What's Missing

Perl 5 isn't perfect, at least as it behaves by default. Some options are available in the core. More are available from the CPAN. Experienced Perl developers have their own idea of how an ideal Perl 5 should behave, and they often use their own configurations very effectively.

Novices may not know how Perl can help them write programs better. A handful of core modules will make you much more productive.

Missing Defaults

Perl 5's design process in 1993 and 1994 tried to anticipate new directions for the language, but it's impossible to predict the future. Perl 5 added many great new features, but it also kept compatibility with the previous seven years of Perl 1 through Perl 4. Sixteen years later, the best way to write clean, maintainable, powerful, and succinct Perl 5 code is very different from Perl 5.000. The default behaviors sometimes get in the way; fortunately, better behaviors are available.

The strict Pragma

The `strict` pragma (*pragmas*) allows you to forbid (or re-enable) various language constructs which offer power but also the potential for accidental abuse.

`strict` performs three functions: forbidding symbolic references, requiring variable declarations, and forbidding the use of undeclared barewords (*barewords*). While the occasional use of symbolic references is necessary to perform symbol-table manipulation and exporting (barring the use of helper modules, such as `Moose`), the use of a variable as a variable name offers the possibility of subtle errors of action at a distance--or, worse, the possibility of poorly validated user input manipulating internal-only data for malicious purposes.

Requiring variable declarations helps to prevent typos in variable names and encourages proper scoping of lexical variables. It's much easier to see the intended scope of a lexical variable if all variables have `my` or `our` declarations in the appropriate scope.

`strict` has a lexical effect, based on the compile-time scope of its use. You may disable certain features of `strict` (within the smallest possible scope, of course) with `no strict`. See `perldoc strict` for more details.

The warnings Pragma

The `warnings` pragma (*handling_warnings*) controls the reporting of various classes of warnings in Perl 5, such as attempting to stringify the `undef` value or using the wrong type of operator on values. It also warns about the use of deprecated features.

The most useful warnings explain that Perl had trouble understanding what you meant and had to guess at the proper interpretation. Even though Perl often guesses correctly, disambiguation on your part will ensure that your programs run correctly.

The `warnings` pragma has a lexical effect on the compile-time scope of its use. You may disable some or all warnings with `no warnings` (within the smallest possible scope, of course). See `perldoc perllexwarn` and `perldoc warnings` for more details.

Combine `use warnings` with `use diagnostics`, and Perl 5 will display expanded diagnostic messages for each warning present in your programs. These expanded diagnostics come from `perldoc perldiag`. This behavior is useful when learning Perl, but it's less useful in code deployed to production, because it can produce verbose error output.

IO::Handle

Perl 5.6.0 added lexical filehandles. Previously, filehandles were all package globals. This was occasionally messy and often confusing. Now that you can write:

```
open my $fh, '>', $file or die "Can't write to '$file': $!\n";
```

... the lexical filehandle in `$fh` is easier to use. The implementation of lexical filehandles creates objects; `$fh` is an instance of `IO::Handle`. Unfortunately, even though `$fh` is an object, you can't call methods on it because nothing has loaded the `IO::Handle` class.

This is occasionally painful when you want to flush the buffer of the associated filehandle, for example. It could be as easy as:

```
$fh->flush();
```

... but only if your program somewhere contains `use IO::Handle`. The solution is to add this line to your programs so that lexical filehandles--the objects as they are--behave as objects should behave.

The autodie Pragma

Perl 5's default error checking is parsimonious. If you're not careful to check the return value of every `open()` call, for example, you could try to read from a closed filehandle--or worse, lose data as you try to write to one. The `autodie` pragma changes the default behavior. If you write:

```
use autodie;

open my $fh, '>', $file;
```

... an unsuccessful `open()` call will throw an exception via Perl 5's normal exception mechanism. Given that the most appropriate approach to a failed system call is throwing an exception, this pragma can remove a lot of boilerplate code and allow you the peace of mind of knowing that you haven't forgotten to check a return value.

This pragma entered the Perl 5 core as of Perl 5.10.1. See `perldoc autodie` for more information.