What to Avoid

Perl 5 isn't perfect. Some features seemed like good ideas at the time, but they're difficult to use correctly. Others don't work as anyone might expect. A few more are simply bad ideas. These features will likely persist--removing a feature from Perl is a serious process reserved for only the most egregious offenses--but you can and should avoid them in almost every case.

Barewords

Perl uses sigils and other punctuation pervasively to help both the parser and the programmer identify the categories of named entities. Even so, Perl is a malleable language. You can write programs in the most creative, maintainable, obfuscated, or bizarre fashion as you prefer. Maintainability is a concern of good programs, but the developers of Perl itself don't presume to dictate what *you* find most maintainable.

Perl's parser understands the built-in Perl keywords and operators; it knows that `bless()` means you're making objects (*blessed_references*). These are rarely ambiguous... but Perl programmers can add complexity to parsing by using *barewords*. A bareword is an identifier without a sigil or other attached disambiguation as to its intended syntactical function. Because there's no Perl 5 keyword `curse`, the literal word `curse` appearing in source code is ambiguous. Did you intend to use a variable `$curse` or to call a function `curse()`? The `strict` pragma warns about use of such ambiguous barewords for good reason.

Even so, barewords are permissible in several places in Perl 5 for good reason.

Good Uses of Barewords

Hash keys in Perl 5 are barewords. These are usually not ambiguous because their use as keys is sufficient for the parser to identify them as the equivalent of single-quoted strings. Yet be aware that attempting to evaluate a function call or a built-in operator (such as `shift`) to *produce* a hash key may not do what you expect, unless you disambiguate.

```
# 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 is also sufficient to disambiguate
my $value = $items{B<+>shift};
```

Package names in Perl 5 are barewords in a sense. Good naming conventions for packages (initial caps) help prevent unwanted surprises, but the parser uses a heuristic to determine whether `Package->method()` means to call a function named `Package()` and then call the `method()` method on its results or whether to treat `Package` as the name of a package. You can disambiguate this with the postfix package separator (`::`), but that's rare and admittedly ugly:

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

# definitely a class method
Package::->method();
```

The special named code blocks provide their own types of barewords. `AUTOLOAD`, `BEGIN`, `CHECK`, `DESTROY`, `END`, `INIT`, and `UNITCHECK` *declare* functions, but they do not need the `sub` keyword to do so. You may be familiar with the idiom of writing `BEGIN` without `sub`:

```
    package Monkey::Butler;

    BEGIN { initialize_simians( __PACKAGE__ ) }
```

You *can* leave off the `sub` on `AUTOLOAD()` declarations, but that's uncommon.

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 PASS;
```

Be aware that these constants do *not* interpolate in interpolation contexts such as double-quoted strings.

Constants are a special case of prototyped functions (*prototypes*). If you've pre-declared a prototype for a function, you may use that function as a bareword; Perl 5 knows everything it needs to know to parse all occurrences of that function appropriately. Note that all of the drawbacks of prototypes still apply.

Ill-Advised Uses of Barewords

Barewords should be rare in modern Perl code; their ambiguity produces fragile code. You can avoid them in almost every case, but you may encounter several poor uses of barewords in legacy code.

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`.

Code written without `strict 'subs'` in effect may use bareword function names. You may safely parenthesize the argument lists to these functions without changing the intent of the codeUse `perl -MO=Deparse,-p` to discover how Perl parses them, then parenthesize accordingly..

Along similar lines, old code may not take pains to quote the *values* of hash pairs appropriately:

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

Because neither the `Floyd()` nor `Annette()` functions exist, Perl parses these hash values as strings. The `strict 'subs'` pragma makes the parser give an error in this situation.

Finally, the built-in `sort` operator can take as its second argument the *name* of a function to use for sorting. Instead provide a *reference* to the function to use for sorting to avoid the use of barewords:

```
# poor style; do not use
my @sorted = sort compare_lengths @unsorted;

# better style
my $comparison = \&compare_lengths;
my @sorted     = sort $comparison @unsorted;
```

The result is one line longer, but it avoids the use of a bareword. Unlike other bareword examples, Perl's parser needs no disambiguation for this syntax. There is only one way for it to interpret `compare_lengths`. Even so, the clarity of an explicit reference can help human readers.

Even so, Perl 5's parser does not understand the single-line version:

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

This is 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.

Indirect Objects

A constructor in Perl 5 is anything which returns an object; `new` is not a built-in function. By convention, constructors are class methods named `new()`, but you have the flexibility to choose a different approach to meet your needs. 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 unambiguous:

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

These syntaxes are equivalent in behavior, except when they're not.

The first form is the indirect object form (more precisely, the *dative* case), where 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.

Bareword Indirect Invocations

One problem is that the name of the method is a bareword (*barewords*). The parser must perform several heuristics to determine the proper interpretation. While these heuristics are well-tested and *almost* always correct, their failure modes are confusing. Worse, they're fragile in the face of the *order* of compilation and module loading.

Parsing is more difficult for humans *and* the computer when the constructor takes arguments. The indirect style may resemble:

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

... thus making the classname `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 at the current point in the parse, 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 function with (prototypes) *prototypes* with a name which just happens to conflict somehow with the name of a class or a method called indirectly. This is infrequent, but so difficult to debug that avoiding this syntax is always worthwhile.

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} "This is a diagnostic message!";
```

`print`, `close`, and `say`--all keywords 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. In the previous example, Perl will try to call the `say` method on the `$config` object. The solution is to disambiguate the expression which produces the intended invocant:

```
say B<{>$config->{output}B<}> "This is a 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 );
```

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. Alternately, consider loading the core `IO::Handle` module which allows you to perform IO operations by calling methods on filehandle objects (such as lexical filehandles).

For supreme paranoia, you may disambiguate class method calls further by appending `::` to the end of class names, such as `CGI::->new()`. Very little code does this in practice, however.

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 declaration. Novices commonly assume that these prototypes serve as function signatures. They do not; instead they serve two separate purposes. They offer hints to the parser to change the way it parses functions and their arguments. They also modify the way Perl 5 handles arguments to those functions.

To declare a function prototype, add it after the name:

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

You may add prototypes to function forward declarations. You may also omit them from forward declarations. If you use a forward declaration with a prototype, that prototype must be present in the full function declaration; Perl will give a prototype mismatch warning if not. The converse is not true: you may omit the prototype from a forward declaration and include it for the full declaration.

There's little reason to omit the prototype from a forward declaration except for the desire to write too-clever code.

The original intent of prototypes was to allow users to define their own functions which behaved like (certain) built-in operators. Consider the behavior of the `push` operator, which takes an array and a list. While Perl 5 would normally flatten the array and list into a single list at the call site, the Perl 5 parser knows that a call to `push` must effectively pass the array as a single unit so that `push` can operate on the array in place.

The builtin `prototype` takes the name of a function and returns a string representing its prototype. To see the prototype of a built-in keyword, use the `CORE::` form:

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

Some builtins have prototypes you cannot emulate. In these cases, `prototype` will return `undef`:

```
$ B<perl -E "say prototype 'CORE::system' // 'undef' ">
undef
# You can't emulate builtin function C<system>'s calling convention.

$ B<perl -E "say prototype 'CORE::prototype' // 'undef' ">
undef
# Builtin function C<prototype> has no prototype.
```

Look at `push` again:

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

The `@` character represents a list. The backslash forces the use of a *reference* to the corresponding argument. Thus this function takes a reference to an array (because you can't take a reference to a list) and a list of values. `mypush` might be:

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

Valid prototype characters include $ to force a scalar argument, % to mark a hash (most often used as a reference), and & which marks a code block. See `perldoc perlsub` for full documentation.

The Problem with Prototypes

Prototypes can change the parsing of subsequent code and they can coerce the types of arguments. They don't serve as documentation to 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 do *not* work on anything more complex than a simple expression:

```
    sub mypush(\@@);

    # compilation error: prototype mismatch
    # (expected array, got scalar assignment)
    mypush( my $elems = [], 1 .. 20 );
```

Those aren't even the *subtler* kinds of confusion you can get from prototypes.

Good Uses of Prototypes

As long as code maintainers do not confuse them for full function signatures, prototypes have a few valid uses.

First, they are often necessary to emulate and override built-in keywords with user-defined functions. You must first check that you *can* override the built-in keyword by checking that `prototype` does not return `undef`. Once you know the prototype of the keyword, use a forward declaration of a function with the same name as the core keyword:

```
    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. A function declared with an empty prototype (as opposed to *no* prototype) which evaluates to a single expression becomes a constant in the Perl 5 optree rather than a function call:

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

After it processed that prototype declaration, the Perl 5 optimizer knows it should substitute the calculated value of pi whenever it encounters a bareword or parenthesized call to `PI` in the rest of the source code (with respect to scoping and visibility).

Rather than defining constants directly, the core `constant` pragma handles the details for you and may be clearer to read. If you want to interpolate constants into strings, the `Readonly` module from the CPAN may be more useful.

The final reason to use a prototype 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. Suppose that you want to test Perl 5's exception message when attempting to invoke a method on an undefined value:

```
    use Test::More tests => 1;
    use Test::Exception;

    throws_ok
        { my $not_an_object; $not_an_object->some_method() }
        qr/Can't call method "some_method" on an undefined value/,
        'Calling a method on an undefined invocant should throw exception';
```

The exported `throws_ok()` function has a prototype of `&$;$`. Its first argument is a block, which Perl upgrades to a full-fledged anonymous function. The second requirement is a scalar. The third argument is optional.

The most careful readers may have spotted a syntax oddity notable in its absence: there is no trailing comma after the end of the anonymous function passed as the first argument to `throws_ok()`. This is a quirk of the Perl 5 parser. Adding the comma causes a syntax error. The parser expects whitespace, not the comma operator.

The "no commas here" rule is a drawback of the prototype syntax.

You can use this API without the prototype. It's slightly less attractive:

```
    use Test::More tests => 1;
    use Test::Exception;

    throws_okB<(>
        B<sub> { my $not_an_object; $not_an_object->some_method() }B<,>
        qr/Can't call method "some_method" on an undefined value/,
        'Calling a method on an undefined invocant should throw
exception'B<)>;
```

A sparing use of function prototypes to remove the need for the `sub` keyword is reasonable. Few other uses of prototypes are compelling enough to overcome their drawbacks.

Ben Tilly suggests a fourth: when defining a custom function to use with `sort`. Declare this function with a prototype of `($$)` and Perl will pass its arguments in `@_` rather than the package globals $a and $b. This is a rare case, but it can save you time debugging.

Method-Function Equivalence

Perl 5's object system is deliberately minimal (*blessed_references*). Because a class is a package, Perl itself makes no strong distinction between a function stored in a package and a method stored in a package. The same keyword, `sub`, expresses both. Documentation and the convention of treating the first parameter as `$self` can imply intent to readers of the code, but Perl itself will treat any function of the appropriate name it can find in an appropriate package as a method if you try to call it 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.

Both have their problems; avoid them.

Caller-side

Suppose you have a class which contains several methods:

```
package Order;

use List::Util 'sum';

...

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

If you have 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 the encapsulation of objects in subtle ways. It avoids method lookup altogether.

If $o were *not* an `Order` object, but instead a subclass or allomorph (*roles*) of `Order` which overrode `calculate_price()`, the method-as-a-function call would call the *wrong* method. As well, if the internal implementation of `calculate_price()` were to change--perhaps inherited from elsewhere or delegated through `AUTOLOAD()`--the caller might break.

Perl has one circumstance where this behavior may seem necessary. If you force method resolution without performing 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 has changed in between storing the reference and invoking the reference, the reference may no longer refer to the current 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.

If you use this form of invocation, 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 manually inspect `@_` to determine whether the first argument is a likely invocant. If so, they perform particular manipulations to make sure that any object state the function needs to access is available. If the first argument is not a likely invocant, the function must consult global data elsewhere.

As with all heuristics, there are corner cases. It's difficult to predict exactly which invocants are potentially valid for a given method, especially when considering that users can create their own subclasses. The documentation burden is also greater--the explanation of a combined procedural and object interface must reflect the dichotomy of the code--as is the potential for misuse. What happens when one part of the project uses the procedural interface and another uses the object interface?

Providing separate procedural and object interfaces to a library may be justifiable. Some designs make some techniques more useful than others. Conflating the two into a single API will create a maintenance burden. Avoid it.

Tie

Overloading (*overloading*) lets you give classes custom behavior for specific types of coercions and accesses. A similar mechanism exists for making variables act like built-in types (scalars, arrays, and hashes), but with more specific behaviors. This mechanism uses the `tie` keyword; it is *tying*.

The original use of `tie` was to produce a hash stored on disk, rather than in memory. This allowed the use of dbm files from Perl, as well as the ability to access files larger than could fit in memory. The core module `Tie::File` provides a similar system by which to handle data files too large to fit in memory.

The class to which you `tie` a variable must conform to a specific, well-documented interface for the specific data type. `perldoc perltie` is the primary source of information about these interfaces, though the core modules `Tie::StdScalar`, `Tie::StdArray`, and `Tie::StdHash` are more useful in practice. Inherit from them to start, and override only those specific methods you need to modify.

The documentation and implementations of these parent classes are in the `Tie::Scalar`, `Tie::Array`, and `Tie::Hash` modules. Consequently you must also `use Tie::Scalar` if you want to inherit from `Tie::StdScalar`. If `tie()` hasn't confused you, the organization of this code might.

Tying Variables

Given a variable to tie, tie it with the syntax:

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

... where 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 the name of the file to which to tie the array.

The tying function resembles a constructor: `TIESCALAR`, `TIEARRAY()`, `TIEHASH()`, or `TIEHANDLE()` for scalars, arrays, hashes, and filehandles respectively. These functions return a new object which the `tie()` keyword returns as well. Most people ignore it.

The `tied()` operator returns the same object when used on a tied variable and `undef` otherwise. Again, few people use that object. Instead, they check the boolean value of this object when they use use `tied()` to determine whether the given variable is tied.

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, you probably need to override `FETCH` and `STORE`, may need to override `TIESCALAR()`, and probably can ignore `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. Be aware that `Tie::StdScalar` does not have its own *.pm* file, so you must use `Tie::Scalar` to make it available.

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 make for confusing interfaces in almost all cases, due mostly to their rarity. Unless you have a very good reason for making objects behave as if they were built-in data types, avoid creating your own ties.

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; far too much code does not expect to work with tied variables. Code which violates encapsulation may prohibit good and valid uses of cleverness. This is unfortunate, but violating the expectations of library code tends to reveal bugs that are often out of your power to fix.