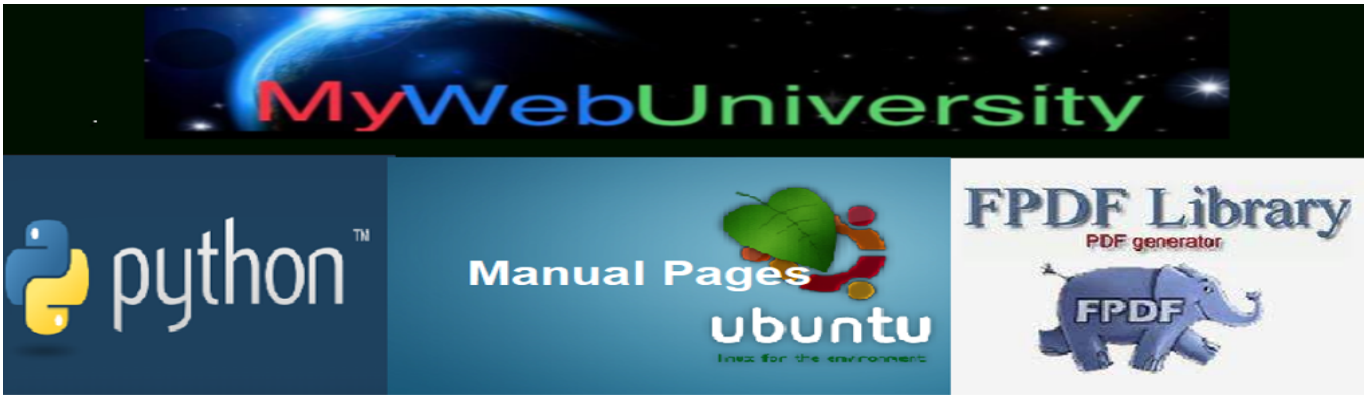

Rocky Enterprise Linux 9.2 Manual Pages on command 'perlpragma.1'

$ man perlpragma.1

PERLPRAGMA(1) Perl Programmers Reference Guide PERLPRAGMA(1)

NAME

perlpragma - how to write a user pragma

DESCRIPTION

A pragma is a module which influences some aspect of the compile time or run time behaviour of Perl, such as "strict" or "warnings". With Perl 5.10 you are no longer limited to the built in pragmata; you can now create user pragmata that modify the behaviour of user functions within a lexical scope.

A basic example

For example, say you need to create a class implementing overloaded mathematical operators, and would like to provide your own pragma that functions much like "use integer;" You'd like this code

```
use MyMaths;
my $l = MyMaths->new(1.2);
my $r = MyMaths->new(3.4);
print "A: ", $l + $r, "\n";
use myint;
print "B: ", $l + $r, "\n";
{
    no myint;
    print "C: ", $l + $r, "\n";
}
print "D: ", $l + $r, "\n";
```

```
    no myint;
    print "E: ", $l + $r, "\n";
```

to give the output

```
    A: 4.6
    B: 4
    C: 4.6
    D: 4
    E: 4.6
```

i.e., where "use myint;" is in effect, addition operations are forced to integer, whereas by default they are not, with the default behaviour being restored via "no myint;"

The minimal implementation of the package "MyMaths" would be something like this:

```
    package MyMaths;
    use warnings;
    use strict;
    use myint();
    use overload '+' => sub {
        my ($l, $r) = @_;
        # Pass 1 to check up one call level from here
        if (myint::in_effect(1)) {
            int($$l) + int($$r);
        } else {
            $$l + $$r;
        }
    };
    sub new {
        my ($class, $value) = @_;
        bless \$value, $class;
    }
    1;
```

Note how we load the user pragma "myint" with an empty list "()" to prevent its "import" being called.

The interaction with the Perl compilation happens inside package "myint":

```
    package myint;
```

```
use strict;
use warnings;
sub import {
    $^H{"myint/in_effect"} = 1;
}
sub unimport {
    $^H{"myint/in_effect"} = 0;
}
sub in_effect {
    my $level = shift // 0;
    my $hinthash = (caller($level))[10];
    return $hinthash->{"myint/in_effect"};
}
1;
```

As pragmata are implemented as modules, like any other module, "use myint;" becomes

```
BEGIN {
    require myint;
    myint->import();
}
```

and "no myint;" is

```
BEGIN {
    require myint;
    myint->unimport();
}
```

Hence the "import" and "unimport" routines are called at compile time for the user's code. User pragmata store their state by writing to the magical hash "%^H", hence these two routines manipulate it. The state information in "%^H" is stored in the optree, and can be retrieved read-only at runtime with "caller()", at index 10 of the list of returned results. In the example pragma, retrieval is encapsulated into the routine "in_effect()", which takes as parameter the number of call frames to go up to find the value of the pragma in the user's script. This uses "caller()" to determine the value of $^H{"myint/in_effect"} when each line of the user's script was called, and therefore provide the correct semantics in the subroutine implementing the overloaded addition.

Key naming

There is only a single "%^H", but arbitrarily many modules that want to use its scoping semantics. To avoid stepping on each other's toes, they need to be sure to use different keys in the hash. It is therefore conventional for a module to use only keys that begin with the module's name (the name of its main package) and a "/" character. After this module-identifying prefix, the rest of the key is entirely up to the module: it may include any characters whatsoever. For example, a module "Foo::Bar" should use keys such as "Foo::Bar/baz" and "Foo::Bar/$%/_!". Modules following this convention all play nicely with each other.

The Perl core uses a handful of keys in "%^H" which do not follow this convention, because they predate it. Keys that follow the convention won't conflict with the core's historical keys.

Implementation details

The optree is shared between threads. This means there is a possibility that the optree will outlive the particular thread (and therefore the interpreter instance) that created it, so true Perl scalars cannot be stored in the optree. Instead a compact form is used, which can only store values that are integers (signed and unsigned), strings or "undef" - references and floating point values are stringified. If you need to store multiple values or complex structures, you should serialise them, for example with "pack". The deletion of a hash key from "%^H" is recorded, and as ever can be distinguished from the existence of a key with value "undef" with "exists".

Don't attempt to store references to data structures as integers which are retrieved via "caller" and converted back, as this will not be threadsafe. Accesses would be to the structure without locking (which is not safe for Perl's scalars), and either the structure has to leak, or it has to be freed when its creating thread terminates, which may be before the optree referencing it is deleted, if other threads outlive it.