want to step through global destruction.

R

Restart the debugger by `exec()`ing a new session. We try to maintain your history across this, but internal settings and command-line options may be lost.

The following setting are currently preserved: history, breakpoints, actions, debugger options, and the Perl command-line options **-w**, **-I**, and **-e**.

|dbcmd

Run the debugger command, piping DB::OUT into your current pager.

||dbcmd

Same as `|dbcmd` but DB::OUT is temporarily `selected` as well.

= [alias value]

Define a command alias, like

```
= quit q
```

or list current aliases.

command

Execute command as a Perl statement. A trailing semicolon will be supplied. If the Perl statement would otherwise be confused for a Perl debugger, use a leading semicolon, too.

m expr

List which methods may be called on the result of the evaluated expression. The expression may evaluated to a reference to a blessed object, or to a package name.

M

Displays all loaded modules and their versions

man [manpage]

Despite its name, this calls your system's default documentation viewer on the given page, or on the viewer itself if *manpage* is omitted. If that viewer is **man**, the current `Config` information is used to invoke **man** using the proper MANPATH or **-M** *manpath* option. Failed lookups of the form XXX that match known manpages of the form *perlXXX* will be retried. This lets you type `man debug` or `man op` from the debugger.

On systems traditionally bereft of a usable **man** command, the debugger invokes **perldoc**. Occasionally this determination is incorrect due to recalcitrant vendors or rather more felicitously, to enterprising users. If you fall into either category, just manually set the $DB::doccmd variable to whatever viewer to view the Perl documentation on your system. This may be set in an rc file, or through direct assignment. We're still waiting for a working example of something along the lines of:

```
    $DB::doccmd = 'netscape -remote
http://something.here/';
```

Configurable Options

The debugger has numerous options settable using the o command, either interactively or from the environment or an rc file. (./.perldb or ~/.perldb under Unix.)

`recallCommand`, `ShellBang`

The characters used to recall command or spawn shell. By default, both are set to `!`, which is unfortunate.

`pager`

Program to use for output of pager-piped commands (those beginning with a `|` character.) By default, `$ENV{PAGER}` will be used. Because the debugger uses your current terminal characteristics for bold and underlining, if the chosen pager does not pass escape sequences through unchanged, the output of some debugger commands will not be readable when sent through the pager.

`tkRunning`

Run Tk while prompting (with ReadLine).

`signalLevel`, `warnLevel`, `dieLevel`

Level of verbosity. By default, the debugger leaves your exceptions and warnings alone, because altering them can break correctly running programs. It will attempt to print a message when uncaught INT, BUS, or SEGV signals arrive. (But see the mention of signals in *BUGS* below.)

To disable this default safe mode, set these values to something higher than 0. At a level of 1, you get backtraces upon receiving any kind of warning (this is often annoying) or exception (this is often valuable). Unfortunately, the debugger cannot discern fatal exceptions from non-fatal ones. If `dieLevel` is even 1, then your non-fatal exceptions are also traced and unceremoniously altered if they came from `eval'd` strings or from any kind of `eval` within modules you're attempting to load. If `dieLevel` is 2, the debugger doesn't care where they came from: It usurps your exception handler and prints out a trace, then modifies all exceptions with its own embellishments. This may perhaps be useful for some tracing purposes, but tends to hopelessly destroy any program that takes its exception handling seriously.

`AutoTrace`

Trace mode (similar to `t` command, but can be put into `PERLDB_OPTS`).

`LineInfo`

File or pipe to print line number info to. If it is a pipe (say, `|visual_perl_db`), then a short message is used. This is the mechanism used to interact with a slave editor or visual debugger, such as the special `vi` or `emacs` hooks, or the `ddd` graphical debugger.

`inhibit_exit`

If 0, allows *stepping off* the end of the script.

`PrintRet`

Print return value after `r` command if set (default).

`ornaments`

Affects screen appearance of the command line (see *Term::ReadLine*). There is currently no way to disable these, which can render some output illegible on some displays, or with some pagers. This is considered a bug.

`frame`

Affects the printing of messages upon entry and exit from subroutines. If `frame & 2` is false, messages are printed on entry only. (Printing on exit might be useful if interspersed with other messages.)

If `frame & 4`, arguments to functions are printed, plus context and caller info. If `frame & 8`, overloaded `stringify` and `tied` `FETCH` is enabled on the printed arguments. If `frame & 16`, the return value from the subroutine is printed.

The length at which the argument list is truncated is governed by the next option:

`maxTraceLen`

Length to truncate the argument list when the `frame` option's bit 4 is set.

`windowSize`

Change the size of code list window (default is 10 lines).

The following options affect what happens with `V`, `X`, and `x` commands:

`arrayDepth`, `hashDepth`

Print only first N elements (" for all).

`dumpDepth`

Limit recursion depth to N levels when dumping structures. Negative values are interpreted as infinity. Default: infinity.

`compactDump`, `veryCompact`

Change the style of array and hash output. If `compactDump`, short array may be printed on one line.

`globPrint`

Whether to print contents of globs.

`DumpDBFiles`

Dump arrays holding debugged files.

`DumpPackages`

Dump symbol tables of packages.

`DumpReused`

Dump contents of "reused" addresses.

`quote`, `HighBit`, `undefPrint`

Change the style of string dump. The default value for `quote` is `auto`; one can enable double-quotish or single-quotish format by setting it to `"` or `'`, respectively. By default, characters with their high bit set are printed verbatim.

`UsageOnly`

Rudimentary per-package memory usage dump. Calculates total size of strings found in variables in the package. This does not include lexicals in a module's file scope, or lost in closures.

After the rc file is read, the debugger reads the `$ENV{PERLDB_OPTS}` environment variable and parses this as the remainder of a "O ..." line as one might enter at the debugger prompt. You may place the initialization options `TTY`, `noTTY`, `ReadLine`, and `NonStop` there.

If your rc file contains:

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace");
```

then your script will run without human intervention, putting trace information into the file *db.out*. (If you interrupt it, you'd better reset `LineInfo` to */dev/tty* if you expect to see anything.)

`TTY`

The TTY to use for debugging I/O.

`noTTY`

If set, the debugger goes into `NonStop` mode and will not connect to a TTY. If interrupted (or if control goes to the debugger via explicit setting of $DB::signal or $DB::single from the Perl script), it connects to a TTY specified in the `TTY` option at startup, or to a tty found at runtime using the `Term::Rendezvous` module of your choice.

This module should implement a method named `new` that returns an object with two methods: `IN` and `OUT`. These should return filehandles to use for debugging input and output correspondingly. The `new` method should inspect an argument containing the value of `$ENV{PERLDB_NOTTY}` at startup, or `"$ENV{HOME}/.perldbtty$$"` otherwise. This file is not inspected for proper ownership, so security hazards are theoretically possible.

`ReadLine`

If false, readline support in the debugger is disabled in order to debug applications that themselves use ReadLine.

`NonStop`

If set, the debugger goes into non-interactive mode until interrupted, or programmatically by setting $DB::signal or $DB::single.

Here's an example of using the `$ENV{PERLDB_OPTS}` variable:

```
$ PERLDB_OPTS="NonStop frame=2" perl -d myprogram
```

That will run the script **myprogram** without human intervention, printing out the call tree with entry and exit points. Note that `NonStop=1 frame=2` is equivalent to `N f=2`, and that originally, options could be uniquely abbreviated by the first letter (modulo the `Dump*` options). It is nevertheless recommended that you always spell them out in full for legibility and future compatibility.

Other examples include

```
$ PERLDB_OPTS="NonStop LineInfo=listing frame=2" perl -d myprogram
```

which runs script non-interactively, printing info on each entry into a subroutine and each executed line into the file named *listing*. (If you interrupt it, you would better reset `LineInfo` to something "interactive"!)

Other examples include (using standard shell syntax to show environment variable settings):

```
$ ( PERLDB_OPTS="NonStop frame=1 AutoTrace LineInfo=tperl.out"
    perl -d myprogram )
```

which may be useful for debugging a program that uses `Term::ReadLine` itself. Do not forget to detach your shell from the TTY in the window that corresponds to */dev/ttyXX*, say, by issuing a command like

```
$ sleep 1000000
```

See *"Debugger Internals" in perldebguts* for details.

Debugger input/output

Prompt

The debugger prompt is something like

```
DB<8>
```

or even

```
DB<<17>>
```

where that number is the command number, and which you'd use to access with the built-in **csh**-like history mechanism. For example, `!17` would repeat command number 17. The depth of the angle brackets indicates the nesting depth of the debugger. You could get more than one set of brackets, for example, if you'd already at a breakpoint and then printed the result of a function call that itself has a breakpoint, or you step into an expression via `s/n/t expression` command.

Multiline commands

If you want to enter a multi-line command, such as a subroutine definition with several statements or a format, escape the newline that would normally end the debugger command with a backslash. Here's an example:

```
DB<1> for (1..4) {          \
cont:     print "ok\n";    \
cont: }
ok
ok
ok
ok
```

Note that this business of escaping a newline is specific to interactive commands typed into the debugger.

Stack backtrace

Here's an example of what a stack backtrace via `T` command might look like:

```
    $ = main::infested called from file `Ambulation.pm' line 10
    @ = Ambulation::legs(1, 2, 3, 4) called from file
`camel_flea' line 7
    $ = main::pests('bactrian', 4) called from file `camel_flea'
 line 4
```

The left-hand character up there indicates the context in which the function was called, with $ and @ meaning scalar or list contexts respectively, and `.` meaning void context (which is actually a sort of scalar context). The display above says that you were in the function `main::infested` when you ran the stack dump, and that it was called in scalar context from line 10 of the file *Ambulation.pm*, but without any arguments at all, meaning it was called as `&infested`. The next stack frame shows that the function `Ambulation::legs` was called in list context from the *camel_flea* file with four arguments. The last stack frame shows that `main::pests` was called in scalar

context, also from *camel_flea*, but from line 4.

If you execute the `T` command from inside an active `use` statement, the backtrace will contain both a `require` frame and an `eval`) frame.

Line Listing Format

This shows the sorts of output the `l` command can produce:

```
    DB<<13>> l
  101:                @i{@i} = ();
  102:b               @isa{@i,$pack} = ()
  103                     if(exists $i{$prevpack} || exists
$isa{$pack});
  104             }
  105
  106             next
  107==>              if(exists $isa{$pack});
  108
  109:a           if ($extra-- > 0) {
  110:                %isa = ($pack,1);
```

Breakable lines are marked with `:`. Lines with breakpoints are marked by `b` and those with actions by `a`. The line that's about to be executed is marked by `==>`.

Please be aware that code in debugger listings may not look the same as your original source code. Line directives and external source filters can alter the code before Perl sees it, causing code to move from its original positions or take on entirely different forms.

Frame listing

When the `frame` option is set, the debugger would print entered (and optionally exited) subroutines in different styles. See *perldebguts* for incredibly long examples of these.

Debugging compile-time statements

If you have compile-time executable statements (such as code within BEGIN and CHECK blocks or `use` statements), these will *not* be stopped by debugger, although `requires` and INIT blocks will, and compile-time statements can be traced with `AutoTrace` option set in `PERLDB_OPTS`). From your own Perl code, however, you can transfer control back to the debugger using the following statement, which is harmless if the debugger is not running:

```
    $DB::single = 1;
```

If you set `$DB::single` to 2, it's equivalent to having just typed the `n` command, whereas a value of 1 means the `s` command. The `$DB::trace` variable should be set to 1 to simulate having typed the `t` command.

Another way to debug compile-time code is to start the debugger, set a breakpoint on the *load* of some module:

```
    DB<7> b load f:/perllib/lib/Carp.pm
  Will stop on load of `f:/perllib/lib/Carp.pm'.
```

and then restart the debugger using the `R` command (if possible). One can use `b compile subname` for the same purpose.

Debugger Customization

The debugger probably contains enough configuration hooks that you won't ever have to modify it yourself. You may change the behaviour of debugger from within the debugger using its o command, from the command line via the PERLDB_OPTS environment variable, and from customization files.

You can do some customization by setting up a *.perldb* file, which contains initialization code. For instance, you could make aliases like these (the last one is one people expect to be there):

```
$DB::alias{'len'}  = 's/^len(.*)/p length($1)/';
$DB::alias{'stop'} = 's/^stop (at|in)/b/';
$DB::alias{'ps'}   = 's/^ps\b/p scalar /';
$DB::alias{'quit'} = 's/^quit(\s*)/exit/';
```

You can change options from *.perldb* by using calls like this one;

```
parse_options("NonStop=1 LineInfo=db.out AutoTrace=1 frame=2");
```

The code is executed in the package DB. Note that *.perldb* is processed before processing PERLDB_OPTS. If *.perldb* defines the subroutine `afterinit`, that function is called after debugger initialization ends. *.perldb* may be contained in the current directory, or in the home directory. Because this file is sourced in by Perl and may contain arbitrary commands, for security reasons, it must be owned by the superuser or the current user, and writable by no one but its owner.

You can mock TTY input to debugger by adding arbitrary commands to @DB::typeahead. For example, your *.perldb* file might contain:

```
sub afterinit { push @DB::typeahead, "b 4", "b 6"; }
```

Which would attempt to set breakpoints on lines 4 and 6 immediately after debugger initialization. Note that @DB::typeahead is not a supported interface and is subject to change in future releases.

If you want to modify the debugger, copy *perl5db.pl* from the Perl library to another name and hack it to your heart's content. You'll then want to set your PERL5DB environment variable to say something like this:

```
BEGIN { require "myperl5db.pl" }
```

As a last resort, you could also use PERL5DB to customize the debugger by directly setting internal variables or calling debugger functions.

Note that any variables and functions that are not documented in this document (or in *perldebguts*) are considered for internal use only, and as such are subject to change without notice.

Readline Support

As shipped, the only command-line history supplied is a simplistic one that checks for leading exclamation points. However, if you install the Term::ReadKey and Term::ReadLine modules from CPAN, you will have full editing capabilities much like GNU *readline*(3) provides. Look for these in the *modules/by-module/Term* directory on CPAN. These do not support normal **vi** command-line editing, however.

A rudimentary command-line completion is also available. Unfortunately, the names of lexical variables are not available for completion.

Editor Support for Debugging

If you have the FSF's version of **emacs** installed on your system, it can interact with the Perl debugger to provide an integrated software development environment reminiscent of its interactions with C debuggers.

Perl comes with a start file for making **emacs** act like a syntax-directed editor that understands (some of) Perl's syntax. Look in the *emacs* directory of the Perl source distribution.

A similar setup by Tom Christiansen for interacting with any vendor-shipped **vi** and the X11 window system is also available. This works similarly to the integrated multiwindow support that **emacs** provides, where the debugger drives the editor. At the time of this writing, however, that tool's eventual location in the Perl distribution was uncertain.

Users of **vi** should also look into **vim** and **gvim**, the mousey and windy version, for coloring of Perl keywords.

Note that only perl can truly parse Perl, so all such CASE tools fall somewhat short of the mark, especially if you don't program your Perl as a C programmer might.

The Perl Profiler

If you wish to supply an alternative debugger for Perl to run, just invoke your script with a colon and a package argument given to the **-d** flag. The most popular alternative debuggers for Perl is the Perl profiler. Devel::DProf is now included with the standard Perl distribution. To profile your Perl program in the file *mycode.pl*, just type:

```
$ perl -d:DProf mycode.pl
```

When the script terminates the profiler will dump the profile information to a file called *tmon.out*. A tool like **dprofpp**, also supplied with the standard Perl distribution, can be used to interpret the information in that profile.

Debugging regular expressions

`use re 'debug'` enables you to see the gory details of how the Perl regular expression engine works. In order to understand this typically voluminous output, one must not only have some idea about how regular expression matching works in general, but also know how Perl's regular expressions are internally compiled into an automaton. These matters are explored in some detail in *"Debugging regular expressions" in perldebguts*.

Debugging memory usage

Perl contains internal support for reporting its own memory usage, but this is a fairly advanced concept that requires some understanding of how memory allocation works. See *"Debugging Perl memory usage" in perldebguts* for the details.

SEE ALSO

You did try the **-w** switch, didn't you?

perldebtut, *perldebguts*, *re*, *DB*, *Devel::DProf*, *dprofpp*, *Dumpvalue*, and *perlrun*.

When debugging a script that uses #! and is thus normally found in $PATH, the -S option causes perl to search $PATH for it, so you don't have to type the path or `which $scriptname`.

```
$ perl -Sd foo.pl
```

BUGS

You cannot get stack frame information or in any fashion debug functions that were not compiled by Perl, such as those from C or C++ extensions.

If you alter your @_ arguments in a subroutine (such as with `shift` or `pop`), the stack backtrace will not show the original values.

The debugger does not currently work in conjunction with the **-W** command-line switch, because it itself is not free of warnings.

If you're in a slow syscall (like `wait`ing, `accept`ing, or `read`ing from your keyboard or a socket) and haven't set up your own `$SIG{INT}` handler, then you won't be able to CTRL-C your way back to the debugger, because the debugger's own `$SIG{INT}` handler doesn't understand that it needs to raise an exception to longjmp(3) out of slow syscalls.