# Writing Self-Modifying Perl

Spencer Tipping

December 25, 2010

# Contents

# Chapter 1

# Introduction

I've gotten a lot of WTF's[1] about self-modifying Perl scripts. Rightfully so, too. There's no documentation (until now), the interface is opaque and not particularly portable, and they aren't even very human-readable when edited:

```
...
meta::define_form 'meta', sub {
  my ($name, $value) = @_;
  meta::eval_in($value, "meta::$name");
};
meta::meta('configure', <<'__25976e07665878d3fae18f050160343f');
# A function to configure transients. Transients can be used to store any number of
# different things, but one of the more common usages is type descriptors.
sub meta::configure {
  my ($datatype, %options) = @_;
  $transient{$_}{$datatype} = $options{$_} for keys %options;
}
__25976e07665878d3fae18f050160343f
...
```

Despite these shortcomings, though, I think they're fairly useful. So rather than vindicate the idea (which is probably irredeemable), I've written this guide to dive into the mayhem and go from zero to a self-modifying Perl script. At the end, you'll have a script that is functionally equivalent to the `object` script, which I use as the prototype for all of the other ones.[2]

This guide probably isn't for the faint of heart, but if you're not afraid of `eval` then you might like it.

[1] http://www.osnews.com/story/19266/WTFs_m

[2] See http://github.com/spencertipping/perl-objects for the full source.

# Chapter 2

# A Big Quine

At the core of things, a self-modifying Perl script is just a big quine.[1] There are only two real differences:

1. Self-modifying Perl scripts print into their own files rather than to standard output.
2. They print modified versions of themselves, not the original source.

If we're going to write such a script, it's good to start with a simple quine.

## 2.1 A Basic Quine

Some languages make quine-writing easier than others. Perl actually makes it very simple. Here's one:

Listing 2.1 `examples/basic-quine`

```
1 my $code = <<'EOF';
2 print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;
3 EOF
4 print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;
```

The logic is fairly straightforward, though it may not look like it. We're quoting a bunch of stuff using `<<'EOF'`,[2] and storing that into a string. We then put the quoted content outside of the heredoc to let it execute. The duplication is necessary; we want to quote the content and then run it.[3] The key is this line:

```
print 'my $code = <<\'EOF\';', "\n", $code, "EOF\n"; print $code;
```

This code prints the setup to define a new variable `$code` and prints its existing content after that.

[1] A "quine" being a program that prints its own source.

[2] The single-quoted heredoc form doesn't do any interpolation inside the document, which is ideal since we don't want to worry about escaping stuff.

[3] Later on I'll use `eval` to reduce the amount of duplication.

## 2.2 Reducing Duplication

We don't want to write everything in our quine twice. Rather, we want to store most stuff just once and have a quine that scales well. The easiest way to do this is to use a hash to store the state, and serialize each key of the hash in the self-printing code. So instead of creating `$code`, we'll create `%data`:

Listing 2.2 `examples/data-quine`

```
my %data;
$data{code} = <<'EOF';
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
EOF
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
```

This is a good start. Here's how to add attributes without duplication:

Listing 2.3 `examples/data-quine-with-field`

```
my %data;
$data{foo} = <<'EOF';
a string
EOF
$data{code} = <<'EOF';
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
EOF
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{code};
```

## 2.3 Using `eval`

The business about duplicating `$data{code}` is easily remedied by just `eval`ing `$data{code}` at the end. This requires the `eval` section to be duplicated, but it's smaller than `$data{code}`. Here's the quine with that transformation:[4]

Listing 2.4 `examples/data-quine-with-eval`

```
my %data;
$data{foo} = <<'EOF';
```

[4] Note that these quines might not actually print themselves identically due to hash-key ordering. This is fine; all of the keys are printed before we use them.

```
a string
EOF
$data{code} = <<'EOF';
print 'my %data;', "\n";
print '$data{', $_, '} = <<\'EOF\';', "\n$data{$_}EOF\n" for keys %data;
print $data{bootstrap};
EOF
$data{bootstrap} = <<'EOF';
eval $data{code};
EOF
eval $data{code};
```

The advantage to this approach is that all we'll ever have to duplicate is `eval $data{code}` and `my %data;`, which is fairly trivial. It's important that you understand what's going on here, since this idea is integral to everything going forward.