Perl 5 isn't perfect. Some features are difficult to use correctly. Otherwise have never worked well. A few are quirky combinations of other features with strange edge cases. While you're better off avoiding these features, knowing why to avoid them will help you find better solutions.

Barewords

Perl is a malleable language. You can write programs in the most creative, maintainable, obfuscated, or bizarre fashion you prefer. Maintainability is a concern of good programmers, but Perl doesn't presume to dictate what *you* consider maintainable.

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 5 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. In this case, disambiguate by providing arguments, using function argument parentheses, or prepending 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 @_>}

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

Bareword package names

Package names in Perl 5 are also barewords. If you hew to naming conventions where package names have initial capitals and functions do not, you'll rarely encounter naming collisions, but the Perl 5 parser must determine how to parse `Package->method()`. Does it mean "call a function named `Package()` and call `method()` on its return value?" or does it mean "Call a method named `method()` in the `Package` namespace?" The answer varies depending on what code the parser has already encountered in the current namespace.

Force the parser to treat `Package` as a package name by appending the package separator (`::`) Even among people who understand why this works, very few people do it.:

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

# definitely a class method
```

```
Package::->method();
```

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* elide `sub` from `AUTOLOAD()` declarations, 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;
```

Note that 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 knows how to treat that function 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 most uses, but you will encounter several types of barewords in legacy code.

Bareword function calls

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

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 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; the exceptions are `STDIN`, `STDOUT`, and `STDERR`. Fortunately, Perl's parser recognizes these.

Bareword sort functions

Finally, the `sort` builtin can take as its second argument 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 one line longer. Unfortunately, Perl 5'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, consider adding an explanatory comment.

Indirect Objects

Perl 5 has no operator `new`; a constructor in Perl 5 is anything which returns an object. By convention, constructors are class methods named `new()`, but you can choose anything you like. Several old Perl 5 object tutorials promote the use of C++ and Java-style constructor calls:

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

... instead of the obvious method call:

```
my $q = CGI->new();
```

These syntaxes 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 verb (the method) precedes the noun to which it refers (the object). This is fine in spoken languages, but it introduces parsing ambiguities in Perl 5.

As the method name is a bareword (*barewords*), the parser must divine the proper interpretation of the code through the use of several heuristics. While these heuristics are well-tested and *almost* always correct, their failure modes are confusing. Worse yet, they depend on the order of compilation of code and modules.

Parsing difficulty increases when the constructor takes arguments. The indirect style may resemble:

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

... thus making the name of the class look like a function call. Perl 5 *can* disambiguate many of these cases, but its heuristics depend on which package names the parser has seen, which barewords it has already resolved (and how it resolved them), and the *names* of functions already declared in the current package.

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. This is rare, but so unpleasant to debug that it's worth avoiding indirect invocations.

Indirect Notation Scalar Limitations

Another danger of the 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 AS WRITTEN
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   = CGI->new();
my $obj = Class->new( arg => $value );
```

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

```
sub CGI;

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

While this happens rarely, you can disambiguate classnames by appending the package separator (`::`) or by explicitly marking class names as string literals:

```
# package separator
my $q = CGI::->new();

# unambiguously a string literal
```

```
my $q = 'CGI'->new();
```

Yet almost no one ever does this.

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. If you're using Perl 5.14 (or if you load `IO::File` or `IO::Handle`), you can use methods on lexical filehandles Almost no one does this for `print` and `say` though..

The CPAN module `Perl::Critic::Policy::Dynamic::NoIndirect` (a plugin for `Perl::Critic`) can identify indirect invocations during code reviews. 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 optional metadata attached to a function which changes the way the parser understands its arguments. While they may superficially resemble function signatures in other languages, they are very different.

Prototypes allow users to define their own functions which behave like builtins. Consider the builtin `push`, which takes an array and a list. While Perl 5 would normally flatten the array and list into a single list passed to `push`, the parser knows not to flatten the array so that `push` can modify it in place.

Function prototypes are part of 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 there's no reason to do so.

The builtin `prototype` takes the name of a function and returns a string representing its prototype. Use the `CORE::` form to see the prototype of a builtin:

```
$ 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 full documentation.

The Problem with Prototypes

Prototypes change how Perl parses your code and can cause argument type coercions. 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
# (expected array, got 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. Worse yet, these are the *simple* errors prototypes can cause.

Good Uses of Prototypes

Few uses of prototypes are compelling enough to overcome their drawbacks, but they exist.

First, they can allow you to override 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, and 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.

The second reason to use prototypes is 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 computationSee also `Test::Fatal`. 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 tests => 1;
use Test::Exception;

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

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 the Perl 5 parser, which expects whitespace after a prototyped block, not the comma operator. This is a drawback of the prototype syntax.

You may use `throws_ok()` without taking advantage of the prototype:

```
use Test::More tests => 1;
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<)>;
```

A final good use of prototypes is when defining a custom named function to use with `sort`Ben Tilly suggested this example.:

```
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 5's object system is deliberately minimal (*blessed_references*). Because a class is a package, Perl does not distinguish between a function and a method stored in a package. The same builtin, `sub`, declares both. Documentation can clarify your intent, but 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

avoiding method lookup.

If $o were instead a subclass or allomorph (*roles*) of `Order` which overrode `calculate_price()`, bypassing method dispatch would call 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' ))
{
    $o->$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';
$o->$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.

When you use this invocation form, limit the scope of the references.

Callee-side

Because Perl 5 makes no distinction between functions and methods at the point of declaration and 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 core `CGI` module is a prime offender. Its functions apply several heuristics to determine whether their first arguments are invocants.

The drawbacks are many. 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.

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.

The `tie` builtin originally allowed you to use disk space as the backing memory for hashes, so that Perl could access files larger than could easily fit in memory. The core module `Tie::File` provides a similar system, and allows you to treat files as if they were arrays.

The class to which you `tie` a variable must conform to a defined interface for a specific data type. See `perldoc perltie` for an overview, then consult 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 argument is the variable to tie, the second is the name of the class into which to tie it, and `@args` is an optional list of arguments required for the tying function. In the case of `Tie::File`, this is 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 the `tie` and `tied` builtins return this object. Most people use `tied` in a boolean context, however.

Implementing Tied Variables

To implement the class of a tied variable, inherit from a core module such as `Tie::StdScalar` `Tie::StdScalar` lacks its own *.pm* file, so use `Tie::Scalar` to make it available., 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()`.

You can create a class which logs all reads from and writes to a scalar with very little code:

```
package Tie::Scalar::Logged
{
    use Modern::Perl;

    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;
        }
    }

    1;
```

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 and reading from it returns its value.

Similarly, the methods of `Tie::StdArray` and `Tie::StdHash` act on blessed array and hash references, respectively. The `perldoc perltie` documentation explains the copious methods they support, as you can read or write multiple values from them, among other operations.

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*.

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. `tie` is also much slower than using the builtin types due to various reasons of implementation.

Good reasons include to ease debugging (use the logged scalar to help you understand where a value changes) and to make certain impossible operations possible (accessing large files in a memory-efficient way). 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()`.

The final word of warning is both sad and convincing; too much code goes out of its way to *prevent* use of tied variables, often by accident. This is unfortunate, but violating the expectations of library code tends to reveal bugs that are often out of your power to fix.