91.3 Core Enhancements

91.3.1 Interpreter cloning, threads, and concurrency

Perl 5.6.0 introduces the beginnings of support for running multiple interpreters concurrently in different threads. In conjunction with the perl_clone() API call, which can be used to selectively duplicate the state of any given interpreter, it is possible to compile a piece of code once in an interpreter, clone that interpreter one or more times, and run all the resulting interpreters in distinct threads.

On the Windows platform, this feature is used to emulate fork() at the interpreter level. See *perlfork* for details about that.

This feature is still in evolution. It is eventually meant to be used to selectively clone a subroutine and data reachable from that subroutine in a separate interpreter and run the cloned subroutine in a separate thread. Since there is no shared data between the interpreters, little or no locking will be needed (unless parts of the symbol table are explicitly shared). This is obviously intended to be an easy-to-use replacement for the existing threads support.

Support for cloning interpreters and interpreter concurrency can be enabled using the -Dusethreads Configure option (see win32/Makefile for how to enable it on Windows.) The resulting perl executable will be functionally identical to one that was built with -Dmultiplicity, but the perl_clone() API call will only be available in the former.

-Dusethreads enables the cpp macro USE_ITHREADS by default, which in turn enables Perl source code changes that provide a clear separation between the op tree and the data it operates with. The former is immutable, and can therefore be shared between an interpreter and all of its clones, while the latter is considered local to each interpreter, and is therefore copied for each clone.

Note that building Perl with the -Dusemultiplicity Configure option is adequate if you wish to run multiple **independent** interpreters concurrently in different threads. -Dusethreads only provides the additional functionality of the perl_clone() API call and other support for running **cloned** interpreters concurrently.

```
NOTE: This is an experimental feature.  Implementation details are
subject to change.
```

91.3.2 Lexically scoped warning categories

You can now control the granularity of warnings emitted by perl at a finer level using the `use warnings` pragma. *warnings* and *perllexwarn* have copious documentation on this feature.

91.3.3 Unicode and UTF-8 support

Perl now uses UTF-8 as its internal representation for character strings. The `utf8` and `bytes` pragmas are used to control this support in the current lexical scope. See *perlunicode*, *utf8* and *bytes* for more information.

This feature is expected to evolve quickly to support some form of I/O disciplines that can be used to specify the kind of input and output data (bytes or characters). Until that happens, additional modules from CPAN will be needed to complete the toolkit for dealing with Unicode.

```
NOTE: This should be considered an experimental feature.  Implementation
details are subject to change.
```

91.3.4 Support for interpolating named characters

The new `\N` escape interpolates named characters within strings. For example, `"Hi!  \N{WHITE SMILING FACE}"` evaluates to a string with a Unicode smiley face at the end.

91.3.5 "our" declarations

An "our" declaration introduces a value that can be best understood as a lexically scoped symbolic alias to a global variable in the package that was current where the variable was declared. This is mostly useful as an alternative to the `vars` pragma, but also provides the opportunity to introduce typing and other attributes for such variables. See our in *perlfunc*.