

## NAME

diagnostics, `splain` - produce verbose warning diagnostics

## SYNOPSIS

Using the `diagnostics` pragma:

```
use diagnostics;  
use diagnostics -verbose;
```

```
enable diagnostics;  
disable diagnostics;
```

Using the `splain` standalone filter program:

```
perl program 2>diag.out  
splain [-v] [-p] diag.out
```

Using `diagnostics` to get stack traces from a misbehaving script:

```
perl -Mdiagnostics=-traceonly my_script.pl
```

## DESCRIPTION

### The `diagnostics` Pragma

This module extends the terse diagnostics normally emitted by both the perl compiler and the perl interpreter (from running perl with a `-w` switch or `use warnings`), augmenting them with the more explicative and endearing descriptions found in *perl<sub>diag</sub>*. Like the other pragmata, it affects the compilation phase of your program rather than merely the execution phase.

To use in your program as a pragma, merely invoke

```
use diagnostics;
```

at the start (or near the start) of your program. (Note that this *does* enable perl's `-w` flag.) Your whole compilation will then be subject(ed :- ) to the enhanced diagnostics. These still go out **STDERR**.

Due to the interaction between runtime and compiletime issues, and because it's probably not a very good idea anyway, you may not use `no diagnostics` to turn them off at compiletime. However, you may control their behaviour at runtime using the `disable()` and `enable()` methods to turn them off and on respectively.

The **-verbose** flag first prints out the *perl<sub>diag</sub>* introduction before any other diagnostics. The `$diagnostics::PRETTY` variable can generate nicer escape sequences for paggers.

Warnings dispatched from perl itself (or more accurately, those that match descriptions found in *perl<sub>diag</sub>*) are only displayed once (no duplicate descriptions). User code generated warnings a la `warn()` are unaffected, allowing duplicate user messages to be displayed.

This module also adds a stack trace to the error message when perl dies. This is useful for pinpointing what caused the death. The **-traceonly** (or just **-t**) flag turns off the explanations of warning messages leaving just the stack traces. So if your script is dying, run it again with

```
perl -Mdiagnostics=-traceonly my_bad_script
```

to see the call stack at the time of death. By supplying the **-warntrace** (or just **-w**) flag, any warnings emitted will also come with a stack trace.

## The *splain* Program

While apparently a whole nuther program, *splain* is actually nothing more than a link to the (executable) *diagnostics.pm* module, as well as a link to the *diagnostics.pod* documentation. The **-v** flag is like the `use diagnostics -verbose` directive. The **-p** flag is like the `$diagnostics::PRETTY` variable. Since you're post-processing with *splain*, there's no sense in being able to `enable()` or `disable()` processing.

Output from *splain* is directed to **STDOUT**, unlike the pragma.

## EXAMPLES

The following file is certain to trigger a few errors at both runtime and compiletime:

```
use diagnostics;
print NOWHERE "nothing\n";
print STDERR "\n\tThis message should be unadorned.\n";
warn "\tThis is a user warning";
print "\nDIAGNOSTIC TESTER: Please enter a <CR> here: ";
my $a, $b = scalar <STDIN>;
print "\n";
print $x/$y;
```

If you prefer to run your program first and look at its problem afterwards, do this:

```
perl -w test.pl 2>test.out
./splain < test.out
```

Note that this is not in general possible in shells of more dubious heritage, as the theoretical

```
(perl -w test.pl >/dev/tty) >& test.out
./splain < test.out
```

Because you just moved the existing **stdout** to somewhere else.

If you don't want to modify your source code, but still have on-the-fly warnings, do this:

```
exec 3>&1; perl -w test.pl 2>&1 1>&3 3>&- | splain 1>&2 3>&-
```

Nifty, eh?

If you want to control warnings on the fly, do something like this. Make sure you do the `use` first, or you won't be able to get at the `enable()` or `disable()` methods.

```
use diagnostics; # checks entire compilation phase
print "\ntime for 1st bogus diags: SQUAWKINGS\n";
print BOGUS1 'nada';
print "done with 1st bogus\n";

disable diagnostics; # only turns off runtime warnings
print "\ntime for 2nd bogus: (squelched)\n";
print BOGUS2 'nada';
print "done with 2nd bogus\n";

enable diagnostics; # turns back on runtime warnings
print "\ntime for 3rd bogus: SQUAWKINGS\n";
print BOGUS3 'nada';
print "done with 3rd bogus\n";
```

```
disable diagnostics;
print "\ntime for 4th bogus: (squelched)\n";
print BOGUS4 'nada';
print "done with 4th bogus\n";
```

## INTERNALS

Diagnostic messages derive from the *perldiag.pod* file when available at runtime. Otherwise, they may be embedded in the file itself when the *splain* package is built. See the *Makefile* for details.

If an extant `$SIG{__WARN__}` handler is discovered, it will continue to be honored, but only after the `diagnostics::splainthis()` function (the module's `$SIG{__WARN__}` interceptor) has had its way with your warnings.

There is a `$diagnostics::DEBUG` variable you may set if you're desperately curious what sorts of things are being intercepted.

```
BEGIN { $diagnostics::DEBUG = 1 }
```

## BUGS

Not being able to say "no diagnostics" is annoying, but may not be insurmountable.

The `-pretty` directive is called too late to affect matters. You have to do this instead, and *before* you load the module.

```
BEGIN { $diagnostics::PRETTY = 1 }
```

I could start up faster by delaying compilation until it should be needed, but this gets a "panic: top\_level" when using the pragma form in Perl 5.001e.

While it's true that this documentation is somewhat subserious, if you use a program named *splain*, you should expect a bit of whimsy.

## AUTHOR

Tom Christiansen <[tchrist@mox.perl.com](mailto:tchrist@mox.perl.com)>, 25 June 1995.