

NAME

perldebbugs - Guts of Perl debugging

DESCRIPTION

This is not *perldebug*, which tells you how to use the debugger. This manpage describes low-level details concerning the debugger's internals, which range from difficult to impossible to understand for anyone who isn't incredibly intimate with Perl's guts. Caveat lector.

Debugger Internals

Perl has special debugging hooks at compile-time and run-time used to create debugging environments. These hooks are not to be confused with the *perl -Dxxx* command described in *perlrun*, which is usable only if a special Perl is built per the instructions in the *INSTALL* podpage in the Perl source tree.

For example, whenever you call Perl's built-in `caller` function from the package `DB`, the arguments that the corresponding stack frame was called with are copied to the `@DB::args` array. These mechanisms are enabled by calling Perl with the `-d` switch. Specifically, the following additional features are enabled (cf. "*\$^P*" in *perlvar*):

- Perl inserts the contents of `$ENV{PERL5DB}` (or `BEGIN {require 'perl5db.pl'}` if not present) before the first line of your program.
- Each array `@{"_<$filename"}` holds the lines of `$filename` for a file compiled by Perl. The same is also true for `eval`d strings that contain subroutines, or which are currently being executed. The `$filename` for `eval`d strings looks like `(eval 34)`.
Values in this array are magical in numeric context: they compare equal to zero only if the line is not breakable.
- Each hash `%{"_<$filename"}` contains breakpoints and actions keyed by line number. Individual entries (as opposed to the whole hash) are settable. Perl only cares about Boolean true here, although the values used by *perl5db.pl* have the form `"$break_condition\0$action"`.
The same holds for evaluated strings that contain subroutines, or which are currently being executed. The `$filename` for `eval`d strings looks like `(eval 34)`.
- Each scalar `${"_<$filename"}` contains `"_<$filename"`. This is also the case for evaluated strings that contain subroutines, or which are currently being executed. The `$filename` for `eval`d strings looks like `(eval 34)`.
- After each `required` file is compiled, but before it is executed, `DB::postponed(*{"_<$filename"})` is called if the subroutine `DB::postponed` exists. Here, the `$filename` is the expanded name of the `required` file, as found in the values of `%INC`.
- After each subroutine `subname` is compiled, the existence of `$DB::postponed{subname}` is checked. If this key exists, `DB::postponed(subname)` is called if the `DB::postponed` subroutine also exists.
- A hash `%DB::sub` is maintained, whose keys are subroutine names and whose values have the form `filename:startline-endline`. `filename` has the form `(eval 34)` for subroutines defined inside `eval`s.
- When the execution of your program reaches a point that can hold a breakpoint, the `DB::DB()` subroutine is called if any of the variables `$DB::trace`, `$DB::single`, or `$DB::signal` is true. These variables are not localizable. This feature is disabled when executing inside `DB::DB()`, including functions called from it unless `$^D & (1<30)` is true.
- When execution of the program reaches a subroutine call, a call to `&DB::sub(args)` is made

instead, with `$DB::sub` holding the name of the called subroutine. (This doesn't happen if the subroutine was compiled in the `DB` package.)

Note that if `&DB::sub` needs external data for it to work, no subroutine call is possible without it. As an example, the standard debugger's `&DB::sub` depends on the `$DB::deep` variable (it defines how many levels of recursion deep into the debugger you can go before a mandatory break). If `$DB::deep` is not defined, subroutine calls are not possible, even though `&DB::sub` exists.

Writing Your Own Debugger

Environment Variables

The `PERL5DB` environment variable can be used to define a debugger. For example, the minimal "working" debugger (it actually doesn't do anything) consists of one line:

```
sub DB::DB {}
```

It can easily be defined like this:

```
$ PERL5DB="sub DB::DB {}" perl -d your-script
```

Another brief debugger, slightly more useful, can be created with only the line:

```
sub DB::DB {print ++$i; scalar <STDIN>}
```

This debugger prints a number which increments for each statement encountered and waits for you to hit a newline before continuing to the next statement.

The following debugger is actually useful:

```
{
  package DB;
  sub DB {}
  sub sub {print ++$i, " $sub\n"; &$sub}
}
```

It prints the sequence number of each subroutine call and the name of the called subroutine. Note that `&DB::sub` is being compiled into the package `DB` through the use of the `package` directive.

When it starts, the debugger reads your rc file (`./perldb` or `~/.perldb` under Unix), which can set important options. (A subroutine (`&afterinit`) can be defined here as well; it is executed after the debugger completes its own initialization.)

After the rc file is read, the debugger reads the `PERLDB_OPTS` environment variable and uses it to set debugger options. The contents of this variable are treated as if they were the argument of an `o` ... debugger command (q.v. in "Configurable Options" in *perldebug*).

Debugger Internal Variables

In addition to the file and subroutine-related variables mentioned above, the debugger also maintains various magical internal variables.

- `@DB::dbline` is an alias for `@{"::_<current_file">}`, which holds the lines of the currently-selected file (compiled by Perl), either explicitly chosen with the debugger's `f` command, or implicitly by flow of execution.

Values in this array are magical in numeric context: they compare equal to zero only if the line is not breakable.

- `%DB::dbline` is an alias for `%{"::_<current_file">}`, which contains breakpoints and actions keyed by line number in the currently-selected file, either explicitly chosen with the

debugger's `f` command, or implicitly by flow of execution.

As previously noted, individual entries (as opposed to the whole hash) are settable. Perl only cares about Boolean true here, although the values used by *perl5db.pl* have the form `"$break_condition\0$action"`.

Debugger Customization Functions

Some functions are provided to simplify customization.

- See *"Configurable Options" in perldebug* for a description of options parsed by `DB::parse_options(string)`.
- `DB::dump_trace(skip[, count])` skips the specified number of frames and returns a list containing information about the calling frames (all of them, if `count` is missing). Each entry is reference to a hash with keys `context` (either `.`, `$`, or `@`), `sub` (subroutine name, or info about `eval`), `args` (undef or a reference to an array), `file`, and `line`.
- `DB::print_trace(FH, skip[, count[, short]])` prints formatted info about caller frames. The last two functions may be convenient as arguments to `<`, `<<` commands.

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

Frame Listing Output Examples

The `frame` option can be used to control the output of frame information. For example, contrast this expression trace:

```
$ perl -de 42
Stack dump during die enabled outside of evals.

Loading DB routines from perl5db.pl patch level 0.94
Emacs support available.

Enter h or 'h h' for help.

main::(-e:1): 0
DB<1> sub foo { 14 }

DB<2> sub bar { 3 }

DB<3> t print foo() * bar()
main::((eval 172):3): print foo() + bar();
main::foo((eval 168):2):
main::bar((eval 170):2):
42
```

with this one, once the option `frame=2` has been set:

```
DB<4> o f=2
frame = '2'
DB<5> t print foo() * bar()
3: foo() * bar()
entering main::foo
2: sub foo { 14 };
exited main::foo
entering main::bar
2: sub bar { 3 };
```

```
exited main::bar
42
```

By way of demonstration, we present below a laborious listing resulting from setting your `PERLDB_OPTS` environment variable to the value `f=n N`, and running `perl -d -V` from the command line. Examples using various values of `n` are shown to give you a feel for the difference between settings. Long though it may be, this is not a complete listing, but only excerpts.

```
1      entering main::BEGIN
        entering Config::BEGIN
          Package lib/Exporter.pm.
          Package lib/Carp.pm.
          Package lib/Config.pm.
        entering Config::TIEHASH
        entering Exporter::import
          entering Exporter::export
        entering Config::myconfig
        entering Config::FETCH
        entering Config::FETCH
        entering Config::FETCH
        entering Config::FETCH

2      entering main::BEGIN
        entering Config::BEGIN
          Package lib/Exporter.pm.
          Package lib/Carp.pm.
        exited Config::BEGIN
        Package lib/Config.pm.
        entering Config::TIEHASH
        exited Config::TIEHASH
        entering Exporter::import
          entering Exporter::export
          exited Exporter::export
        exited Exporter::import
        exited main::BEGIN
        entering Config::myconfig
        entering Config::FETCH
        exited Config::FETCH
        entering Config::FETCH
        exited Config::FETCH
        entering Config::FETCH

3      in $=main::BEGIN() from /dev/null:0
        in $=Config::BEGIN() from lib/Config.pm:2
          Package lib/Exporter.pm.
          Package lib/Carp.pm.
          Package lib/Config.pm.
        in $=Config::TIEHASH('Config') from lib/Config.pm:644
        in $=Exporter::import('Config', 'myconfig', 'config_vars') from
/dev/null:0
          in $=Exporter::export('Config', 'main', 'myconfig',
'config_vars') from li
        in @=Config::myconfig() from /dev/null:0
          in $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
          in $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
```

```

    in $=Config::FETCH(ref(Config), 'PERL_VERSION') from
lib/Config.pm:574
    in $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from
lib/Config.pm:574
    in $=Config::FETCH(ref(Config), 'osname') from lib/Config.pm:574
    in $=Config::FETCH(ref(Config), 'osvers') from lib/Config.pm:574

4    in $=main::BEGIN() from /dev/null:0
    in $=Config::BEGIN() from lib/Config.pm:2
    Package lib/Exporter.pm.
    Package lib/Carp.pm.
    out $=Config::BEGIN() from lib/Config.pm:0
    Package lib/Config.pm.
    in $=Config::TIEHASH('Config') from lib/Config.pm:644
    out $=Config::TIEHASH('Config') from lib/Config.pm:644
    in $=Exporter::import('Config', 'myconfig', 'config_vars') from
/dev/null:0
    in $=Exporter::export('Config', 'main', 'myconfig',
'config_vars') from lib/
    out $=Exporter::export('Config', 'main', 'myconfig',
'config_vars') from lib/
    out $=Exporter::import('Config', 'myconfig', 'config_vars') from
/dev/null:0
    out $=main::BEGIN() from /dev/null:0
    in @=Config::myconfig() from /dev/null:0
    in $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
    out $=Config::FETCH(ref(Config), 'package') from lib/Config.pm:574
    in $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
    out $=Config::FETCH(ref(Config), 'baserev') from lib/Config.pm:574
    in $=Config::FETCH(ref(Config), 'PERL_VERSION') from
lib/Config.pm:574
    out $=Config::FETCH(ref(Config), 'PERL_VERSION') from
lib/Config.pm:574
    in $=Config::FETCH(ref(Config), 'PERL_SUBVERSION') from
lib/Config.pm:574

5    in $=main::BEGIN() from /dev/null:0
    in $=Config::BEGIN() from lib/Config.pm:2
    Package lib/Exporter.pm.
    Package lib/Carp.pm.
    out $=Config::BEGIN() from lib/Config.pm:0
    Package lib/Config.pm.
    in $=Config::TIEHASH('Config') from lib/Config.pm:644
    out $=Config::TIEHASH('Config') from lib/Config.pm:644
    in $=Exporter::import('Config', 'myconfig', 'config_vars') from
/dev/null:0
    in $=Exporter::export('Config', 'main', 'myconfig',
'config_vars') from lib/E
    out $=Exporter::export('Config', 'main', 'myconfig',
'config_vars') from lib/E
    out $=Exporter::import('Config', 'myconfig', 'config_vars') from
/dev/null:0
    out $=main::BEGIN() from /dev/null:0
    in @=Config::myconfig() from /dev/null:0
    in $=Config::FETCH('Config=HASH(0x1aa444)', 'package') from

```

```

lib/Config.pm:574    out $=Config::FETCH('Config=HASH(0x1aa444)',
'package') from lib/Config.pm:574
    in $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from
lib/Config.pm:574
    out $=Config::FETCH('Config=HASH(0x1aa444)', 'baserev') from
lib/Config.pm:574

6    in $=CODE(0x15eca4)() from /dev/null:0
    in $=CODE(0x182528)() from lib/Config.pm:2
    Package lib/Exporter.pm.
    out $=CODE(0x182528)() from lib/Config.pm:0
    scalar context return from CODE(0x182528): undef
    Package lib/Config.pm.
    in $=Config::TIEHASH('Config') from lib/Config.pm:628
    out $=Config::TIEHASH('Config') from lib/Config.pm:628
    scalar context return from Config::TIEHASH: empty hash
    in $=Exporter::import('Config', 'myconfig', 'config_vars') from
/dev/null:0
    in $=Exporter::export('Config', 'main', 'myconfig',
'config_vars') from lib/Exporter.pm:171
    out $=Exporter::export('Config', 'main', 'myconfig',
'config_vars') from lib/Exporter.pm:171
    scalar context return from Exporter::export: ''
    out $=Exporter::import('Config', 'myconfig', 'config_vars') from
/dev/null:0
    scalar context return from Exporter::import: ''

```

In all cases shown above, the line indentation shows the call tree. If bit 2 of `frame` is set, a line is printed on exit from a subroutine as well. If bit 4 is set, the arguments are printed along with the caller info. If bit 8 is set, the arguments are printed even if they are tied or references. If bit 16 is set, the return value is printed, too.

When a package is compiled, a line like this

```
Package lib/Carp.pm.
```

is printed with proper indentation.

Debugging Regular Expressions

There are two ways to enable debugging output for regular expressions.

If your perl is compiled with `-DDEBUGGING`, you may use the `-Dr` flag on the command line.

Otherwise, one can use `re 'debug'`, which has effects at compile time and run time. Since Perl 5.9.5, this pragma is lexically scoped.

Compile-time Output

The debugging output at compile time looks like this:

```

Compiling REx '[bc]d(ef*g)+h[ij]k$'
size 45 Got 364 bytes for offset annotations.
first at 1
rarest char g at 0
rarest char d at 0
  1: ANYOF[bc](12)
 12: EXACT <d>(14)
 14: CURLYX[0] {1,32767}(28)

```

```

16:  OPEN1(18)
18:    EXACT <e>(20)
20:    STAR(23)
21:      EXACT <f>(0)
23:      EXACT <g>(25)
25:    CLOSE1(27)
27:    WHILEM[1/1](0)
28:  NOTHING(29)
29:  EXACT <h>(31)
31:  ANYOF[ij](42)
42:  EXACT <k>(44)
44:  EOL(45)
45:  END(0)
anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)
  stclass 'ANYOF[bc]' minlen 7
Offsets: [45]
1[4] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 5[1]
0[0] 12[1] 0[0] 6[1] 0[0] 7[1] 0[0] 9[1] 8[1] 0[0] 10[1] 0[0]
11[1] 0[0] 12[0] 12[0] 13[1] 0[0] 14[4] 0[0] 0[0] 0[0] 0[0]
0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 18[1] 0[0] 19[1] 20[0]
Omitting $` $& $' support.

```

The first line shows the pre-compiled form of the regex. The second shows the size of the compiled form (in arbitrary units, usually 4-byte words) and the total number of bytes allocated for the offset/length table, usually $4 + \text{size} * 8$. The next line shows the label *id* of the first node that does a match.

The

```

anchored 'de' at 1 floating 'gh' at 3..2147483647 (checking floating)
  stclass 'ANYOF[bc]' minlen 7

```

line (split into two lines above) contains optimizer information. In the example shown, the optimizer found that the match should contain a substring *de* at offset 1, plus substring *gh* at some offset between 3 and infinity. Moreover, when checking for these substrings (to abandon impossible matches quickly), Perl will check for the substring *gh* before checking for the substring *de*. The optimizer may also use the knowledge that the match starts (at the first *id*) with a character class, and no string shorter than 7 characters can possibly match.

The fields of interest which may appear in this line are

```

anchored STRING at POS
floating STRING at POS1..POS2

```

See above.

```

matching floating/anchored
  Which substring to check first.

```

```

minlen
  The minimal length of the match.

```

```

stclass TYPE
  Type of first matching node.

```

```

noscan
  Don't scan for the found substrings.

```

`isall`

Means that the optimizer information is all that the regular expression contains, and thus one does not need to enter the regex engine at all.

`GPOS`

Set if the pattern contains `\G`.

`plus`

Set if the pattern starts with a repeated char (as in `x+y`).

`implicit`

Set if the pattern starts with `.*`.

`with eval`

Set if the pattern contain eval-groups, such as `(?{ code })` and `(??{ code })`.

`anchored(TYPE)`

If the pattern may match only at a handful of places, with `TYPE` being `BOL`, `MBOL`, or `GPOS`. See the table below.

If a substring is known to match at end-of-line only, it may be followed by `$`, as in floating `'k'$`.

The optimizer-specific information is used to avoid entering (a slow) regex engine on strings that will not definitely match. If the `isall` flag is set, a call to the regex engine may be avoided even when the optimizer found an appropriate place for the match.

Above the optimizer section is the list of *nodes* of the compiled form of the regex. Each line has format

id: *TYPE* *OPTIONAL-INFO* (*next-id*)

Types of Nodes

Here are the possible types, with short descriptions:

TYPE arg-description [num-args] [longjump-len] DESCRIPTION

Exit points

END	no	End of program.
SUCCEED	no	Return from a subroutine, basically.

Anchors:

BOL	no	Match "" at beginning of line.
MBOL	no	Same, assuming multiline.
SBOL	no	Same, assuming singleline.
EOS	no	Match "" at end of string.
EOL	no	Match "" at end of line.
MEOL	no	Same, assuming multiline.
SEOL	no	Same, assuming singleline.
BOUND	no	Match "" at any word boundary using native charset semantics for non-utf8
BOUNDL	no	Match "" at any locale word boundary
BOUNDU	no	Match "" at any word boundary using Unicode semantics
BOUNDA	no	Match "" at any word boundary using ASCII

		semantics
NBOUND	no	Match "" at any word non-boundary using native charset semantics for non-utf8
NBOUNDL	no	Match "" at any locale word non-boundary
NBOUNDU	no	Match "" at any word non-boundary using Unicode semantics
NBOUNDA	no	Match "" at any word non-boundary using ASCII semantics
GPOS	no	Matches where last m//g left off.

[Special] alternatives:

REG_ANY	no	Match any one character (except newline).
SANY	no	Match any one character.
CANY	no	Match any one byte.
ANYOF	sv	Match character in (or not in) this class, single char match only
ANYOF_WARN_SUPER	sv	Match character in (or not in) this class, warn (if enabled) upon matching a char above Unicode max;
ANYOF_SYNTHETIC	sv	Synthetic start class
POSIXD	none	Some [[:class:]] under /d; the FLAGS field gives which one
POSIXL	none	Some [[:class:]] under /l; the FLAGS field gives which one
POSIXU	none	Some [[:class:]] under /u; the FLAGS field gives which one
POSIXA	none	Some [[:class:]] under /a; the FLAGS field gives which one
NPOSIXD	none	complement of POSIXD, [[:^class:]]
NPOSIXL	none	complement of POSIXL, [[:^class:]]
NPOSIXU	none	complement of POSIXU, [[:^class:]]
NPOSIXA	none	complement of POSIXA, [[:^class:]]
CLUMP	no	Match any extended grapheme cluster sequence

Alternation

# BRANCH	The set of branches constituting a single choice are hooked together with their "next" pointers, since precedence prevents anything being concatenated to any individual branch. The "next" pointer of the last BRANCH in a choice points to the thing following the whole choice. This is also where the final "next" pointer of each individual branch points; each branch starts with the operand node of a BRANCH node.	
#		
#		
#		
#		
#		
#		
BRANCH	node	Match this alternative, or the next...

Back pointer

```
# BACK          Normal "next" pointers all implicitly point forward;
#              BACK exists to make loop structures possible.
# not used
BACK           no           Match "", "next" ptr points backward.

# Literals

EXACT          str         Match this string (preceded by length).
EXACTF         str         Match this non-UTF-8 string (not
                           guaranteed to be folded) using /id rules
                           (w/len).
EXACTFL        str         Match this string (not guaranteed to be
                           folded) using /il rules (w/len).
EXACTFU        str         Match this string (folded iff in UTF-8,
                           length in folding doesn't change if not
                           in UTF-8) using /iu rules (w/len).
EXACTFA        str         Match this string (not guaranteed to be
                           folded) using /iaa rules (w/len).
EXACTFU_SS     str         Match this string (folded iff in UTF-8,
                           length in folding may change even if not
                           in UTF-8) using /iu rules (w/len).
EXACTFU_TRICKYFOLD str     Match this folded UTF-8 string using /iu
                           rules

# Do nothing types

NOTHING        no         Match empty string.
# A variant of above which delimits a group, thus stops optimizations
TAIL           no         Match empty string. Can jump here from
                           outside.

# Loops

# STAR, PLUS    '?', and complex '*' and '+', are implemented as
#              circular BRANCH structures using BACK.  Simple cases
#              (one character per match) are implemented with STAR
#              and PLUS for speed and to minimize recursive plunges.
#
STAR           node        Match this (simple) thing 0 or more
                           times.
PLUS           node        Match this (simple) thing 1 or more
                           times.

CURLY          sv 2        Match this simple thing {n,m} times.
CURLYN         no 2        Capture next-after-this simple thing
CURLYM         no 2        Capture this medium-complex thing {n,m}
                           times.
CURLYX         sv 2        Match this complex thing {n,m} times.

# This terminator creates a loop structure for CURLYX
WHILEM         no         Do curly processing and see if rest
                           matches.
```

```
# Buffer related

# OPEN,CLOSE,GROUPP ...are numbered at compile time.
OPEN          num 1      Mark this point in input as start of #n.
CLOSE         num 1      Analogous to OPEN.

REF           num 1      Match some already matched string
REFF          num 1      Match already matched string, folded
                        using native charset semantics for non-
                        utf8
REFFL         num 1      Match already matched string, folded in
                        loc.
REFFU         num 1      Match already matched string, folded
                        using unicode semantics for non-utf8
REFFA         num 1      Match already matched string, folded
                        using unicode semantics for non-utf8, no
                        mixing ASCII, non-ASCII

# Named references. Code in regcomp.c assumes that these all are after
# the numbered references
NREF          no-sv 1     Match some already matched string
NREFF         no-sv 1     Match already matched string, folded
                        using native charset semantics for non-
                        utf8
NREFFL        no-sv 1     Match already matched string, folded in
                        loc.
NREFFU        num 1      Match already matched string, folded
                        using unicode semantics for non-utf8
NREFFA        num 1      Match already matched string, folded
                        using unicode semantics for non-utf8, no
                        mixing ASCII, non-ASCII

IFMATCH       off 1 2     Succeeds if the following matches.
UNLESSM       off 1 2     Fails if the following matches.
SUSPEND       off 1 1     "Independent" sub-RE.
IFTHEN        off 1 1     Switch, should be preceded by switcher.
GROUPP        num 1      Whether the group matched.

# Support for long RE

LONGJMP       off 1 1     Jump far away.
BRANCHJ       off 1 1     BRANCH with long offset.

# The heavy worker

EVAL          evl 1       Execute some Perl code.

# Modifiers

MINMOD        no         Next operator is not greedy.
LOGICAL       no         Next opcode should set the flag only.
```

```
# This is not used yet
RENUM          off 1 1      Group with independently numbered parens.
```

```
# Trie Related
```

```
# Behave the same as A|LIST|OF|WORDS would. The '..C' variants
# have inline charclass data (ascii only), the 'C' store it in the
# structure.
```

```
TRIE           trie 1      Match many EXACT(F[ALU]?)? at once.
                        flags==type
TRIEC          trie       Same as TRIE, but with embedded charclass
charclass      data
```

```
AHOCORASICK    trie 1      Aho Corasick stclass. flags==type
AHOCORASICKC   trie       Same as AHOCORASICK, but with embedded
charclass      charclass data
```

```
# Regex Subroutines
```

```
GOSUB          num/ofs 2L  recurse to paren arg1 at (signed) ofs
                        arg2
GOSTART        no         recurse to start of pattern
```

```
# Special conditionals
```

```
NGROUPP       no-sv 1     Whether the group matched.
INSUBP        num 1       Whether we are in a specific recurse.
DEFINEP       none 1     Never execute directly.
```

```
# Backtracking Verbs
```

```
ENDLIKE       none       Used only for the type field of verbs
OPFAIL        none       Same as (!)
ACCEPT        parno 1     Accepts the current matched string.
```

```
# Verbs With Arguments
```

```
VERB          no-sv 1     Used only for the type field of verbs
PRUNE         no-sv 1     Pattern fails at this startpoint if no-
                        backtracking through this
MARKPOINT     no-sv 1     Push the current location for rollback by
                        cut.
SKIP          no-sv 1     On failure skip forward (to the mark)
                        before retrying
COMMIT        no-sv 1     Pattern fails outright if backtracking
                        through this
CUTGROUP      no-sv 1     On failure go to the next alternation in
                        the group
```

```
# Control what to keep in $&.
```

```
KEEPS         no         $& begins here.
```

```
# New charclass like patterns
```

```
LNBREAK      none       generic newline pattern
```

```
# SPECIAL  REGOPS
```

```
# This is not really a node, but an optimized away piece of a "long"
# node. To simplify debugging output, we mark it as if it were a node
OPTIMIZED      off      Placeholder for dump.
```

```
# Special opcode with the property that no opcode in a compiled program
# will ever be of this type. Thus it can be used as a flag value that
# no other opcode has been seen. END is used similarly, in that an END
# node cant be optimized. So END implies "unoptimizable" and PSEUDO
# mean "not seen anything to optimize yet".
PSEUDO         off      Pseudo opcode for internal use.
```

Following the optimizer information is a dump of the offset/length table, here split across several lines:

```
Offsets: [45]
1[4] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 5[1]
0[0] 12[1] 0[0] 6[1] 0[0] 7[1] 0[0] 9[1] 8[1] 0[0] 10[1] 0[0]
11[1] 0[0] 12[0] 12[0] 13[1] 0[0] 14[4] 0[0] 0[0] 0[0] 0[0]
0[0] 0[0] 0[0] 0[0] 0[0] 0[0] 18[1] 0[0] 19[1] 20[0]
```

The first line here indicates that the offset/length table contains 45 entries. Each entry is a pair of integers, denoted by `offset[length]`. Entries are numbered starting with 1, so entry #1 here is 1[4] and entry #12 is 5[1]. 1[4] indicates that the node labeled 1: (the 1: ANYOF[bc]) begins at character position 1 in the pre-compiled form of the regex, and has a length of 4 characters. 5[1] in position 12 indicates that the node labeled 12: (the 12: EXACT <d>) begins at character position 5 in the pre-compiled form of the regex, and has a length of 1 character. 12[1] in position 14 indicates that the node labeled 14: (the 14: CURLYX[0] {1,32767}) begins at character position 12 in the pre-compiled form of the regex, and has a length of 1 character---that is, it corresponds to the + symbol in the precompiled regex.

0[0] items indicate that there is no corresponding node.

Run-time Output

First of all, when doing a match, one may get no run-time output even if debugging is enabled. This means that the regex engine was never entered and that all of the job was therefore done by the optimizer.

If the regex engine was entered, the output may look like this:

```
Matching '[bc]d(ef*g)+h[ij]k$' against 'abcdefg__gh__'
Setting an EVAL scope, savestack=3
 2 <ab> <cdefg__gh__> | 1: ANYOF
 3 <abc> <defg__gh__> | 11: EXACT <d>
 4 <abcd> <efg__gh__> | 13: CURLYX {1,32767}
 4 <abcd> <efg__gh__> | 26: WHILEM
0 out of 1..32767 cc=ffffff31c
 4 <abcd> <efg__gh__> | 15: OPEN1
 4 <abcd> <efg__gh__> | 17: EXACT <e>
 5 <abcde> <fg__gh__> | 19: STAR
    EXACT <f> can match 1 times out of 32767...
Setting an EVAL scope, savestack=3
 6 <bcdef> <g__gh__> | 22: EXACT <g>
 7 <bcdefg> <__gh__> | 24: CLOSE1
 7 <bcdefg> <__gh__> | 26: WHILEM
    1 out of 1..32767 cc=ffffff31c
```

```
Setting an EVAL scope, savestack=12
7 <bcdefg> <__gh__> | 15:          OPEN1
7 <bcdefg> <__gh__> | 17:          EXACT <e>
  restoring \1 to 4(4)..7
  failed, try continuation...
7 <bcdefg> <__gh__> | 27:          NOTHING
7 <bcdefg> <__gh__> | 28:          EXACT <h>
  failed...
failed...
```

The most significant information in the output is about the particular *node* of the compiled regex that is currently being tested against the target string. The format of these lines is

STRING-OFFSET <PRE-STRING> <POST-STRING> |ID: TYPE

The *TYPE* info is indented with respect to the backtracking level. Other incidental information appears interspersed within.

Debugging Perl Memory Usage

Perl is a profligate wastrel when it comes to memory use. There is a saying that to estimate memory usage of Perl, assume a reasonable algorithm for memory allocation, multiply that estimate by 10, and while you still may miss the mark, at least you won't be quite so astonished. This is not absolutely true, but may provide a good grasp of what happens.

Assume that an integer cannot take less than 20 bytes of memory, a float cannot take less than 24 bytes, a string cannot take less than 32 bytes (all these examples assume 32-bit architectures, the result are quite a bit worse on 64-bit architectures). If a variable is accessed in two of three different ways (which require an integer, a float, or a string), the memory footprint may increase yet another 20 bytes. A sloppy malloc(3) implementation can inflate these numbers dramatically.

On the opposite end of the scale, a declaration like

```
sub foo;
```

may take up to 500 bytes of memory, depending on which release of Perl you're running.

Anecdotal estimates of source-to-compiled code bloat suggest an eightfold increase. This means that the compiled form of reasonable (normally commented, properly indented etc.) code will take about eight times more space in memory than the code took on disk.

The **-DL** command-line switch is obsolete since circa Perl 5.6.0 (it was available only if Perl was built with **-DDEBUGGING**). The switch was used to track Perl's memory allocations and possible memory leaks. These days the use of malloc debugging tools like *Purify* or *valgrind* is suggested instead. See also "*PERL_MEM_LOG*" in *perlhacktips*.

One way to find out how much memory is being used by Perl data structures is to install the Devel::Size module from CPAN: it gives you the minimum number of bytes required to store a particular data structure. Please be mindful of the difference between the `size()` and `total_size()`.

If Perl has been compiled using Perl's malloc you can analyze Perl memory usage by setting `$ENV{PERL_DEBUG_MSTATS}`.

Using `$ENV{PERL_DEBUG_MSTATS}`

If your perl is using Perl's malloc() and was compiled with the necessary switches (this is the default), then it will print memory usage statistics after compiling your code when

`$ENV{PERL_DEBUG_MSTATS} > 1`, and before termination of the program when `$ENV{PERL_DEBUG_MSTATS} >= 1`. The report format is similar to the following example:

```
$ PERL_DEBUG_MSTATS=2 perl -e "require Carp"
```

```

Memory allocation statistics after compilation: (buckets 4(4)..8188(8192)
 14216 free:   130   117   28    7    9    0    2    2    1 0 0
437    61    36    0    5
 60924 used:   125   137   161   55    7    8    6   16    2 0 1
 74   109   304   84   20
Total sbrk(): 77824/21:119. Odd ends: pad+heads+chain+tail: 0+636+0+2048.
Memory allocation statistics after execution:   (buckets 4(4)..8188(8192)
 30888 free:   245    78   85   13    6    2    1    3    2 0 1
315   162    39   42   11
 175816 used:   265   176 1112  111   26  22  11   27    2 1 1
196   178 1066  798   39
Total sbrk(): 215040/47:145. Odd ends: pad+heads+chain+tail:
0+2192+0+6144.

```

It is possible to ask for such a statistic at arbitrary points in your execution using the `mstat()` function out of the standard `Devel::Peek` module.

Here is some explanation of that format:

```
buckets SMALLEST(APPROX)..GREATEST(APPROX)
```

Perl's `malloc()` uses bucketed allocations. Every request is rounded up to the closest bucket size available, and a bucket is taken from the pool of buckets of that size.

The line above describes the limits of buckets currently in use. Each bucket has two sizes: memory footprint and the maximal size of user data that can fit into this bucket. Suppose in the above example that the smallest bucket were size 4. The biggest bucket would have usable size 8188, and the memory footprint would be 8192.

In a Perl built for debugging, some buckets may have negative usable size. This means that these buckets cannot (and will not) be used. For larger buckets, the memory footprint may be one page greater than a power of 2. If so, the corresponding power of two is printed in the `APPROX` field above.

Free/Used

The 1 or 2 rows of numbers following that correspond to the number of buckets of each size between `SMALLEST` and `GREATEST`. In the first row, the sizes (memory footprints) of buckets are powers of two--or possibly one page greater. In the second row, if present, the memory footprints of the buckets are between the memory footprints of two buckets "above".

For example, suppose under the previous example, the memory footprints were

```

      free:    8    16    32    64   128  256 512 1024 2048 4096
8192
      4     12    24    48    80

```

With a non-`DEBUGGING` perl, the buckets starting from 128 have a 4-byte overhead, and thus an 8192-long bucket may take up to 8188-byte allocations.

```
Total sbrk(): SBRKed/SBRKs:CONTINUOUS
```

The first two fields give the total amount of memory perl `sbrk(2)`ed (ess-broken? :-) and number of `sbrk(2)`s used. The third number is what perl thinks about continuity of returned chunks. So long as this number is positive, `malloc()` will assume that it is probable that `sbrk(2)` will provide continuous memory.

Memory allocated by external libraries is not counted.

```
pad: 0
```

The amount of `sbrk(2)`ed memory needed to keep buckets aligned.

```
heads: 2192
```

Although memory overhead of bigger buckets is kept inside the bucket, for smaller buckets, it is kept in separate areas. This field gives the total size of these areas.

`chain: 0`

`malloc()` may want to subdivide a bigger bucket into smaller buckets. If only a part of the deceased bucket is left unsubdivided, the rest is kept as an element of a linked list. This field gives the total size of these chunks.

`tail: 6144`

To minimize the number of `sbrk(2)`s, `malloc()` asks for more memory. This field gives the size of the yet unused part, which is `sbrk(2)`ed, but never touched.

SEE ALSO

perldebug, *perlguts*, *perlrun* *re*, and *Devel::DProf*.