Baby Perl will only get you so far. Language fluency allows you to use the natural patterns and idioms of the language. Effective programmers understand how Perl's features interact and combine.

Prepare for the second learning curve of Perl: Perlish thinking. The result is concise, powerful, and Perlish code.

Idioms

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

Perl idioms aren't quite language features or design techniques. They're mannerisms and mechanisms that, taken together, give your code a Perlish accent. You don't have to use them, but they play to Perl's strengths.

The Object as $self

Perl 5'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 a method's invocant. By convention, most Perl 5 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 `MooseX::Method::Signatures` assume you will use `$self` as the name of object invocants.

Named Parameters

List processing is a fundamental component of Perl's expression evaluation. The ability for Perl programmers to chain expressions which evaluate to variable-length lists provides countless opportunities to manipulate data effectively.

While Perl 5'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',
);
```

The callee side can unpack these parameters into a hash and treat the 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.

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

The *Schwartzian transform* is an elegant demonstration of Perl's pervasive list handling as an idiom handily borrowed from the Lisp family of languages.

Suppose you have a Perl hash which associates the names of your co-workers with their phone extensions:

```
my %extensions =
(
    001 => 'Armon',
    002 => 'Wesley',
    003 => 'Gerald',
    005 => 'Rudy',
    007 => 'Brandon',
    008 => 'Patrick',
    011 => 'Luke',
    012 => 'LaMarcus',
    017 => 'Chris',
    020 => 'Maurice',
    023 => 'Marcus',
    024 => 'Andre',
    052 => 'Greg',
    088 => 'Nic',
);
```

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 is easy to sort--in this case, two-element anonymous arrays:

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

`sort` takes the 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` takes its 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 come before $b in the results, the block must return -1. If both values are sufficiently equal in the sorting terms, the block must return 0. Finally, if $a should come 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 itself uses Perl's pervasive list processing to get rid of the temporary variables. The combination is:

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

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

Easy File Slurping

local is essential to managing Perl 5'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 = do { local $/; <$fh> };

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

$/ is the input record separator. localizing it sets its value to undef, pending assignment. That localization takes place *before* the assignment. As the value of the separator is undefined, Perl happily reads the entire contents of the filehandle in one swoop and assigns that value to `$/`. Because a `do` block evaluates to the value of the last expression evaluated within the block, this evaluates to the value of the assignment: the contents of the file. Even though `$/` immediately reverts to its previous state at the end of the block, `$file` now contains the contents of the file.

The second example contains no assignment and merely returns the single line read from the filehandle.

The third 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::Slurp` module from the CPAN is a worthy (and often faster) alternative.

Handling Main

Perl requires no special syntax for creating closures (*closures*); you can close over a lexical variable inadvertently. 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 5's compilation process; a variable you *thought* would be initialized to a specific value may not get initialized until much later.

To avoid this, wrap the main code of your program in a simple 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( @ARGS );>
```

Calling `main()` *before* anything else in the program forces you to be explicit about initialization and order of compilation. Calling `exit` with `main()`'s return value prevents any other bare code from running, though be sure to return `0` from `main()` on successful execution.

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 a module is Perl code, in the same way that a program is. 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 is the number of call frames (*recursion*) which to report. `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 two arguments, no more and no less. You *could* write:

```
use Carp 'croak';

sub groom_monkeys
{
    if (@_ != 2)
    {
        croak 'Grooming requires 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 'Grooming requires two monkeys!' if @_ != 2;
```

... which may read more simply as:

```
croak 'Grooming requires 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.

Regex En Passant

Many Perl 5 idioms rely on the fact that expressions evaluate to values:

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

While that code is obviously clunky, it 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 modify the name, perhaps removing 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;
```

Perl 5.14 added the non-destructive substitution modifier `/r`, so that you can write `my $normalized_name = $name =~ tr/A-Za-z//dcr;`.

First, assign the value of `$name` to `$normalized_name`, as the parentheses affect the precedence so that assignment happens first. The assignment expression evaluates to the *variable* `$normalized_name`, so that 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 "Next year I will be $next_age";
```

Unary Coercions

Perl 5'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 the evaluation of a value a specific way.

To ensure that Perl treats a value as numeric, add zero:

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

To ensure that Perl treats a value as boolean, double negate it:

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

To ensure that Perl treats a value as a string, concatenate it with the empty string:

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

Though the need for these coercions is vanishingly rare, you should understand these idioms if you encounter them. 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 5 provides several *super global variables* that are truly global, not scoped to a package or file. Unfortunately, their global availability means that any direct or indirect modifications may have effects on other parts of the program--and they're terse. Experienced Perl 5 programmers have memorized some of them. Few people have memorized all of them. Only a handful are ever useful. `perldoc perlvar` contains the exhaustive list of such variables.

Managing Super Globals

Perl 5 continues to move more global behavior into lexical behavior, so you can avoid many of these globals. When you can't avoid them, use `local` in the smallest possible scope to constrain any modifications. You are still susceptible to any changes code you *call* makes to those globals, 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 `$/` lasts only through the end of the block. There is a low chance that any Perl code will run as a result of reading lines from the filehandleA tied filehandle (*tie*) is one of the few possibilities. and change the value of `$/` 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 can be susceptible to race conditions, in that `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';
```

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

Three regex-related super globals ($&, `` $` ``, and `$'`) impose a global performance penalty for *all* regular expressions within a program. 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 should use the `@-` variable as a replacement for the terrible three.

I don't understand this strategy. What is being replaced? In Modern Perl, we have the /p modifier with gives us the not-so-terrible three, ${^MATCH}, ${^PREMATCH}, and ${^POSTMATCH}.

Useful Super Globals

Most modern Perl 5 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 in real programs.

* `$/` (or `$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 line 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.

* `$|` (`$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 appropriate value for other code elsewhere. Many places within Perl 5 itself make system calls without your knowledge, so the value of this variable can change out from under you. Copy it *immediately* after causing a system call for the most accurate results.

* `$"` (`$LIST_SEPARATOR`) is a 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 especially safe signals.

Alternatives to Super Globals

The worst culprits for action at a distance relate to IO and exceptional conditions. Using `Try::Tiny` (*exception_caveats*) will help insulate you from the tricky semantics of proper exception handling. `local`izing and copying the value of `$!` can help you avoid strange behaviors when Perl makes implicit system calls. Using `IO::File` and its methods on lexical filehandles (*file_handling_variables*) helps prevent unwanted global changes to IO behavior.