NAME

Attribute::Handlers - Simpler definition of attribute handlers

VERSION

This document describes version 0.93 of Attribute::Handlers, released July 20, 2011.

SYNOPSIS

```
    package MyClass;
    require 5.006;
    use Attribute::Handlers;
    no warnings 'redefine';

    sub Good : ATTR(SCALAR) {
my ($package, $symbol, $referent, $attr, $data) = @_;

# Invoked for any scalar variable with a :Good attribute,
# provided the variable was declared in MyClass (or
# a derived class) or typed to MyClass.

# Do whatever to $referent here (executed in CHECK phase).
...
    }

    sub Bad : ATTR(SCALAR) {
# Invoked for any scalar variable with a :Bad attribute,
# provided the variable was declared in MyClass (or
# a derived class) or typed to MyClass.
...
    }

    sub Good : ATTR(ARRAY) {
# Invoked for any array variable with a :Good attribute,
# provided the variable was declared in MyClass (or
# a derived class) or typed to MyClass.
...
    }

    sub Good : ATTR(HASH) {
# Invoked for any hash variable with a :Good attribute,
# provided the variable was declared in MyClass (or
# a derived class) or typed to MyClass.
...
    }

    sub Ugly : ATTR(CODE) {
# Invoked for any subroutine declared in MyClass (or a
# derived class) with an :Ugly attribute.
...
    }

    sub Omni : ATTR {
# Invoked for any scalar, array, hash, or subroutine
# with an :Omni attribute, provided the variable or
```

```
# subroutine was declared in MyClass (or a derived class)
# or the variable was typed to MyClass.
# Use ref($_[2]) to determine what kind of referent it was.
...
   }

   use Attribute::Handlers autotie => { Cycle => Tie::Cycle };

   my $next : Cycle(['A'..'Z']);
```

DESCRIPTION

This module, when inherited by a package, allows that package's class to define attribute handler subroutines for specific attributes. Variables and subroutines subsequently defined in that package, or in packages derived from that package may be given attributes with the same names as the attribute handler subroutines, which will then be called in one of the compilation phases (i.e. in a `BEGIN`, `CHECK`, `INIT`, or `END` block). (`UNITCHECK` blocks don't correspond to a global compilation phase, so they can't be specified here.)

To create a handler, define it as a subroutine with the same name as the desired attribute, and declare the subroutine itself with the attribute `:ATTR`. For example:

```
    package LoudDecl;
    use Attribute::Handlers;

    sub Loud :ATTR {
my ($package, $symbol, $referent, $attr, $data, $phase,
    $filename, $linenum) = @_;
print STDERR
    ref($referent), " ",
    *{$symbol}{NAME}, " ",
    "($referent) ", "was just declared ",
    "and ascribed the ${attr} attribute ",
    "with data ($data)\n",
    "in phase $phase\n",
    "in file $filename at line $linenum\n";
   }
```

This creates a handler for the attribute `:Loud` in the class LoudDecl. Thereafter, any subroutine declared with a `:Loud` attribute in the class LoudDecl:

```
    package LoudDecl;

    sub foo: Loud {...}
```

causes the above handler to be invoked, and passed:

[0]
: the name of the package into which it was declared;

[1]
: a reference to the symbol table entry (typeglob) containing the subroutine;

[2]
: a reference to the subroutine;

[3]

the name of the attribute;

[4]

any data associated with that attribute;

[5]

the name of the phase in which the handler is being invoked;

[6]

the filename in which the handler is being invoked;

[7]

the line number in this file.

Likewise, declaring any variables with the `:Loud` attribute within the package:

```
package LoudDecl;

my $foo :Loud;
my @foo :Loud;
my %foo :Loud;
```

will cause the handler to be called with a similar argument list (except, of course, that `$_[2]` will be a reference to the variable).

The package name argument will typically be the name of the class into which the subroutine was declared, but it may also be the name of a derived class (since handlers are inherited).

If a lexical variable is given an attribute, there is no symbol table to which it belongs, so the symbol table argument (`$_[1]`) is set to the string `'LEXICAL'` in that case. Likewise, ascribing an attribute to an anonymous subroutine results in a symbol table argument of `'ANON'`.

The data argument passes in the value (if any) associated with the attribute. For example, if `&foo` had been declared:

```
sub foo :Loud("turn it up to 11, man!") {...}
```

then a reference to an array containing the string `"turn it up to 11, man!"` would be passed as the last argument.

Attribute::Handlers makes strenuous efforts to convert the data argument (`$_[4]`) to a usable form before passing it to the handler (but see *Non-interpretive attribute handlers*). If those efforts succeed, the interpreted data is passed in an array reference; if they fail, the raw data is passed as a string. For example, all of these:

```
sub foo :Loud(till=>ears=>are=>bleeding) {...}
sub foo :Loud(qw/till ears are bleeding/) {...}
sub foo :Loud(qw/till, ears, are, bleeding/) {...}
sub foo :Loud(till,ears,are,bleeding) {...}
```

causes it to pass `['till','ears','are','bleeding']` as the handler's data argument. While:

```
sub foo :Loud(['till','ears','are','bleeding']) {...}
```

causes it to pass `[ ['till','ears','are','bleeding'] ]`; the array reference specified in the data being passed inside the standard array reference indicating successful interpretation.

However, if the data can't be parsed as valid Perl, then it is passed as an uninterpreted string. For example:

```
sub foo :Loud(my,ears,are,bleeding) {...}
sub foo :Loud(qw/my ears are bleeding) {...}
```

cause the strings `'my,ears,are,bleeding'` and `'qw/my ears are bleeding'` respectively to be passed as the data argument.

If no value is associated with the attribute, `undef` is passed.

Typed lexicals

Regardless of the package in which it is declared, if a lexical variable is ascribed an attribute, the handler that is invoked is the one belonging to the package to which it is typed. For example, the following declarations:

```
package OtherClass;

my LoudDecl $loudobj : Loud;
my LoudDecl @loudobjs : Loud;
my LoudDecl %loudobjex : Loud;
```

causes the LoudDecl::Loud handler to be invoked (even if OtherClass also defines a handler for `:Loud` attributes).

Type-specific attribute handlers

If an attribute handler is declared and the `:ATTR` specifier is given the name of a built-in type (SCALAR, ARRAY, HASH, or CODE), the handler is only applied to declarations of that type. For example, the following definition:

```
package LoudDecl;

sub RealLoud :ATTR(SCALAR) { print "Yeeeeow!" }
```

creates an attribute handler that applies only to scalars:

```
package Painful;
use base LoudDecl;

my $metal : RealLoud;           # invokes &LoudDecl::RealLoud
my @metal : RealLoud;           # error: unknown attribute
my %metal : RealLoud;           # error: unknown attribute
sub metal : RealLoud {...}      # error: unknown attribute
```

You can, of course, declare separate handlers for these types as well (but you'll need to specify `no warnings 'redefine'` to do it quietly):

```
package LoudDecl;
use Attribute::Handlers;
no warnings 'redefine';

sub RealLoud :ATTR(SCALAR) { print "Yeeeeow!" }
sub RealLoud :ATTR(ARRAY) { print "Urrrrrrrrr!" }
sub RealLoud :ATTR(HASH) { print "Arrrrrgggghhhhhh!" }
sub RealLoud :ATTR(CODE) { croak "Real loud sub torpedoed" }
```

You can also explicitly indicate that a single handler is meant to be used for all types of referents like so:

```
    package LoudDecl;
    use Attribute::Handlers;

    sub SeriousLoud :ATTR(ANY) { warn "Hearing loss imminent" }
```

(I.e. `ATTR(ANY)` is a synonym for `:ATTR`).

Non-interpretive attribute handlers

Occasionally the strenuous efforts Attribute::Handlers makes to convert the data argument (`$_[4]`) to a usable form before passing it to the handler get in the way.

You can turn off that eagerness-to-help by declaring an attribute handler with the keyword `RAWDATA`. For example:

```
    sub Raw          : ATTR(RAWDATA) {...}
    sub Nekkid       : ATTR(SCALAR,RAWDATA) {...}
    sub Au::Naturale : ATTR(RAWDATA,ANY) {...}
```

Then the handler makes absolutely no attempt to interpret the data it receives and simply passes it as a string:

```
    my $power : Raw(1..100);        # handlers receives "1..100"
```

Phase-specific attribute handlers

By default, attribute handlers are called at the end of the compilation phase (in a `CHECK` block). This seems to be optimal in most cases because most things that can be defined are defined by that point but nothing has been executed.

However, it is possible to set up attribute handlers that are called at other points in the program's compilation or execution, by explicitly stating the phase (or phases) in which you wish the attribute handler to be called. For example:

```
    sub Early    :ATTR(SCALAR,BEGIN) {...}
    sub Normal   :ATTR(SCALAR,CHECK) {...}
    sub Late     :ATTR(SCALAR,INIT) {...}
    sub Final    :ATTR(SCALAR,END) {...}
    sub Bookends :ATTR(SCALAR,BEGIN,END) {...}
```

As the last example indicates, a handler may be set up to be (re)called in two or more phases. The phase name is passed as the handler's final argument.

Note that attribute handlers that are scheduled for the `BEGIN` phase are handled as soon as the attribute is detected (i.e. before any subsequently defined `BEGIN` blocks are executed).

Attributes as tie interfaces

Attributes make an excellent and intuitive interface through which to tie variables. For example:

```
    use Attribute::Handlers;
    use Tie::Cycle;

    sub UNIVERSAL::Cycle : ATTR(SCALAR) {
 my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
 $data = [ $data ] unless ref $data eq 'ARRAY';
```

```
tie $$referent, 'Tie::Cycle', $data;
   }

   # and thereafter...

   package main;

   my $next : Cycle('A'..'Z');     # $next is now a tied variable

   while (<>) {
print $next;
   }
```

Note that, because the `Cycle` attribute receives its arguments in the `$data` variable, if the attribute is given a list of arguments, `$data` will consist of a single array reference; otherwise, it will consist of the single argument directly. Since Tie::Cycle requires its cycling values to be passed as an array reference, this means that we need to wrap non-array-reference arguments in an array constructor:

```
    $data = [ $data ] unless ref $data eq 'ARRAY';
```

Typically, however, things are the other way around: the tieable class expects its arguments as a flattened list, so the attribute looks like:

```
    sub UNIVERSAL::Cycle : ATTR(SCALAR) {
my ($package, $symbol, $referent, $attr, $data, $phase) = @_;
my @data = ref $data eq 'ARRAY' ? @$data : $data;
tie $$referent, 'Tie::Whatever', @data;
   }
```

This software pattern is so widely applicable that Attribute::Handlers provides a way to automate it: specifying `'autotie'` in the `use Attribute::Handlers` statement. So, the cycling example, could also be written:

```
    use Attribute::Handlers autotie => { Cycle => 'Tie::Cycle' };

    # and thereafter...

    package main;

    my $next : Cycle(['A'..'Z']);     # $next is now a tied variable

   while (<>) {
print $next;
   }
```

Note that we now have to pass the cycling values as an array reference, since the `autotie` mechanism passes `tie` a list of arguments as a list (as in the Tie::Whatever example), *not* as an array reference (as in the original Tie::Cycle example at the start of this section).

The argument after `'autotie'` is a reference to a hash in which each key is the name of an attribute to be created, and each value is the class to which variables ascribed that attribute should be tied.

Note that there is no longer any need to import the Tie::Cycle module -- Attribute::Handlers takes care of that automagically. You can even pass arguments to the module's `import` subroutine, by

appending them to the class name. For example:

```
    use Attribute::Handlers
  autotie => { Dir => 'Tie::Dir qw(DIR_UNLINK)' };
```

If the attribute name is unqualified, the attribute is installed in the current package. Otherwise it is installed in the qualifier's package:

```
    package Here;

    use Attribute::Handlers autotie => {
         Other::Good => Tie::SecureHash, # tie attr installed in Other::
                Bad => Tie::Taxes,      # tie attr installed in Here::
     UNIVERSAL::Ugly => Software::Patent # tie attr installed everywhere
    };
```

Autoties are most commonly used in the module to which they actually tie, and need to export their attributes to any module that calls them. To facilitate this, Attribute::Handlers recognizes a special "pseudo-class" -- `__CALLER__`, which may be specified as the qualifier of an attribute:

```
    package Tie::Me::Kangaroo:Down::Sport;

    use Attribute::Handlers autotie =>
  { '__CALLER__::Roo' => __PACKAGE__ };
```

This causes Attribute::Handlers to define the `Roo` attribute in the package that imports the Tie::Me::Kangaroo:Down::Sport module.

Note that it is important to quote the __CALLER__::Roo identifier because a bug in perl 5.8 will refuse to parse it and cause an unknown error.

Passing the tied object to tie

Occasionally it is important to pass a reference to the object being tied to the TIESCALAR, TIEHASH, etc. that ties it.

The `autotie` mechanism supports this too. The following code:

```
    use Attribute::Handlers autotieref => { Selfish => Tie::Selfish };
    my $var : Selfish(@args);
```

has the same effect as:

```
    tie my $var, 'Tie::Selfish', @args;
```

But when `"autotieref"` is used instead of `"autotie"`:

```
    use Attribute::Handlers autotieref => { Selfish => Tie::Selfish };
    my $var : Selfish(@args);
```

the effect is to pass the `tie` call an extra reference to the variable being tied:

```
    tie my $var, 'Tie::Selfish', \$var, @args;
```

EXAMPLES

If the class shown in *SYNOPSIS* were placed in the MyClass.pm module, then the following code:

```
    package main;
    use MyClass;

    my MyClass $slr :Good :Bad(1**1-1) :Omni(-vorous);

    package SomeOtherClass;
    use base MyClass;

    sub tent { 'acle' }

    sub fn :Ugly(sister) :Omni('po',tent()) {...}
    my @arr :Good :Omni(s/cie/nt/);
    my %hsh :Good(q/bye/) :Omni(q/bus/);
```

would cause the following handlers to be invoked:

```
    # my MyClass $slr :Good :Bad(1**1-1) :Omni(-vorous);

    MyClass::Good:ATTR(SCALAR)( 'MyClass',          # class
                                'LEXICAL',          # no typeglob
                                \$slr,              # referent
                                'Good',             # attr name
                                undef               # no attr data
                                'CHECK',            # compiler phase
                              );

    MyClass::Bad:ATTR(SCALAR)( 'MyClass',           # class
                               'LEXICAL',           # no typeglob
                               \$slr,               # referent
                               'Bad',               # attr name
                               0                    # eval'd attr data
                               'CHECK',             # compiler phase
                             );

    MyClass::Omni:ATTR(SCALAR)( 'MyClass',          # class
                                'LEXICAL',          # no typeglob
                                \$slr,              # referent
                                'Omni',             # attr name
                                '-vorous'           # eval'd attr data
                                'CHECK',            # compiler phase
                              );

    # sub fn :Ugly(sister) :Omni('po',tent()) {...}

    MyClass::UGLY:ATTR(CODE)( 'SomeOtherClass',     # class
                              \*SomeOtherClass::fn, # typeglob
                              \&SomeOtherClass::fn, # referent
                              'Ugly',               # attr name
                              'sister'              # eval'd attr data
                              'CHECK',              # compiler phase
                            );

    MyClass::Omni:ATTR(CODE)( 'SomeOtherClass',     # class
```

```
                                \*SomeOtherClass::fn, # typeglob
                                \&SomeOtherClass::fn, # referent
                                'Omni',               # attr name
                                ['po','acle']         # eval'd attr data
                                'CHECK',              # compiler phase
                               );

    # my @arr :Good :Omni(s/cie/nt/);

    MyClass::Good:ATTR(ARRAY)( 'SomeOtherClass',    # class
                               'LEXICAL',           # no typeglob
                               \@arr,               # referent
                               'Good',              # attr name
                               undef                # no attr data
                               'CHECK',             # compiler phase
                              );

    MyClass::Omni:ATTR(ARRAY)( 'SomeOtherClass',    # class
                               'LEXICAL',           # no typeglob
                               \@arr,               # referent
                               'Omni',              # attr name
                               ""                   # eval'd attr data
                               'CHECK',             # compiler phase
                              );

    # my %hsh :Good(q/bye) :Omni(q/bus/);

    MyClass::Good:ATTR(HASH)( 'SomeOtherClass',     # class
                              'LEXICAL',            # no typeglob
                              \%hsh,                # referent
                              'Good',               # attr name
                              'q/bye'               # raw attr data
                              'CHECK',              # compiler phase
                             );

    MyClass::Omni:ATTR(HASH)( 'SomeOtherClass',     # class
                              'LEXICAL',            # no typeglob
                              \%hsh,                # referent
                              'Omni',               # attr name
                              'bus'                 # eval'd attr data
                              'CHECK',              # compiler phase
                             );
```

Installing handlers into UNIVERSAL, makes them...err..universal. For example:

```
    package Descriptions;
    use Attribute::Handlers;

    my %name;
    sub name { return $name{$_[2]}||*{$_[1]}{NAME} }

    sub UNIVERSAL::Name :ATTR {
        $name{$_[2]} = $_[4];
```

```
    }

    sub UNIVERSAL::Purpose :ATTR {
        print STDERR "Purpose of ", &name, " is $_[4]\n";
    }

    sub UNIVERSAL::Unit :ATTR {
        print STDERR &name, " measured in $_[4]\n";
    }
```

Let's you write:

```
    use Descriptions;

    my $capacity : Name(capacity)
                 : Purpose(to store max storage capacity for files)
                 : Unit(Gb);

    package Other;

    sub foo : Purpose(to foo all data before barring it) { }

    # etc.
```

UTILITY FUNCTIONS

This module offers a single utility function, `findsym()`.

findsym

```
    my $symbol = Attribute::Handlers::findsym($package, $referent);
```

The function looks in the symbol table of `$package` for the typeglob for `$referent`, which is a reference to a variable or subroutine (SCALAR, ARRAY, HASH, or CODE). If it finds the typeglob, it returns it. Otherwise, it returns undef. Note that `findsym` memoizes the typeglobs it has previously successfully found, so subsequent calls with the same arguments should be much faster.

DIAGNOSTICS

`Bad attribute type: ATTR(%s)`

An attribute handler was specified with an `:ATTR(`*`ref_type`*`)`, but the type of referent it was defined to handle wasn't one of the five permitted: SCALAR, ARRAY, HASH, CODE, or ANY.

`Attribute handler %s doesn't handle %s attributes`

A handler for attributes of the specified name *was* defined, but not for the specified type of declaration. Typically encountered whe trying to apply a VAR attribute handler to a subroutine, or a SCALAR attribute handler to some other type of variable.

`Declaration of %s attribute in package %s may clash with future reserved word`

A handler for an attributes with an all-lowercase name was declared. An attribute with an all-lowercase name might have a meaning to Perl itself some day, even though most don't yet. Use a mixed-case attribute name, instead.

`Can't have two ATTR specifiers on one subroutine`

You just can't, okay? Instead, put all the specifications together with commas between them in a single `ATTR(`*`specification`*`)`.

`Can't autotie a %s`

You can only declare autoties for types `"SCALAR"`, `"ARRAY"`, and `"HASH"`. They're the only things (apart from typeglobs -- which are not declarable) that Perl can tie.

`Internal error: %s symbol went missing`

Something is rotten in the state of the program. An attributed subroutine ceased to exist between the point it was declared and the point at which its attribute handler(s) would have been called.

`Won't be able to apply END handler`

You have defined an END handler for an attribute that is being applied to a lexical variable. Since the variable may not be available during END this won't happen.

AUTHOR

Damian Conway (damian@conway.org). The maintainer of this module is now Rafael Garcia-Suarez (rgarciasuarez@gmail.com).

Maintainer of the CPAN release is Steffen Mueller (smueller@cpan.org). Contact him with technical difficulties with respect to the packaging of the CPAN module.

BUGS

There are undoubtedly serious bugs lurking somewhere in code this funky :-) Bug reports and other feedback are most welcome.

COPYRIGHT AND LICENSE