

Filehandle Ties

Robby Walker, webmaster@cd-lab.com

Abstract

Perl can tie scalars, arrays, hashes, and filehandles to a user-defined class, to seamlessly extend its abilities without changing its syntax. Code that operates on a normal variable works equally well on a tied variable. I examine output filehandle ties by writing 3 filehandle tie modules.

1 Introduction

Tie-ing a filehandle is the same as tie-ing any other variable type.¹ In code listing 16 I tie the HANDLE filehandle to the fictitious module Module::Name. After the tie, when I use HANDLE like a filehandle, Perl will use Module::Name to determine what to do. The tie() command is a constructor since it returns an object². I do not need to use or save this object since I can simply use the filehandle, but sometimes the object is useful, as I show later.

Code Listing 16: Tie-ing a filehandle

```
1 $object = tie *HANDLE, 'Module::Name', args,...
```

2 Writing Filehandle Tie Modules

Filehandle tie modules must at least implement the special TIEHANDLE and at least one of the input or output methods. A tied filehandle can respond to any method that a normal filehandle can, although I have to implement the right method to do that. Table 2 shows a summary of filehandle operations and the corresponding methods in the tied module.

2.0.1 Streamlining Tie Modules

I wrote Tie::FileHandle::Base as a base class to provide default implementations of many of the above methods. For example, in code listing 17 I implemented PRINTF to rely on PRINT. If I do not want that, I simply implement another PRINTF in my derived class.

Code Listing 17: Tie::FileHandle::Base PRINTF implementation

```
1 sub PRINTF { my $self = shift; $self->PRINT(sprintf @_) }
```

¹See the perltie man page for details on the basic of tie()

²I can also get this object from the tied variable with the tied() command: `$object = tied *HANDLE`

Table 1: Tie filehandle equivalencies

Tied Handle	Equivalent To
tie HANDLE, 'Module', @args	\$object = Module->TIEHANDLE(@args)
open HANDLE @args	\$object->OPEN(@args)
binmode HANDLE \$mode	\$object->BINMODE(\$mode)
print HANDLE @args	\$object->PRINT(@args)
printf HANDLE @args	\$object->PRINTF(@args)
syswrite HANDLE @args	\$object->WRITE(@args)
getc HANDLE	\$object->GETC
readline HANDLE	\$object->READLINE
<HANDLE>	\$object->READLINE
read HANDLE @args -OR- sysread HANDLE @args	\$object->READ(@args)
seek HANDLE @args	\$object->SEEK(@args)
tell HANDLE	\$object->TELL
fileno HANDLE	\$object->FILENO
eof HANDLE	\$object->EOF
close HANDLE	\$object->CLOSE

3 Output Multiplexing

In many cases I want to send output of a program not only to standard output but also to a file. However, if I have already written my code it would be a painful and tedious task to replace every print statement with two (one for STDOUT and one for the file), and my code also becomes more difficult to maintain. I could concatenate the data into a string then print it twice at the very end of my program, but this is also tedious (a problem I address later) since I must be careful to lead all paths to those final two print statements.

Although either of the above options is manageable they lack the elegance and simplicity that makes Perl fun. A tied filehandle can take care of this behind-the-scenes so my code does not have to change. In code listing 18 I create a small module, `Tie::FileHandle::MultiPlex`, whose `PRINT` method can simultaneously send output to many filehandles. It uses `Tie::FileHandle::Base` which provides default implementations of the other filehandle methods.

Code Listing 18: `Tie::FileHandle::MultiPlex` module

```

1 package Tie::FileHandle::MultiPlex;
2 use base qw(Tie::FileHandle::Base);
3
4 sub TIEHANDLE { my $class = shift; bless [ @_ ], $class }
5
6 sub PRINT {
7     my $self = shift;
8     print $_ $_[0] for @$self;
9 }
10
11 1;

```

The program in code listing 19 combines the STDOUT, OTHER, and HANDLES filehandles in the OUT filehandle through the tie. The TIEHANDLE method in Tie::FileHandle::MultiPlex gets the typeglob references (*STDOUT) as arguments and creates the behind-the-scenes object which Perl ties to the OUT filehandle. When I print to the OUT handle, Perl knows that it is a tied filehandle so it looks in the Tie::FileHandle::MultiPlex module for the PRINT method. The PRINT method cycles through the filehandles in the behind-the-scenes object and sends the output to each one of them.

Code Listing 19: Using Tie::FileHandle::MultiPlex

```

1 use Tie::FileHandle::MultiPlex;
2
3 tie *OUT, 'Tie::FileHandle::MultiPlex', \*STDOUT, \*OTHER, \*HANDLES;
4
5 print OUT "stuff to be multiplexed";

```

4 Output Buffering

If I want to capture a program's output and only print it if some condition is met, I can run the code in an eval block and only create output if the eval did not produce any errors. PHP offers this functionality and my only major gripe with Perl is that does not. Of course the beauty of Perl is that not only can I do just about anything, but TMTOWTDI³.

In code listing 20, I create a tied filehandle module, Tie::FileHandle::Buffer, that saves the output in a string rather than outputting it. I can later retrieve the buffer when I decide I really want to output it. The behind-the-scenes object is a blessed scalar.

Code Listing 20: Tie::FileHandle::Buffer module

```

1 package Tie::FileHandle::Buffer;
2 use base qw(Tie::FileHandle::Base);
3
4 sub TIEHANDLE {
5     my( $self, $class ) = ( '', shift );
6     bless \$self, $class;
7 }
8
9 sub PRINT { ${$_[0]} .= $_[1] }
10
11 sub get_contents { ${$_[0]} }
12
13 sub clear { ${$_[0]} = '' }
14
15 1;

```

In code listing 21 I use this module to print to HANDLE, which is really a buffer. When I want to output the data, I call the get_contents method on the tied object. In this example I had a use for the tie object so I saved it, although I did not do that in the earlier example.

³There is more than one way to do it

Code Listing 21: Using Tie::FileHandle::Buffer

```

1  #!/usr/bin/perl
2  my $object = tie *HANDLE, 'Tie::FileHandle::Buffer';
3
4  print HANDLE "Hello handle!\n";
5
6  print "Object says: ", $object->get_contents;

```

I use the Tie::FileHandle::Buffer module in the Output::Buffer module in code listing 22. An Output::Buffer object wraps the Tie::FileHandle::Buffer so I can trap program output until I call either flush() or clean(), or until the object goes out of scope, which I specify with the behavior parameter I pass to the new() method. In code listing 23 I chose the FLUSH behavior, and in code listing 24, I chose the CLEAN behavior.

Code Listing 22: Output::Buffer module

```

1  package Output::Buffer;
2
3  # BEHAVIORAL CONSTANTS
4  use constant WARN => 2;
5  use constant FLUSH => 1;
6  use constant CLEAN => 0;
7
8  # EXPORT
9  our @ISA      = qw(Exporter);
10 our @EXPORT_OK = qw(WARN FLUSH CLEAN);
11 our %EXPORT_TAGS = ( constants => [@EXPORT_OK] );
12
13 # DEPENDENCIES
14 use Tie::FileHandle::Buffer;
15 use Symbol;
16 use Carp;
17
18 # Create a new output buffer
19 # Usage: my $buffer = Output::Buffer->new( behavior )
20 # where behavior is either FLUSH, CLEAN, or WARN
21 #
22 #   FLUSH - when the object loses scope, print its buffer
23 #   CLEAN - when the object loses scope, discard its buffer
24 #   WARN  - when the object loses scope, discard its buffer
25 #           but issue a warning
26 sub new {
27     my( $class, $behavior ) = @_;
28     my $fh = gensym; # create an anonymous filehandle
29     my $object = tie *{$fh}, 'Tie::FileHandle::Buffer';
30     # store our behavior, our handle, and the handle we replaced
31     bless [ $behavior, $object, select $fh ], $class;
32 }
33
34 # clean the output buffer, discarding its contents
35 sub clean { $_[0]->[1]->clear; }
36
37 # get our contents
38 sub get_contents { $_[0]->[1]->get_contents; }
39

```

```

40 # flush the output buffer, printing its contents
41 sub flush {
42     my $self = shift;
43
44     my $handle = $self->[2];
45
46     # print our contents to our parent
47     print $handle $self->get_contents;
48
49     # then discard them
50     $self->clean;
51 }
52
53 # Our scope has ended - deal with it by acting out our behavior
54 sub DESTROY {
55     my $self = shift;
56
57     if ( $self->[0] == FLUSH ) {
58         # FLUSH means flush!
59         $self->flush;
60     } else {
61         # only WARN carps - and only if there was buffered output
62         carp "Discarded output buffer contents"
63             if ( ($self->[0] == WARN) && (length($self->get_contents) != 0) );
64         # both CLEAN and WARN imply cleaning
65         $self->clean;
66     }
67
68     # return the old filehandle to domination
69     my $handle = $self->[2];
70     select $handle;
71 }
72
73 1;

```

Code listings 23 and 24 show two ways that I can use `Output::Buffer`. In listing 23 I flush the output at the end of the eval, while in listing 24 I clear `$buffer` if eval set `$@` (indicating an error). Otherwise, when I undefine `$buffer` in line 10, the `DESTROY` method outputs the contents of the buffer.

Code Listing 23: Using `Output::Buffer`, with flush

```

1 use Output::Buffer qw(:constants);
2
3 eval {
4     my $buffer = Output::Buffer->new( CLEAN );
5
6     # LOTS OF EXCEPTION-PRONE CODE HERE
7
8     # if we got here, everything worked so flush
9     $buffer->flush();
10 };

```

Code Listing 24: Using Output::Buffer, with clean

```
1 use Output::Buffer qw(:constants);
2
3 my $buffer = Output::Buffer->new( FLUSH );
4
5 eval {
6     # CODE
7 };
8
9 $buffer->clean if $0;
10 undef $buffer;
```

5 References

The perltie man page

Programming Perl, 3rd Edition, Larry Wall, et al., O'Reilly & Associates.

Many modules on CPAN implement tied filehandles, including the ones that I used in this article,

Tie::FileHandle::Base
Tie::FileHandle::Buffer

and modules from other authors, including:

Tie::Handle
Tie::Syslog
Tie::STDERR
Tie::PerFH