Perl isn't perfect, but it is malleable--because no single configuration is ideal for every programmer and every purpose. Some useful behaviors are available as core libraries. More are available from the CPAN. Effective Perl programmers take full advantage of the options available to them.

Useful Core Modules

Perl's language design process has always tried to combine practicality with expandability, but it was as impossible to predict the future in 1994 as it is in 2015. Perl 5 expanded the language and made the CPAN possible, but it also retained backwards compatibility with most Perl 1 code written as far back as 1987.

The best Perl code of 2015 is very different from the best Perl code of 1994 or the best Perl code of 1987, and part of that is due to its core library.

The strict Pragma

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

`strict` forbids symbolic references, requires variable declarations (*lexical_scope*), and prohibits the use of undeclared barewords (*barewords*). While symbolic references are occasionally necessary (*import*), 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 private data for malicious purposes.

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

`strict` takes effect in lexical scopes. See `perldoc strict` for more details.

The warnings Pragma

The `warnings` pragma (*handling_warnings*) controls the reporting of various warning classes, 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 takes effect in lexical scopes. See `perldoc perllexwarn` and `perldoc warnings` for more details.

If you use both `warnings` with `diagnostics`, you'll get expanded diagnostic messages for each warning present in your programs, straight out of `perldoc perldiag`. It's a great help when learning Perl, but be sure to disable `diagnostics` before deploying your program, lest you fill up your logs or expose debugging information to users.

The autodie Pragma

Perl leaves error handling (or error ignoring) up to you. If you forget 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 this for you. If you write:

```
use autodie;

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

... an unsuccessful `open()` call will throw an exception. 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.

One caveat of `autodie` is that it can be a sledgehammer when you need a finishing hammer; if you only need a couple of system calls checked for you, you can limit its imports accordingly. See `perldoc autodie` for more information.

Perl Version Numbers

If you encounter a piece of Perl code without knowing when it was written or who wrote it, can you tell which version of Perl it requires? If you have a lot of experience with Perl both before and after the release of Perl 5.10, you might remember which version added `say` and when `autodie` entered the core. Otherwise, you might have to guess, trawl through `perldelta` files, or use `CPAN::MinimumVersion` from the CPAN.

There's no requirement for you to add the minimum required Perl version number to all new code you write, but it *can* clarify your intentions. For example, if you've tested your code with Perl 5.18 and use only features present in Perl 5.18, write:

```
use 5.018;
```

... and you'll document your intent. You'll also make it easier for tools to identify the particular features of Perl you may or may not use in this code. If someone comes along later and proves that the code works just fine on Perl 5.14, you can change the version number--and you'll do so based on practical evidence.

What's Next?

Although Perl includes an extensive core library, it's not comprehensive. Many of the best modules are available outside of the core, from the CPAN (*cpan*). The `Task::Kensho` meta-distribution includes several other distributions which represent the best the CPAN has to offer. When you need to solve a problem, look there first.

The CPAN has plenty of other gems, though. For example, if you want to:

* *Access a database via SQL*, use the `DBI` module
* *Embed a lightweight, single-file database*, use the `DBD::SQLite` module
* *Manage your database schemas*, use `Sqitch`
* *Represent database entities as objects*, use `DBIx::Class`
* *Perform basic web programming*, use `Plack`
* *Use a powerful web framework*, use `Mojolicious`, `Dancer`, or `Catalyst`
* *Process structured data files*, use `Text::CSV_XS` (or `Text::CSV`)
* *Manage module installations for applications*, use `Carton`
* *Manipulate numeric data*, use `PDL`
* *Manipulate images*, use `Imager`
* *Access shared libraries*, use `FFI::Platypus`
* *Extract data from XML files*, use `XML::Rabbit`
* *Keep your code tidy*, use `Perl::Tidy`
* *Watch for problems beyond strictures and warnings*, use `Perl::Critic`

... and the list goes on. Skim the CPAN recent uploads pagehttp://search.cpan.org/recent frequently to see what's new and what's updated.

Thinking in Perl

As is true of any creative endeavor, learning Perl never stops. While "Modern Perl" describes how the best Perl programmers approach their craft, their techniques and tools always evolve. What's great in 2015 and 2016 might not have been imagined even five years ago--and the greatness of 2020 and beyond might be mere inklings in the mind of an enterprising Perl hacker right now.

Now you have the chance to shape that future. It's up to you to continue discovering how to make Perl work for you and how to make Perl better, whether learning from the global Perl community, perusing the documentation of the core and CPAN modules, and by careful practice, discovering what works for you and what helps you write the right code.

Perl's not perfect (though it improves, year after year, release after release). It can be as clean or as messy as you need it to be, depending on the problems you have to solve. It's up to you to use it well.

As a wise person once said, "May you do good things with Perl."