"Simple" means different things to different programmers. Effective programmers understand how Perl's features interact and combine. Their fluent code takes advantage of language patterns and idioms. The result of this Perlish thinking is concise, powerful, and useful code--and it's simple when you understand it.

Idioms

Every language has common patterns of expression, or *idioms*. The earth revolves, but we speak of the sun rising or setting. We brag about clever hacks but cringe at nasty hacks and code smells.

Perl has idioms; they're both language features and design techniques. They're mannerisms and mechanisms that give your code a Perlish accent. You don't have to use them to get your job done, but they play to Perl's strengths.

The Object as $self

Perl's object system (*moose*) treats the invocant of a method as a mundane parameter. Regardless of whether you invoke a class or an instance method, the first element of `@_` is always the invocant. By convention, most Perl code uses `$class` as the name of the class method invocant and `$self` for the name of the object invocant. This convention is strong enough that useful extensions such as `Moops` assume you will use `$self` as the name of object invocants.

Named Parameters

Perl loves lists. Lists are a fundamental element of Perl. List flattening and binding lets you chain together multiple expressions to manipulate data in every way possible.

While Perl's argument passing simplicity (everything flattens into `@_`) is occasionally too simple, assigning from `@_` in list context allows you to unpack named parameters as pairs. The fat comma (*declaring_hashes*) operator turns an ordinary list into an obvious list of pairs of arguments:

```
make_ice_cream_sundae(
    whipped_cream => 1,
    sprinkles     => 1,
    banana        => 0,
    ice_cream     => 'mint chocolate chip',
);
```

You can unpack these parameters into a hash and treat that hash as if it were a single argument:

```
sub make_ice_cream_sundae {
    B<my %args    = @_;>
    my $dessert = get_ice_cream( $args{ice_cream} );
    ...
}
```

Perl Best Practices suggests passing hash references instead. This allows Perl to perform caller-side validation of the hash reference. If you pass the wrong number of arguments, you'll get an error where you *call* the function.

This technique works well with `import()` (*importing*) or other methods; process as many parameters as you like before slurping the remainder into a hash:

```
sub import {
    B<my ($class, %args)  = @_;>
    my $calling_package = caller();
    ...
}
```

The *Schwartzian transform* is an elegant demonstration of the pervasive list handling idiom borrowed from Lisp. Suppose you have a Perl hash which associates the names of your co-workers with their phone extensions:

```
    my %extensions = (
        '000' => 'Damian',
        '002' => 'Wesley',
        '042' => 'Robin',
        '088' => 'Nic',
    );
```

Fat comma hash key quoting only works on things that look like barewords. With the leading zero, these keys look like octal numbers. Everyone makes this mistake at least once.

To sort this list by name alphabetically, you must sort the hash by its values, not its keys. Getting the values sorted correctly is easy:

```
    my @sorted_names = sort values %extensions;
```

... but you need an extra step to preserve the association of names and extensions, hence the Schwartzian transform. First, convert the hash into a list of data structures which will be easier to sort--in this case, two-element anonymous arrays:

```
    my @pairs = map  { [ B<$_, $extensions{$_}> ] } keys %extensions;
```

`sort` takes this list of anonymous arrays and compares their second elements (the names) as strings:

```
    my @sorted_pairs = sort { $a->[1] cmp $b->[1] } @pairs;
```

The block provided to `sort` receives arguments in two package-scoped (*scope*) variables: $a and $b. (See `perldoc -f sort` for an extensive discussion of the implications of this scoping.) The `sort` block takes its arguments two at a time. The first becomes the contents of $a and the second the contents of $b. If $a should sort ahead of $b in the results, the block must return -1. If both values sort to the same position, the block must return 0. Finally, if $a should sort after $b in the results, the block should return 1. Any other return values are errors.

Reversing the hash *in place* would work if no one had the same name. This particular data set presents no such problem, but code defensively.

The `cmp` operator performs string comparisons and the <=> performs numeric comparisons. Given `@sorted_pairs`, a second `map` operation converts the data structure to a more usable form:

```
    my @formatted_exts = map { "$_->[1], ext. $_->[0]" } @sorted_pairs;
```

... and now you can print the whole thing:

```
    say for @formatted_exts;
```

The Schwartzian transformation chains all of these expressions together to elide those temporary variables:

```
say for
    map  { " $_->[1], ext. $_->[0]"          }
    sort {   $a->[1] cmp   $b->[1]           }
    map  { [ $_      =>    $extensions{$_} ] }
    keys %extensions;
```

Read the expression from right to left, in evaluation order. For each key in the extensions hash, make a two-item anonymous array containing the key and the value. Sort that list of anonymous arrays by their second elements, the hash values. Format a string of output from those sorted arrays.

The Schwartzian transform pipeline of `map`-`sort`-`map` transforms a data structure into another form easier for sorting and then transforms it back into the first from--or another form.

While this sorting example is simple, consider the case of calculating a cryptographic hash for a large file. The Schwartzian transform is especially useful because it effectively caches any expensive calculations by performing them once in the first-executed `map`.

Easy File Slurping

`local` is essential to managing Perl's magic global variables. You must understand scope (*scope*) to use `local` effectively--but if you do, you can use tight and lightweight scopes in interesting ways. For example, to slurp files into a scalar in a single expression:

```
my $file = do { local $/; <$fh> };

# or
my $file; { local $/; $file = <$fh> };
```

`$/` is the input record separator. `local`izing it sets its value to `undef`, pending assignment. As the value of the separator is undefined, Perl happily reads the entire contents of the filehandle in one swoop. Because a do block evaluates to the value of the last expression evaluated within the block, this evaluates to the data read from the filehandle: the contents of the file. At the end of the expression, `$/` has reverted to its previous state and `$file` contains the contents of the file.

The second example avoids a second copy of the string containing the file's contents; it's not as pretty, but it uses the least amount of memory.

This useful example is admittedly maddening for people who don't understand both `local` and scoping. The `File::Slurper` module from the CPAN is a worthy (and often faster) alternative.

Handling Main

Many programs commonly set up several file-scoped lexical variables before handing off processing to other functions. It's tempting to use these variables directly, rather than passing values to and returning values from functions, especially as programs grow. Unfortunately, these programs may come to rely on subtleties of what happens when during Perl's compilation process--a variable you *thought* would be initialized to a specific value may not get initialized until much later. Remember that Perl requires no special syntax for creating closures (*closures*)--you can close over a lexical variable inadvertently.

To avoid this, wrap the main code of your program in a single function, `main()`. Encapsulate your variables to their proper scopes. Then add a single line to the beginning of your program, after you've used all of the modules and pragmas you need:

```
#!/usr/bin/perl
```

```
    use Modern::Perl;

    B<exit main( @ARGV );>

    sub main {
        ...

        # successful exit
        return 0;
    }

    sub other_functions { ... }
```

Calling `main()` *before* anything else in the program forces you to be explicit about initialization and compilation order. Calling `exit` with `main()`'s return value prevents any other bare code from running.

Controlled Execution

The effective difference between a program and a module is in its intended use. Users invoke programs directly, while programs load modules after execution has already begun. Yet both modules and programs are merely Perl code. Making a module executable is easy. So is making a program behave as a module (useful for testing parts of an existing program without formally making a module). All you need to do is to discover *how* Perl began to execute a piece of code.

`caller`'s single optional argument governs the number of call frames (*recursion*) to look back through. `caller(0)` reports information about the current call frame. To allow a module to run correctly as a program *or* a module, put all executable code in functions, add a `main()` function, and write a single line at the start of the module:

```
    main() unless caller(0);
```

If there's *no* caller for the module, someone invoked it directly as a program (with `perl path/to/Module.pm` instead of `use Module;`).

The eighth element of the list returned from `caller` in list context is a true value if the call frame represents `use` or `require` and `undef` otherwise. While that's more accurate, few people use it.

Postfix Parameter Validation

The CPAN has several modules which help verify the parameters of your functions; `Params::Validate` and `MooseX::Params::Validate` are two good options. Simple validation is easy even without those modules.

Suppose your function takes exactly two arguments. You *could* write:

```
    use Carp 'croak';

    sub groom_monkeys {
        if (@_ != 2) {
            croak 'Can only groom two monkeys!';
        }
        ...
```

```
}
```

... but from a linguistic perspective, the consequences are more important than the check and deserve to be at the *start* of the expression:

```
croak 'Can only groom two monkeys!' unless @_ == 2;
```

This early return technique--especially with postfix conditionals--can simplify the rest of the code. Each such assertion is effectively a single row in a truth table. Alternately, function signatures (*function_signatures*) of some kind will handle this case for you.

Regex En Passant

Many Perl idioms rely on the fact that expressions evaluate to values:

```
say my $ext_num = my $extension = 42;
```

That clunky code demonstrates how to use the value of one expression in another expression. This isn't a new idea; you've likely used the return value of a function in a list or as an argument to another function before. You may not have realized its implications.

Suppose you want to extract a first name from a first name plus surname combination with a precompiled regular expression in `$first_name_rx`:

```
my ($first_name) = $name =~ /($first_name_rx)/;
```

In list context, a successful regex match returns a list of all captures (*regex_captures*, and Perl assigns the first one to `$first_name`.

To remove all non-word characters to create a useful user name for a system account, you could write:

```
(my $normalized_name = $name) =~ tr/A-Za-z//dc;
```

Newer code can use the non-destructive substitution modifier `/r`: `my $normalized_name = $name =~ tr/A-Za-z//dc`**`r`**`;`.

First, assign the value of `$name` to `$normalized_name`. The parentheses change precedence so that assignment happens first. The assignment expression evaluates to the *variable* `$normalized_name`. That variable becomes the first operand to the transliteration operator.

This technique works on other in-place modification operators:

```
my $age = 14;
(my $next_age = $age)++;

say "I am $age, but next year I will be $next_age";
```

Unary Coercions

Perl's type system almost always does the right thing when you choose the correct operators. Use the string concatenation operator and Perl will treat both operands as strings. Use the addition operator and Perl will treat both operands as numeric.

Occasionally you have to give Perl a hint about what you mean with a *unary coercion* to force a specific evaluation of a value.

Add zero to treat a value as numeric:

```
    my $numeric_value = 0 + $value;
```

Double negate a value to treat it as a boolean:

```
    my $boolean_value = !! $value;
```

Concatenate a value with the empty string to treat it as a string:

```
    my $string_value = '' . $value;
```

The need for these coercions is vanishingly rare, but it happens. While it may look like it would be safe to remove a "useless" `+ 0` from an expression, doing so may well break the code.

Global Variables

Perl provides several *super global variables*. They're not scoped to a package or file. They're really, truly global. Unfortunately, any direct or indirect modifications of these variables may change the behavior of other parts of the program. Experienced Perl hackers have memorized some of them. Few people have memorized all of them--they're terse. Only a handful are regularly useful. `perldoc perlvar` contains the exhaustive list of these variables.

Managing Super Globals

As Perl evolves, it moves more global behavior into lexical behavior, so that you can avoid many of these globals. When you must use them, use `local` in the smallest possible scope to constrain any modifications. You are still susceptible to any changes made to these variables from code you *call*, but you reduce the likelihood of surprising code *outside* of your scope. As the easy file slurping idiom (*easy_file_slurping*) demonstrates, `local` is often the right approach:

```
    my $file; { B<local $/>; $file = <$fh> };
```

The effect of `local`izing `$/` lasts only through the end of the block. There is a low chance that any Perl code will run as a result of reading lines from the filehandle and change the value of `$/` within the `do` block.

Not all cases of using super globals are this easy to guard, but this often works.

Other times you need to *read* the value of a super global and hope that no other code has modified it. Catching exceptions with an `eval` block is susceptible to at least one race condition where `DESTROY()` methods invoked on lexicals that have gone out of scope may reset `$@`:

```
    local $@;
    eval { ... };

    if (B<my $exception = $@>) { ... }
```

Copy `$@` *immediately* after catching an exception to preserve its contents. See also `Try::Tiny` instead (*exception_caveats*).

English Names

The core `English` module provides verbose names for punctuation-heavy super globals. Import them into a namespace with:

```
    use English '-no_match_vars'; # unnecessary in 5.20 and 5.22
```

This allows you to use the verbose names documented in `perldoc perlvar` within the scope of this pragma.

Three regex-related super globals ($&, `` $` ``, and `$'`) used to impose a global performance penalty for *all* regular expressions within a program. This has been fixed in Perl 5.20. If you forget the `-no_match_vars` import, your program will suffer the penalty even if you don't explicitly read from those variables. Modern Perl programs can use the `@-` variable instead of them.

Useful Super Globals

Most programs can get by with using only a couple of the super globals. You're most likely to encounter only a few of these variables.

`$/` (or `$INPUT_RECORD_SEPARATOR` from the `English` pragma) is a string of zero or more characters which denotes the end of a record when reading input a record at a time. By default, this is your platform-specific newline character sequence. If you undefine this value, Perl will attempt to read the entire file into memory. If you set this value to a *reference* to an integer, Perl will try to read that many *bytes* per record (so beware of Unicode concerns). If you set this value to an empty string (`''`), Perl will read in a paragraph at a time, where a paragraph is a chunk of text followed by an arbitrary number of newlines.

`$.` (`$INPUT_LINE_NUMBER`) contains the number of records read from the most recently-accessed filehandle. You can read from this variable, but writing to it has no effect. Localizing this variable will localize the filehandle to which it refers. Yes, that's confusing.

`$|` (`$OUTPUT_AUTOFLUSH`) governs whether Perl will flush everything written to the currently selected filehandle immediately or only when Perl's buffer is full. Unbuffered output is useful when writing to a pipe or socket or terminal which should not block waiting for input. This variable will coerce any values assigned to it to boolean values.

`@ARGV` contains the command-line arguments passed to the program.

`$!` (`$ERRNO`) is a dualvar (*dualvars*) which contains the result of the *most recent* system call. In numeric context, this corresponds to C's `errno` value, where anything other than zero indicates an error. In string context, this evaluates to the appropriate system error string. Localize this variable before making a system call (implicitly or explicitly) to avoid overwriting the `errno` value for other code elsewhere. Perl's internals make system calls sometimes, so the value of this variable can change out from under you. Copy it *immediately* after causing a system call for accurate results.

`$"` (`$LIST_SEPARATOR`) contains the string used to separate array and list elements interpolated into a string.

`%+` contains named captures from successful regular expression matches (*named_captures*).

`$@` (`$EVAL_ERROR`) contains the value thrown from the most recent exception (*catching_exceptions*).

`$0` (`$PROGRAM_NAME`) contains the name of the program currently executing. You may modify this value on some Unix-like platforms to change the name of the program as it appears to other programs on the system, such as `ps` or `top`.

`$$` (`$PID`) contains the process id of the currently running instance of the program as the operating system understands it. This will vary between `fork()`ed programs and *may* vary between threads in the same program.

`@INC` holds a list of filesystem paths in which Perl will look for files to load with `use` or `require`. See `perldoc -f require` for other items this array can contain.

`%SIG` maps OS and low-level Perl signals to function references used to handle those signals. Trap the standard Ctrl-C interrupt by catching the `INT` signal, for example. See `perldoc perlipc` for more information about signals and signal handling.

Alternatives to Super Globals

IO and exceptions are the worst perpetrators of action at a distance. Use `Try::Tiny` (*exception_caveats*) to insulate you from the tricky semantics of proper exception handling. `localize` and copy the value of `$!` to avoid strange behaviors when Perl makes implicit system calls. Use `IO::File` and its methods on lexical filehandles (*file_handling_variables*) to limit unwanted global changes to IO behavior.