Perl is a malleable language. You can write programs in whichever creative, maintainable, obfuscated, or bizarre fashion you prefer. Good programmers write code that they want to maintain, but Perl won't decide for you what *you* consider maintainable.

Perl isn't perfect. Some features are difficult to use correctly. Others seem great but don't work all that well. Some have strange edge cases. Knowing what to avoid in Perl--and when to avoid it--will help you write robust code that survives the twin tests of time and real users.

Barewords

Perl's parser understands Perl's builtins and operators. It uses sigils to identify variables and other punctuation to recognize function and method calls. Yet sometimes the parser has to guess what you mean, especially when you use a *bareword*--an identifier without a sigil or other syntactically significant punctuation.

Good Uses of Barewords

Though the `strict` pragma (*pragmas*) rightly forbids ambiguous barewords, some barewords are acceptable.

Bareword hash keys

Hash keys in Perl are usually *not* ambiguous because the parser can identify them as string keys; `pinball` in `$games{pinball}` is obviously a string.

Occasionally this interpretation is not what you want, especially when you intend to *evaluate* a builtin or a function to produce the hash key. To make these cases clear, pass arguments to the function, use parentheses, or prepend a unary plus to force the evaluation of the builtin:

```
# the literal 'shift' is the key
my $value = $items{B<shift>};

# the value produced by shift is the key
my $value = $items{B<shift @_>}

# the function returns the key
my $value = $items{B<myshift( @_ )>}

# unary plus indicates the builtin shift
my $value = $items{B<+>shift};
```

Bareword package names

Package names are also barewords. If your naming conventions rule that package names have initial capitals and functions do not, you'll rarely encounter naming collisions. Even still, Perl must determine how to parse `Package->method`. Does it mean "call a function named `Package()` and call `method()` on its return value?" or "Call a method named `method()` in the `Package` namespace?" The answer depends on the code Perl has already compiled when it encounters that method call.

Force the parser to treat `Package` as a package name by appending the package separator (`::`) or make it a literal string:

```
# probably a class method
Package->method;

# definitely a class method
PackageB<::>->method;

# a slightly less ugly class method
B<'>PackageB<'>->method;

# package separator
my $q = Plack::Request::->new;

# unambiguously a string literal
my $q = 'Plack::Request'->new;
```

Almost no real code does this, but it's unambiguous, so be able to read it.

Bareword named code blocks

The special named code blocks `AUTOLOAD`, `BEGIN`, `CHECK`, `DESTROY`, `END`, `INIT`, and `UNITCHECK` are barewords which *declare* functions without the `sub` builtin. You've seen this before (*code_generation*):

```
package Monkey::Butler;

BEGIN { initialize_simians( __PACKAGE__ ) }

sub AUTOLOAD { ... }
```

While you *can* declare `AUTOLOAD()` without using `sub`, few people do.

Bareword constants

Constants declared with the `constant` pragma are usable as barewords:

```
# don't use this for real authentication
use constant NAME     => 'Bucky';
use constant PASSWORD => '|38fish!head74|';

return unless $name eq NAME && $pass eq PASSWORD;
```

These constants do *not* interpolate in double-quoted strings.

Constants are a special case of prototyped functions (*prototypes*). When you predeclare a function with a prototype, the parser will treat all subsequent uses of that bareword specially--and will warn about ambiguous parsing errors. All other drawbacks of prototypes still apply.

Ill-Advised Uses of Barewords

No matter how cautiously you code, barewords still produce ambiguous code. You can avoid the worst abuses, but you will encounter several types of barewords in legacy code.

Bareword hash values

Some old code may not take pains to quote the *values* of hash pairs:

```
# poor style; do not use
my %parents = (
    mother => Annette,
    father => Floyd,
);
```

When neither the `Floyd()` nor `Annette()` functions exist, Perl will interpret these barewords as strings. `strict 'subs'` will produce an error in this situation.

Bareword function calls

Code written without `strict 'subs'` may use bareword function names. Adding parentheses will make the code pass strictures. Use `perl -MO=Deparse,-p` (see `perldoc B::Deparse`) to discover how Perl parses them, then parenthesize accordingly.

Bareword filehandles

Prior to lexical filehandles (*lexical_filehandles*), all file and directory handles used barewords. You can almost always safely rewrite this code to use lexical filehandles. Perl's parser recognizes the special exceptions of `STDIN`, `STDOUT`, and `STDERR`.

Bareword sort functions

The second operand of the `sort` builtin can be the *name* of a function to use for sorting. While this is rarely ambiguous to the parser, it can confuse *human* readers. The alternative of providing a function reference in a scalar is little better:

```
# bareword style
my @sorted = sort compare_lengths @unsorted;

# function reference in scalar
my $comparison = \&compare_lengths;
my @sorted     = sort $comparison @unsorted;
```

The second option avoids the use of a bareword, but the result is longer. Unfortunately, Perl's parser *does not* understand the single-line version due to the special parsing of `sort`; you cannot use an arbitrary expression (such as taking a reference to a named function) where a block or a scalar might otherwise go.

```
# does not work
my @sorted = sort \&compare_lengths @unsorted;
```

In both cases, the way `sort` invokes the function and provides arguments can be confusing (see `perldoc -f sort` for the details). Where possible, consider using the block form of `sort` instead. If you must use either function form, add a comment about what you're doing and why.

Indirect Objects

Perl is not a pure object-oriented language. It has no operator `new`; a constructor is anything which returns an object. By convention, constructors are class methods named `new()`, but you can name

these methods anything you want, or even use *functions*. Several old Perl OO tutorials promote the use of C++ and Java-style constructor calls:

```
my $q = B<new> Alces; # DO NOT USE
```

... instead of the obvious method call:

```
my $q = Alces->new;
```

These examples produce equivalent behavior, except when they don't.

Bareword Indirect Invocations

In the indirect object form (more precisely, the *dative* case) of the first example, the method precedes the invocant. This is fine in spoken languages where verbs and nouns are more obvious, but it introduces parsing ambiguities in Perl.

Because the method's name is a bareword (*barewords*), the parser uses several heuristics to figure out the proper interpretation of this code. While these heuristics are well-tested and *almost* always correct, their failure modes are confusing. Things get worse when you pass arguments to a constructor:

```
my $obj = new Class( arg => $value ); # DO NOT USE
```

In this example, the *name* of the class looks like a function call. Perl can and does often get this right, but its heuristics depend on which package names the parser has seen, which barewords it has already resolved, how it resolved those barewords, and the *names* of functions already declared in the current package. For an exhaustive list of these conditions, you have to read the source code of Perl's parser--not something the average Perl programmer wants to do (see `intuit_method` in *toke.c*, if you're really curious--but feel free to forget this suggestion ever existed).

Imagine running afoul of a prototyped function (*prototypes*) with a name which just happens to conflict somehow with the name of a class or a method called indirectly, such as a poorly-named `JSON()` method in the same file where the `JSON` module is used, to pick an example that actually happened. This is rare, but it's very unpleasant to debug. Avoid indirect invocations instead.

Indirect Notation Scalar Limitations

Another danger of the indirect syntax is that the parser expects a single scalar expression as the object. Printing to a filehandle stored in an aggregate variable *seems* obvious, but it is not:

```
# DOES NOT WORK
say $config->{output} 'Fun diagnostic message!';
```

Perl will attempt to call `say` on the `$config` object.

`print`, `close`, and `say`--all builtins which operate on filehandles--operate in an indirect fashion. This was fine when filehandles were package globals, but lexical filehandles (*lexical_filehandles*) make the indirect object syntax problems obvious. To solve this, disambiguate the subexpression which produces the intended invocant:

```
say B<{>$config->{output}B<}> 'Fun diagnostic message!';
```

Alternatives to Indirect Notation

Direct invocation notation does not suffer this ambiguity problem. To construct an object, call the constructor method on the class name directly:

```
my $q   = Plack::Request->new;
```

```
my $obj = Class->new( arg => $value );
```

This syntax *still* has a bareword problem in that if you have a function named `Request` in the `Plack` namespace, Perl will interpret the bareword class name as a call to the function, as:

```
sub Plack::Request;

# you wrote Plack::Request->new, but Perl saw
my $q = Plack::Request()->new;
```

Disambiguate this syntax as usual (*bareword_package_names*).

For the limited case of filehandle operations, the dative use is so prevalent that you can use the indirect invocation approach if you surround your intended invocant with curly brackets. You *can* use methods on lexical filehandles, though almost no one ever does this for `print` and `say`.

The CPAN module `Perl::Critic::Policy::Dynamic::NoIndirect` (a plugin for `Perl::Critic`) can analyze your code to find indirect invocations. The CPAN module `indirect` can identify and prohibit their use in running programs:

```
# warn on indirect use
no indirect;

# throw exceptions on their use
no indirect ':fatal';
```

Prototypes

A *prototype* is a piece of metadata attached to a function or variable. A function prototype changes how Perl's parser understands it.

Prototypes allow you to define your own functions which behave like builtins. Consider the builtin `push`, which takes an array and a list. While Perl would normally flatten the array and list into a single list passed to `push`, Perl knows to treat the array as a *container* and does not flatten its values. In effect, this is like passing a reference to an array and a list of values to `push`--because Perl's parser understands this is what `push` needs to do.

Function prototypes attach to function declarations:

```
sub foo        B<(&@)>;
sub bar        B<($$)> { ... }
my  $baz = sub B<(&&)> { ... };
```

Any prototype attached to a forward declaration must match the prototype attached to the function declaration. Perl will give a warning if this is not true. Strangely you may omit the prototype from a forward declaration and include it for the full declaration--but the only reason to do so is to win a trivia contest.

The builtin `prototype` takes the name of a function and returns a string representing its prototype.

To see the prototype of a builtin, prepend `CORE::` to its name for `prototype`'s operand:

```
$ B<perl -E "say prototype 'CORE::push';">
\@@
$ B<perl -E "say prototype 'CORE::keys';">
\%
$ B<perl -E "say prototype 'CORE::open';">
*;$@
```

`prototype` will return `undef` for those builtins whose functions you cannot emulate:

```
B<say prototype 'CORE::system' // 'undef'>
# undef; cannot emulate builtin C<system>

B<say prototype 'CORE::prototype' // 'undef'>
# undef; builtin C<prototype> has no prototype
```

Remember `push`?

```
$ B<perl -E "say prototype 'CORE::push';">
\@@
```

The `@` character represents a list. The backslash forces the use of a *reference* to the corresponding argument. This prototype means that `push` takes a reference to an array and a list of values. You might write `mypush` as:

```
sub mypush (\@@) {
    my ($array, @rest) = @_;
    push @$array, @rest;
}
```

Other prototype characters include `$` to force a scalar argument, `%` to mark a hash (most often used as a reference), and `&` to identify a code block. See `perldoc perlsub` for more information.

The Problem with Prototypes

Prototypes change how Perl parses your code and how Perl coerces arguments passed to your functions. While these prototypes may superficially resemble function signatures (*function_signatures*) in other languages, they are very different. They do not document the number or types of arguments functions expect, nor do they map arguments to named parameters.

Prototype coercions work in subtle ways, such as enforcing scalar context on incoming arguments:

```
sub numeric_equality($$) {
    my ($left, $right) = @_;
    return $left == $right;
}

my @nums = 1 .. 10;

say 'They're equal, whatever that means!' if numeric_equality @nums,
10;
```

... but only work on simple expressions:

```
sub mypush(\@@);

# compilation error: prototype mismatch
# (expects array, gets scalar assignment)
```

```
mypush( my $elems = [], 1 .. 20 );
```

To debug this, users of `mypush` must know both that a prototype exists, and the limitations of the array prototype. That's a lot of cognitive burden to put on a user--and if you think this error message is inscrutable, wait until you see the *complicated* prototype errors.

Good Uses of Prototypes

Prototypes *do* have a few good uses that outweigh their problems. For example, you can use a prototyped function to override one of Perl's builtins. First check that you *can* override the builtin by examining its prototype in a small test program. Then use the `subs` pragma to tell Perl that you plan to override a builtin. Finally declare your override with the correct prototype:

```
use subs 'push';

sub push (\@@) { ... }
```

Beware that the `subs` pragma is in effect for the remainder of the *file*, regardless of any lexical scoping.

You may also usefully use prototypes to define compile-time constants. When Perl encounters a function declared with an empty prototype (as opposed to *no* prototype) *and* this function evaluates to a single constant expression, the optimizer will turn all calls to that function into constants instead of function calls:

```
sub PI () { 4 * atan2(1, 1) }
```

All subsequent code will use the calculated value of pi in place of the bareword `PI` or a call to `PI()`, with respect to scoping and visibility.

The core pragma `constant` handles these details for you. The `Const::Fast` module from the CPAN creates constant scalars which you can interpolate into strings.

A reasonable use of prototypes is to extend Perl's syntax to operate on anonymous functions as blocks. The CPAN module `Test::Exception` uses this to good effect to provide a nice API with delayed computation. Its `throws_ok()` function takes three arguments: a block of code to run, a regular expression to match against the string of the exception, and an optional description of the test:

```
use Test::More;
use Test::Exception;

throws_ok
    { my $unobject; $unobject->yoink }
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail';

done_testing();
```

The exported `throws_ok()` function has a prototype of `&$;$`. Its first argument is a block, which becomes an anonymous function. The second argument is a scalar. The third argument is optional.

Careful readers may have spotted the absence of a comma after the block. This is a quirk of Perl's parser, which expects whitespace after a prototyped block, not the comma operator. This is a

drawback of the prototype syntax. If that bothers you, use `throws_ok()` without taking advantage of the prototype:

```
use Test::More;
use Test::Exception;

throws_okB<(>
    B<sub> { my $unobject; $unobject->yoink() }B<,>
    qr/Can't call method "yoink" on an undefined/,
    'Method on undefined invocant should fail' B<)>;

done_testing();
```

`Test::Fatal` allows similar testing and uses a simpler approach to avoid this ambiguity.

Ben Tilly suggests a final good use of prototypes, to define a custom named function to use with `sort`:

```
sub length_sort ($$) {
    my ($left, $right) = @_;
    return length($left) <=> length($right);
}

my @sorted = sort length_sort @unsorted;
```

The prototype of $$ forces Perl to pass the sort pairs in `@_`. `sort`'s documentation suggests that this is slightly slower than using the package globals $a and $b, but using lexical variables often makes up for any speed penalty.

Method-Function Equivalence

Perl's object system is deliberately minimal (*blessed_references*). A class is a package, and Perl does not distinguish between a function and a method stored in a package. The same builtin, `sub`, declares both. Perl will happily dispatch to a function called as a method. Likewise, you can invoke a method as if it were a function--fully-qualified, exported, or as a reference--if you pass in your own invocant manually.

Invoking the wrong thing in the wrong way causes problems.

Caller-side

Consider a class with several methods:

```
package Order {

    use List::Util 'sum';

    sub calculate_price {
        my $self = shift;
        return sum( 0, $self->get_items );
    }

    ...
```

```
}
```

Given an `Order` object `$o`, the following invocations of this method *may* seem equivalent:

```
my $price = $o->calculate_price;

# broken; do not use
my $price = Order::calculate_price( $o );
```

Though in this simple case, they produce the same output, the latter violates object encapsulation by bypassing method lookup.

If `$o` were instead a subclass or allomorph (*roles*) of `Order` which overrode `calculate_price()`, that example has just called the wrong method. Any change to the implementation of `calculate_price()`, such as a modification of inheritance or delegation through `AUTOLOAD()` --might break calling code.

Perl has one circumstance where this behavior may seem necessary. If you force method resolution without dispatch, how do you invoke the resulting method reference?

```
my $meth_ref = $o->can( 'apply_discount' );
```

There are two possibilities. The first is to discard the return value of the `can()` method:

```
$o->apply_discount if $o->can( 'apply_discount' );
```

The second is to use the reference itself with method invocation syntax:

```
if (my $meth_ref = $o->can( 'apply_discount' )) {
    $o->B<$meth_ref>();
}
```

When `$meth_ref` contains a function reference, Perl will invoke that reference with `$o` as the invocant. This works even under strictures, as it does when invoking a method with a scalar containing its name:

```
my $name = 'apply_discount';
$o->B<$name>();
```

There is one small drawback in invoking a method by reference; if the structure of the program changes between storing the reference and invoking the reference, the reference may no longer refer to the most appropriate method. If the `Order` class has changed such that `Order::apply_discount` is no longer the right method to call, the reference in `$meth_ref` will not have updated.

That's an unlikely circumstance, but limit the scope of a method reference when you use this invocation form just in case.

Callee-side

Because it's *possible* (however inadvisable) to invoke a given function as a function or a method, it's possible to write a function callable as either.

The `CGI` module has these two-faced functions. Every one of them must apply several heuristics to determine whether the first argument is an invocant. This causes problems. It's difficult to predict

exactly which invocants are potentially valid for a given method, especially when you may have to deal with subclasses. Creating an API that users cannot easily misuse is more difficult too, as is your documentation burden. What happens when one part of the project uses the procedural interface and another uses the object interface?

If you *must* provide a separate procedural and OO interface to a library, create two separate APIs.

Automatic Dereferencing

Perl can automatically dereference certain references on your behalf. Given an array reference in `$arrayref`, you can write:

```
push $arrayref, qw( list of values );
```

Given an expression which returns an array reference, you can do the same:

```
push $houses{$location}[$closets], \@new_shoes;
```

After Perl 5.18, you must enable this feature with `use experimental 'autoderef';`. That should be a sign to tread carefully here.

The same goes for the array operators `pop`, `shift`, `unshift`, `splice`, `keys`, `values`, and `each` and the hash operators `keys`, `values`, and `each`. If the reference provided is not of the proper type--if it does not dereference properly--Perl will throw an exception. While this may seem more dangerous than explicitly dereferencing references directly, it is in fact the same behavior:

```
my $ref = sub { ... };

# will throw an exception
push  $ref, qw( list of values );

# will also throw an exception
push @$ref, qw( list of values );
```

Unfortunately, this automatic dereferencing has two problems. First, it only works on plain variables. If you have a `blessed` array or hash, a `tied` hash, or an object with array or hash overloading, Perl will throw a runtime exception instead of dereferencing the reference.

Second, remember that `each`, `keys`, and `values` can operate on both arrays and hashes. You can't look at:

```
my @items = each $ref;
```

... and tell whether `@items` contains a list of key/value pairs or index/value pairs, because you don't know whether you should expect `$ref` to refer to a hash or an array. Yes, choosing good variable names will help, but this code is intrinsically confusing.

Neither of these drawbacks make this syntax *unusable* in general, but its rough edges and potential for confusing readers make it less useful than it could be.

Tie

Where overloading (*overloading*) allows you to customize the behavior of classes and objects for specific types of coercion, a mechanism called *tying* allows you to customize the behavior of primitive

variables (scalars, arrays, hashes, and filehandles). Any operation you might perform on a tied variable translates to a specific method call on an object.

For example, the `tie` builtin allows core `Tie::File` module to treat files as if they were arrays of records, letting you `push` and `pop` and `shift` as you see fit. (`tie` was intended to use file-backed stores for hashes and arrays, so that Perl could use data too large to fit in available memory. RAM was more expensive 20 years ago.)

The class to which you `tie` a variable must conform to a defined interface for a specific data type. Read `perldoc perltie` for an overview, then see the core modules `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` for specific details. Start by inheriting from one of those classes, then override any specific methods you need to modify.

If `tie` weren't confusing enough, `Tie::Scalar`, `Tie::Array`, and `Tie::Hash` define the necessary interfaces to tie scalars, arrays, and hashes, but `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` provide the default implementations.

Tying Variables

To tie a variable:

```
use Tie::File;
tie my @file, 'Tie::File', @args;
```

The first operand is the variable to tie. The second is the name of the class into which to tie it. `@args` is an optional list of arguments required for the tying function. In the case of `Tie::File`, `@args` should contain a valid filename.

Tying functions resemble constructors: `TIESCALAR`, `TIEARRAY()`, `TIEHASH()`, or `TIEHANDLE()` for scalars, arrays, hashes, and filehandles respectively. Each function returns a new object which represents the tied variable. Both `tie` and `tied` return this object, though most people use `tied` in a boolean context.

Implementing Tied Variables

To implement the class of a tied variable, inherit from a core module such as `Tie::StdScalar`, Then override the specific methods for the operations you want to change. In the case of a tied scalar, these are likely `FETCH` and `STORE`, possibly `TIESCALAR()`, and probably not `DESTROY()`.

Here's a class which logs all reads from and writes to a scalar:

```
package Tie::Scalar::Logged {
    use Tie::Scalar;
    use parent -norequire => 'Tie::StdScalar';

    sub STORE {
        my ($self, $value) = @_;
        Logger->log("Storing <$value> (was [$$self])", 1);
        $$self = $value;
    }

    sub FETCH {
        my $self = shift;
        Logger->log("Retrieving <$$self>", 1);
        return $$self;
    }
}
```

Assume that the `Logger` class method `log()` takes a string and the number of frames up the call stack of which to report the location.

Within the `STORE()` and `FETCH()` methods, `$self` works as a blessed scalar. Assigning to that scalar reference changes the value of the scalar. Reading from it returns its value.

Similarly, the methods of `Tie::StdArray` and `Tie::StdHash` act on blessed array and hash references, respectively. Again, `perldoc perltie` explains the methods tied variables support, such as reading or writing multiple values at once.

The `-norequire` option prevents the `parent` pragma from attempting to load a file for `Tie::StdScalar`, as that module is part of the file *Tie/Scalar.pm*. That's right, there's no *.pm* file for `Tie::StdScalar`. Isn't this fun?

When to use Tied Variables

Tied variables seem like fun opportunities for cleverness, but they can produce confusing interfaces. Unless you have a very good reason for making objects behave as if they were builtin data types, avoid creating your own ties without good reason. `tie`d variables are also much slower than builtin data types.

With that said, tied variables can help you debug tricky code (use the logged scalar to help you understand *where* a value changes) or to make certain impossible things possible (access large files without running out of memory). Tied variables are less useful as the primary interfaces to objects; it's often too difficult and constraining to try to fit your whole interface to that supported by `tie()`.

A final word of warning is a sad indictment of lazy programming: a lot of code goes out of its way to *prevent* use of tied variables, often by accident. This is unfortunate, but library code is sometimes fast and lazy with what it expects, and you can't always fix it.