NAME

attributes - get/set subroutine or variable attributes

SYNOPSIS

```
sub foo : method ;
my ($x,@y,%z) : Bent = 1;
my $s = sub : method { ... };

use attributes (); # optional, to get subroutine declarations
my @attrlist = attributes::get(\&foo);

use attributes 'get'; # import the attributes::get subroutine
my @attrlist = get \&foo;
```

DESCRIPTION

Subroutine declarations and definitions may optionally have attribute lists associated with them. (Variable `my` declarations also may, but see the warning below.) Perl handles these declarations by passing some information about the call site and the thing being declared along with the attribute list to this module. In particular, the first example above is equivalent to the following:

```
use attributes __PACKAGE__, \&foo, 'method';
```

The second example in the synopsis does something equivalent to this:

```
use attributes ();
my ($x,@y,%z);
attributes::->import(__PACKAGE__, \$x, 'Bent');
attributes::->import(__PACKAGE__, \@y, 'Bent');
attributes::->import(__PACKAGE__, \%z, 'Bent');
($x,@y,%z) = 1;
```

Yes, that's a lot of expansion.

WARNING: attribute declarations for variables are still evolving. The semantics and interfaces of such declarations could change in future versions. They are present for purposes of experimentation with what the semantics ought to be. Do not rely on the current implementation of this feature.

There are only a few attributes currently handled by Perl itself (or directly by this module, depending on how you look at it.) However, package-specific attributes are allowed by an extension mechanism. (See *Package-specific Attribute Handling* below.)

The setting of subroutine attributes happens at compile time. Variable attributes in `our` declarations are also applied at compile time. However, `my` variables get their attributes applied at run-time. This means that you have to *reach* the run-time component of the `my` before those attributes will get applied. For example:

```
my $x : Bent = 42 if 0;
```

will neither assign 42 to $x *nor* will it apply the `Bent` attribute to the variable.

An attempt to set an unrecognized attribute is a fatal error. (The error is trappable, but it still stops the compilation within that `eval`.) Setting an attribute with a name that's all lowercase letters that's not a built-in attribute (such as "foo") will result in a warning with **-w** or `use warnings 'reserved'`.

What import does

In the description it is mentioned that

```
sub foo : method;
```

is equivalent to

```
use attributes __PACKAGE__, \&foo, 'method';
```

As you might know this calls the `import` function of `attributes` at compile time with these parameters: 'attributes', the caller's package name, the reference to the code and 'method'.

```
attributes->import( __PACKAGE__, \&foo, 'method' );
```

So you want to know what `import` actually does?

First of all `import` gets the type of the third parameter ('CODE' in this case). `attributes.pm` checks if there is a subroutine called `MODIFY_<reftype>_ATTRIBUTES` in the caller's namespace (here: 'main'). In this case a subroutine `MODIFY_CODE_ATTRIBUTES` is required. Then this method is called to check if you have used a "bad attribute". The subroutine call in this example would look like

```
MODIFY_CODE_ATTRIBUTES( 'main', \&foo, 'method' );
```

`MODIFY_<reftype>_ATTRIBUTES` has to return a list of all "bad attributes". If there are any bad attributes `import` croaks.

(See *Package-specific Attribute Handling* below.)

Built-in Attributes

The following are the built-in attributes for subroutines:

lvalue

Indicates that the referenced subroutine is a valid lvalue and can be assigned to. The subroutine must return a modifiable value such as a scalar variable, as described in *perlsub*.

This module allows one to set this attribute on a subroutine that is already defined. For Perl subroutines (XSUBs are fine), it may or may not do what you want, depending on the code inside the subroutine, with details subject to change in future Perl versions. You may run into problems with lvalue context not being propagated properly into the subroutine, or maybe even assertion failures. For this reason, a warning is emitted if warnings are enabled. In other words, you should only do this if you really know what you are doing. You have been warned.

method

Indicates that the referenced subroutine is a method. A subroutine so marked will not trigger the "Ambiguous call resolved as CORE::%s" warning.

prototype(..)

The "prototype" attribute is an alternate means of specifying a prototype on a sub. The desired prototype is within the parens.

The prototype from the attribute is assigned to the sub immediately after the prototype from the sub, which means that if both are declared at the same time, the traditionally defined prototype is ignored. In other words, `sub foo($$) : prototype(@) {}` is indistinguishable from `sub foo(@){}`.

If illegalproto warnings are enabled, the prototype declared inside this attribute will be sanity checked at compile time.

locked

The "locked" attribute is deprecated, and has no effect in 5.10.0 and later. It was used as part of the now-removed "Perl 5.005 threads".

const

This experimental attribute, introduced in Perl 5.22, only applies to anonymous subroutines. It causes the subroutine to be called as soon as the `sub` expression is evaluated. The return value is captured and turned into a constant subroutine.

The following are the built-in attributes for variables:

shared

Indicates that the referenced variable can be shared across different threads when used in conjunction with the *threads* and *threads::shared* modules.

unique

The "unique" attribute is deprecated, and has no effect in 5.10.0 and later. It used to indicate that a single copy of an `our` variable was to be used by all interpreters should the program happen to be running in a multi-interpreter environment.

Available Subroutines

The following subroutines are available for general use once this module has been loaded:

get

This routine expects a single parameter--a reference to a subroutine or variable. It returns a list of attributes, which may be empty. If passed invalid arguments, it uses die() (via *Carp::croak*) to raise a fatal exception. If it can find an appropriate package name for a class method lookup, it will include the results from a `FETCH_`*`type`*`_ATTRIBUTES` call in its return list, as described in *Package-specific Attribute Handling* below. Otherwise, only *built-in attributes* will be returned.

reftype

This routine expects a single parameter--a reference to a subroutine or variable. It returns the built-in type of the referenced variable, ignoring any package into which it might have been blessed. This can be useful for determining the *type* value which forms part of the method names described in *Package-specific Attribute Handling* below.

Note that these routines are *not* exported by default.

Package-specific Attribute Handling

WARNING: the mechanisms described here are still experimental. Do not rely on the current implementation. In particular, there is no provision for applying package attributes to 'cloned' copies of subroutines used as closures. (See *"Making References" in perlref* for information on closures.) Package-specific attribute handling may change incompatibly in a future release.

When an attribute list is present in a declaration, a check is made to see whether an attribute 'modify' handler is present in the appropriate package (or its @ISA inheritance tree). Similarly, when `attributes::get` is called on a valid reference, a check is made for an appropriate attribute 'fetch' handler. See *EXAMPLES* to see how the "appropriate package" determination works.

The handler names are based on the underlying type of the variable being declared or of the reference passed. Because these attributes are associated with subroutine or variable declarations, this deliberately ignores any possibility of being blessed into some package. Thus, a subroutine declaration uses "CODE" as its *type*, and even a blessed hash reference uses "HASH" as its *type*.

The class methods invoked for modifying and fetching are these:

FETCH_*type*_ATTRIBUTES

This method is called with two arguments: the relevant package name, and a reference to a

variable or subroutine for which package-defined attributes are desired. The expected return value is a list of associated attributes. This list may be empty.

MODIFY_*type*_ATTRIBUTES

This method is called with two fixed arguments, followed by the list of attributes from the relevant declaration. The two fixed arguments are the relevant package name and a reference to the declared subroutine or variable. The expected return value is a list of attributes which were not recognized by this handler. Note that this allows for a derived class to delegate a call to its base class, and then only examine the attributes which the base class didn't already handle for it.

The call to this method is currently made *during* the processing of the declaration. In particular, this means that a subroutine reference will probably be for an undefined subroutine, even if this declaration is actually part of the definition.

Calling `attributes::get()` from within the scope of a null package declaration `package ;` for an unblessed variable reference will not provide any starting package name for the 'fetch' method lookup. Thus, this circumstance will not result in a method call for package-defined attributes. A named subroutine knows to which symbol table entry it belongs (or originally belonged), and it will use the corresponding package. An anonymous subroutine knows the package name into which it was compiled (unless it was also compiled with a null package declaration), and so it will use that package name.

Syntax of Attribute Lists

An attribute list is a sequence of attribute specifications, separated by whitespace or a colon (with optional whitespace). Each attribute specification is a simple name, optionally followed by a parenthesised parameter list. If such a parameter list is present, it is scanned past as for the rules for the `q()` operator. (See *"Quote and Quote-like Operators" in perlop*.) The parameter list is passed as it was found, however, and not as per `q()`.

Some examples of syntactically valid attribute lists:

```
switch(10,foo(7,3))  :  expensive
Ugly('\(") :Bad
_5x5
lvalue method
```

Some examples of syntactically invalid attribute lists (with annotation):

```
switch(10,foo()  # ()-string not balanced
Ugly('(')   # ()-string not balanced
5x5    # "5x5" not a valid identifier
Y2::north   # "Y2::north" not a simple identifier
foo + bar   # "+" neither a colon nor whitespace
```

EXPORTS

Default exports

None.

Available exports

The routines `get` and `reftype` are exportable.

Export tags defined

The `:ALL` tag will get all of the above exports.

EXAMPLES

Here are some samples of syntactically valid declarations, with annotation as to how they resolve internally into `use attributes` invocations by perl. These examples are primarily useful to see how the "appropriate package" is found for the possible method lookups for package-defined attributes.

1. Code:

```
package Canine;
package Dog;
my Canine $spot : Watchful ;
```

   Effect:

```
use attributes ();
attributes::->import(Canine => \$spot, "Watchful");
```

2. Code:

```
package Felis;
my $cat : Nervous;
```

   Effect:

```
use attributes ();
attributes::->import(Felis => \$cat, "Nervous");
```

3. Code:

```
package X;
sub foo : lvalue ;
```

   Effect:

```
use attributes X => \&foo, "lvalue";
```

4. Code:

```
package X;
sub Y::x : lvalue { 1 }
```

   Effect:

```
use attributes Y => \&Y::x, "lvalue";
```

5. Code:

```
package X;
sub foo { 1 }

package Y;
BEGIN { *bar = \&X::foo; }

package Z;
sub Y::bar : lvalue ;
```

   Effect:

```
use attributes X => \&X::foo, "lvalue";
```

This last example is purely for purposes of completeness. You should not be trying to mess with the attributes of something in a package that's not your own.

MORE EXAMPLES

1.
```
sub MODIFY_CODE_ATTRIBUTES {
   my ($class,$code,@attrs) = @_;

   my $allowed = 'MyAttribute';
   my @bad = grep { $_ ne $allowed } @attrs;

   return @bad;
}

sub foo : MyAttribute {
   print "foo\n";
}
```

This example runs. At compile time `MODIFY_CODE_ATTRIBUTES` is called. In that subroutine, we check if any attribute is disallowed and we return a list of these "bad attributes".

As we return an empty list, everything is fine.

2.
```
sub MODIFY_CODE_ATTRIBUTES {
   my ($class,$code,@attrs) = @_;

   my $allowed = 'MyAttribute';
   my @bad = grep{ $_ ne $allowed }@attrs;

   return @bad;
}

sub foo : MyAttribute Test {
   print "foo\n";
}
```

This example is aborted at compile time as we use the attribute "Test" which isn't allowed. `MODIFY_CODE_ATTRIBUTES` returns a list that contains a single element ('Test').

SEE ALSO

"Private Variables via my()" in perlsub and *"Subroutine Attributes" in perlsub* for details on the basic declarations; *"use" in perlfunc* for details on the normal invocation mechanism.