SYNOPSIS

```
use warnings;
no warnings;

use warnings "all";
no warnings "all";

use warnings::register;
if (warnings::enabled()) {
    warnings::warn("some warning");
}

if (warnings::enabled("void")) {
    warnings::warn("void", "some warning");
}

if (warnings::enabled($object)) {
    warnings::warn($object, "some warning");
}

warnings::warnif("some warning");
warnings::warnif("void", "some warning");
warnings::warnif($object, "some warning");
```

DESCRIPTION

The `warnings` pragma gives control over which warnings are enabled in which parts of a Perl program. It's a more flexible alternative for both the command line flag **-w** and the equivalent Perl variable, `$^W`.

This pragma works just like the `strict` pragma. This means that the scope of the warning pragma is limited to the enclosing block. It also means that the pragma setting will not leak across files (via `use`, `require` or `do`). This allows authors to independently define the degree of warning checks that will be applied to their module.

By default, optional warnings are disabled, so any legacy code that doesn't attempt to control the warnings will work unchanged.

All warnings are enabled in a block by either of these:

```
use warnings;
use warnings 'all';
```

Similarly all warnings are disabled in a block by either of these:

```
no warnings;
no warnings 'all';
```

For example, consider the code below:

```
use warnings;
my @a;
{
    no warnings;
my $b = @a[0];
}
```

```
    my $c = @a[0];
```

The code in the enclosing block has warnings enabled, but the inner block has them disabled. In this case that means the assignment to the scalar `$c` will trip the `"Scalar value @a[0] better written as $a[0]"` warning, but the assignment to the scalar `$b` will not.

Default Warnings and Optional Warnings

Before the introduction of lexical warnings, Perl had two classes of warnings: mandatory and optional.

As its name suggests, if your code tripped a mandatory warning, you would get a warning whether you wanted it or not. For example, the code below would always produce an `"isn't numeric"` warning about the "2:".

```
    my $a = "2:" + 3;
```

With the introduction of lexical warnings, mandatory warnings now become *default* warnings. The difference is that although the previously mandatory warnings are still enabled by default, they can then be subsequently enabled or disabled with the lexical warning pragma. For example, in the code below, an `"isn't numeric"` warning will only be reported for the `$a` variable.

```
    my $a = "2:" + 3;
    no warnings;
    my $b = "2:" + 3;
```

Note that neither the **-w** flag or the `$^W` can be used to disable/enable default warnings. They are still mandatory in this case.

What's wrong with -w and $^W

Although very useful, the big problem with using **-w** on the command line to enable warnings is that it is all or nothing. Take the typical scenario when you are writing a Perl program. Parts of the code you will write yourself, but it's very likely that you will make use of pre-written Perl modules. If you use the **-w** flag in this case, you end up enabling warnings in pieces of code that you haven't written.

Similarly, using `$^W` to either disable or enable blocks of code is fundamentally flawed. For a start, say you want to disable warnings in a block of code. You might expect this to be enough to do the trick:

```
    {
        local ($^W) = 0;
my $a =+ 2;
my $b; chop $b;
    }
```

When this code is run with the **-w** flag, a warning will be produced for the `$a` line: `"Reversed += operator"`.

The problem is that Perl has both compile-time and run-time warnings. To disable compile-time warnings you need to rewrite the code like this:

```
    {
        BEGIN { $^W = 0 }
my $a =+ 2;
my $b; chop $b;
    }
```

The other big problem with `$^W` is the way you can inadvertently change the warning setting in unexpected places in your code. For example, when the code below is run (without the **-w** flag), the

second call to `doit` will trip a `"Use of uninitialized value"` warning, whereas the first will not.

```
sub doit
{
    my $b; chop $b;
}

doit();

{
    local ($^W) = 1;
    doit()
}
```

This is a side-effect of `$^W` being dynamically scoped.

Lexical warnings get around these limitations by allowing finer control over where warnings can or can't be tripped.

Controlling Warnings from the Command Line

There are three Command Line flags that can be used to control when warnings are (or aren't) produced:

-w

This is the existing flag. If the lexical warnings pragma is **not** used in any of you code, or any of the modules that you use, this flag will enable warnings everywhere. See *Backward Compatibility* for details of how this flag interacts with lexical warnings.

-W

If the **-W** flag is used on the command line, it will enable all warnings throughout the program regardless of whether warnings were disabled locally using `no warnings` or `$^W =0`. This includes all files that get included via `use`, `require` or `do`. Think of it as the Perl equivalent of the "lint" command.

-X

Does the exact opposite to the **-W** flag, i.e. it disables all warnings.

Backward Compatibility

If you are used to working with a version of Perl prior to the introduction of lexically scoped warnings, or have code that uses both lexical warnings and `$^W`, this section will describe how they interact.

How Lexical Warnings interact with **-w**/`$^W`:

1. If none of the three command line flags (**-w**, **-W** or **-X**) that control warnings is used and neither `$^W` nor the `warnings` pragma are used, then default warnings will be enabled and optional warnings disabled. This means that legacy code that doesn't attempt to control the warnings will work unchanged.

2. The **-w** flag just sets the global `$^W` variable as in 5.005. This means that any legacy code that currently relies on manipulating `$^W` to control warning behavior will still work as is.

3. Apart from now being a boolean, the `$^W` variable operates in exactly the same horrible uncontrolled global way, except that it cannot disable/enable default warnings.

4. If a piece of code is under the control of the `warnings` pragma, both the `$^W` variable and the **-w** flag will be ignored for the scope of the lexical warning.

5. The only way to override a lexical warnings setting is with the **-W** or **-X** command line flags.

The combined effect of 3 & 4 is that it will allow code which uses the `warnings` pragma to control the warning behavior of $^W-type code (using a `local $^W=0`) if it really wants to, but not vice-versa.

Category Hierarchy

A hierarchy of "categories" have been defined to allow groups of warnings to be enabled/disabled in isolation.

The current hierarchy is:

```
all -+
     |
     +- closure
     |
     +- deprecated
     |
     +- exiting
     |
     +- experimental --+
     |                 |
     |                 +- experimental::autoderef
     |                 |
     |                 +- experimental::bitwise
     |                 |
     |                 +- experimental::const_attr
     |                 |
     |                 +- experimental::lexical_subs
     |                 |
     |                 +- experimental::lexical_topic
     |                 |
     |                 +- experimental::postderef
     |                 |
     |                 +- experimental::re_strict
     |                 |
     |                 +- experimental::refaliasing
     |                 |
     |                 +- experimental::regex_sets
     |                 |
     |                 +- experimental::signatures
     |                 |
     |                 +- experimental::smartmatch
     |                 |
     |                 +- experimental::win32_perlio
     |
     +- glob
     |
     +- imprecision
     |
     +- io ------------+
     |                 |
     |                 +- closed
     |                 |
     |                 +- exec
     |                 |
     |                 +- layer
```

```
|                 |
|                 +- newline
|                 |
|                 +- pipe
|                 |
|                 +- syscalls
|                 |
|                 +- unopened
|
+- locale
|
+- misc
|
+- missing
|
+- numeric
|
+- once
|
+- overflow
|
+- pack
|
+- portable
|
+- recursion
|
+- redefine
|
+- redundant
|
+- regexp
|
+- severe --------+
|                 |
|                 +- debugging
|                 |
|                 +- inplace
|                 |
|                 +- internal
|                 |
|                 +- malloc
|
+- signal
|
+- substr
|
+- syntax --------+
|                 |
|                 +- ambiguous
|                 |
|                 +- bareword
|                 |
|                 +- digit
|                 |
|                 +- illegalproto
```

```
|                 |
|                 +- parenthesis
|                 |
|                 +- precedence
|                 |
|                 +- printf
|                 |
|                 +- prototype
|                 |
|                 +- qw
|                 |
|                 +- reserved
|                 |
|                 +- semicolon
|
+- taint
|
+- threads
|
+- uninitialized
|
+- unpack
|
+- untie
|
+- utf8 ----------+
|                 |
|                 +- non_unicode
|                 |
|                 +- nonchar
|                 |
|                 +- surrogate
|
+- void
```

Just like the "strict" pragma any of these categories can be combined

```
use warnings qw(void redefine);
no warnings qw(io syntax untie);
```

Also like the "strict" pragma, if there is more than one instance of the `warnings` pragma in a given scope the cumulative effect is additive.

```
use warnings qw(void); # only "void" warnings enabled
...
use warnings qw(io);   # only "void" & "io" warnings enabled
...
no warnings qw(void);  # only "io" warnings enabled
```

To determine which category a specific warning has been assigned to see *perldiag*.

Note: Before Perl 5.8.0, the lexical warnings category "deprecated" was a sub-category of the "syntax" category. It is now a top-level category in its own right.

Note: Before 5.21.0, the "missing" lexical warnings category was internally defined to be the same as the "uninitialized" category. It is now a top-level category in its own right.

Fatal Warnings

The presence of the word "FATAL" in the category list will escalate warnings in those categories into fatal errors in that lexical scope.

NOTE: FATAL warnings should be used with care, particularly `FATAL => 'all'`.

Libraries using *warnings::warn* for custom warning categories generally don't expect *warnings::warn* to be fatal and can wind up in an unexpected state as a result. For XS modules issuing categorized warnings, such unanticipated exceptions could also expose memory leak bugs.

Moreover, the Perl interpreter itself has had serious bugs involving fatalized warnings. For a summary of resolved and unresolved problems as of January 2015, please see *this perl5-porters post*.

While some developers find fatalizing some warnings to be a useful defensive programming technique, using `FATAL => 'all'` to fatalize all possible warning categories -- including custom ones -- is particularly risky. Therefore, the use of `FATAL => 'all'` is *discouraged*.

The *strictures* module on CPAN offers one example of a warnings subset that the module's authors believe is relatively safe to fatalize.

NOTE: users of FATAL warnings, especially those using `FATAL => 'all'`, should be fully aware that they are risking future portability of their programs by doing so. Perl makes absolutely no commitments to not introduce new warnings or warnings categories in the future; indeed, we explicitly reserve the right to do so. Code that may not warn now may warn in a future release of Perl if the Perl5 development team deems it in the best interests of the community to do so. Should code using FATAL warnings break due to the introduction of a new warning we will NOT consider it an incompatible change. Users of FATAL warnings should take special caution during upgrades to check to see if their code triggers any new warnings and should pay particular attention to the fine print of the documentation of the features they use to ensure they do not exploit features that are documented as risky, deprecated, or unspecified, or where the documentation says "so don't do that", or anything with the same sense and spirit. Use of such features in combination with FATAL warnings is ENTIRELY AT THE USER'S RISK.

The following documentation describes how to use FATAL warnings but the perl5 porters strongly recommend that you understand the risks before doing so, especially for library code intended for use by others, as there is no way for downstream users to change the choice of fatal categories.

In the code below, the use of `time`, `length` and `join` can all produce a `"Useless use of xxx in void context"` warning.

```
use warnings;

time;

{
    use warnings FATAL => qw(void);
    length "abc";
}

join "", 1,2,3;

print "done\n";
```

When run it produces this output

```
Useless use of time in void context at fatal line 3.
Useless use of length in void context at fatal line 7.
```

The scope where `length` is used has escalated the `void` warnings category into a fatal error, so the program terminates immediately when it encounters the warning.

To explicitly turn off a "FATAL" warning you just disable the warning it is associated with. So, for example, to disable the "void" warning in the example above, either of these will do the trick:

```
no warnings qw(void);
no warnings FATAL => qw(void);
```

If you want to downgrade a warning that has been escalated into a fatal error back to a normal warning, you can use the "NONFATAL" keyword. For example, the code below will promote all warnings into fatal errors, except for those in the "syntax" category.

```
use warnings FATAL => 'all', NONFATAL => 'syntax';
```

As of Perl 5.20, instead of `use warnings FATAL => 'all';` you can use:

```
use v5.20;       # Perl 5.20 or greater is required for the following
use warnings 'FATAL';  # short form of "use warnings FATAL => 'all';"
```

If you want your program to be compatible with versions of Perl before 5.20, you must use `use warnings FATAL => 'all';` instead. (In previous versions of Perl, the behavior of the statements `use warnings 'FATAL';`, `use warnings 'NONFATAL';` and `no warnings 'FATAL';` was unspecified; they did not behave as if they included the `=> 'all'` portion. As of 5.20, they do.)

Reporting Warnings from a Module

The `warnings` pragma provides a number of functions that are useful for module authors. These are used when you want to report a module-specific warning to a calling module has enabled warnings via the `warnings` pragma.

Consider the module `MyMod::Abc` below.

```
package MyMod::Abc;

use warnings::register;

sub open {
    my $path = shift;
    if ($path !~ m#^/#) {
        warnings::warn("changing relative path to /var/abc")
            if warnings::enabled();
        $path = "/var/abc/$path";
    }
}

1;
```

The call to `warnings::register` will create a new warnings category called "MyMod::Abc", i.e. the new category name matches the current package name. The `open` function in the module will display a warning message if it gets given a relative path as a parameter. This warnings will only be displayed if the code that uses `MyMod::Abc` has actually enabled them with the `warnings` pragma like below.

```
use MyMod::Abc;
use warnings 'MyMod::Abc';
...
abc::open("../fred.txt");
```

It is also possible to test whether the pre-defined warnings categories are set in the calling module with the `warnings::enabled` function. Consider this snippet of code:

```
package MyMod::Abc;

sub open {
    warnings::warnif("deprecated",
                     "open is deprecated, use new instead");
    new(@_);
}

sub new
...
1;
```

The function `open` has been deprecated, so code has been included to display a warning message whenever the calling module has (at least) the "deprecated" warnings category enabled. Something like this, say.

```
use warnings 'deprecated';
use MyMod::Abc;
...
MyMod::Abc::open($filename);
```

Either the `warnings::warn` or `warnings::warnif` function should be used to actually display the warnings message. This is because they can make use of the feature that allows warnings to be escalated into fatal errors. So in this case

```
use MyMod::Abc;
use warnings FATAL => 'MyMod::Abc';
...
MyMod::Abc::open('../fred.txt');
```

the `warnings::warnif` function will detect this and die after displaying the warning message.

The three warnings functions, `warnings::warn`, `warnings::warnif` and `warnings::enabled` can optionally take an object reference in place of a category name. In this case the functions will use the class name of the object as the warnings category.

Consider this example:

```
package Original;

no warnings;
use warnings::register;

sub new
{
    my $class = shift;
    bless [], $class;
}

sub check
{
    my $self = shift;
```

```
        my $value = shift;

        if ($value % 2 && warnings::enabled($self))
          { warnings::warn($self, "Odd numbers are unsafe") }
    }

    sub doit
    {
        my $self = shift;
        my $value = shift;
        $self->check($value);
        # ...
    }

    1;

    package Derived;

    use warnings::register;
    use Original;
    our @ISA = qw( Original );
    sub new
    {
        my $class = shift;
        bless [], $class;
    }

    1;
```

The code below makes use of both modules, but it only enables warnings from `Derived`.

```
    use Original;
    use Derived;
    use warnings 'Derived';
    my $a = Original->new();
    $a->doit(1);
    my $b = Derived->new();
    $a->doit(1);
```

When this code is run only the `Derived` object, `$b`, will generate a warning.

```
    Odd numbers are unsafe at main.pl line 7
```

Notice also that the warning is reported at the line where the object is first used.

When registering new categories of warning, you can supply more names to warnings::register like this:

```
    package MyModule;
    use warnings::register qw(format precision);

    ...

    warnings::warnif('MyModule::format', '...');
```

FUNCTIONS

use warnings::register

Creates a new warnings category with the same name as the package where the call to the pragma is used.

warnings::enabled()

Use the warnings category with the same name as the current package.

Return TRUE if that warnings category is enabled in the calling module. Otherwise returns FALSE.

warnings::enabled($category)

Return TRUE if the warnings category, `$category`, is enabled in the calling module. Otherwise returns FALSE.

warnings::enabled($object)

Use the name of the class for the object reference, `$object`, as the warnings category.

Return TRUE if that warnings category is enabled in the first scope where the object is used. Otherwise returns FALSE.

warnings::fatal_enabled()

Return TRUE if the warnings category with the same name as the current package has been set to FATAL in the calling module. Otherwise returns FALSE.

warnings::fatal_enabled($category)

Return TRUE if the warnings category `$category` has been set to FATAL in the calling module. Otherwise returns FALSE.

warnings::fatal_enabled($object)

Use the name of the class for the object reference, `$object`, as the warnings category.

Return TRUE if that warnings category has been set to FATAL in the first scope where the object is used. Otherwise returns FALSE.

warnings::warn($message)

Print `$message` to STDERR.

Use the warnings category with the same name as the current package.

If that warnings category has been set to "FATAL" in the calling module then die. Otherwise return.

warnings::warn($category, $message)

Print `$message` to STDERR.

If the warnings category, `$category`, has been set to "FATAL" in the calling module then die. Otherwise return.

warnings::warn($object, $message)

Print `$message` to STDERR.

Use the name of the class for the object reference, `$object`, as the warnings category.

If that warnings category has been set to "FATAL" in the scope where `$object` is first used then die. Otherwise return.

warnings::warnif($message)

Equivalent to:

```
if (warnings::enabled())
```

```
    { warnings::warn($message) }
```

warnings::warnif($category, $message)

Equivalent to:

```
    if (warnings::enabled($category))
      { warnings::warn($category, $message) }
```

warnings::warnif($object, $message)

Equivalent to:

```
    if (warnings::enabled($object))
      { warnings::warn($object, $message) }
```

warnings::register_categories(@names)

This registers warning categories for the given names and is primarily for use by the warnings::register pragma.

See also *"Pragmatic Modules" in perlmodlib* and *perldiag*.