NAME

perlref - Perl references and nested data structures

NOTE

This is complete documentation about all aspects of references. For a shorter, tutorial introduction to just the essential features, see perlreftut.

DESCRIPTION

Before release 5 of Perl it was difficult to represent complex data structures, because all references had to be symbolic — and even then it was difficult to refer to a variable instead of a symbol table entry. Perl now not only makes it easier to use symbolic references to variables, but also lets you have ‘‘hard’’ references to any piece of data or code. Any scalar may hold a hard reference. Because arrays and hashes contain scalars, you can now easily build arrays of arrays, arrays of hashes, hashes of arrays, arrays of hashes of functions, and so on.

Hard references are smart — they keep track of reference counts for you, automatically freeing the thing referred to when its reference count goes to zero. (Reference counts for values in self-referential or cyclic data structures may not go to zero without a little help; see ‘‘Circular References’’ for a detailed explanation.) If that thing happens to be an object, the object is destructed. See perlobj for more about objects. (In a sense, everything in Perl is an object, but we usually reserve the word for references to objects that have been officially ‘‘blessed’’ into a class package.)

Symbolic references are names of variables or other objects, just as a symbolic link in a Unix filesystem contains merely the name of a file. The `*glob` notation is something of a symbolic reference. (Symbolic references are sometimes called ‘‘soft references’’, but please don’t call them that; references are confusing enough without useless synonyms.)

In contrast, hard references are more like hard links in a Unix file system: They are used to access an underlying object without concern for what its (other) name is. When the word ‘‘reference’’ is used without an adjective, as in the following paragraph, it is usually talking about a hard reference.

References are easy to use in Perl. There is just one overriding principle: in general, Perl does no implicit referencing or dereferencing. When a scalar is holding a reference, it always behaves as a simple scalar. It doesn’t magically start being an array or hash or subroutine; you have to tell it explicitly to do so, by dereferencing it.

Making References

References can be created in several ways.

1. By using the backslash operator on a variable, subroutine, or value. (This works much like the & (address-of) operator in C.) This typically creates *another* reference to a variable, because there’s already a reference to the variable in the symbol table. But the symbol table reference might go away, and you’ll still have the reference that the backslash returned. Here are some examples:

   ```
   $scalarref = \$foo;
   $arrayref = \@ARGV;
   $hashref = \%ENV;
   $coderef = \&handler;
   $globref = \*foo;
   ```

   It isn’t possible to create a true reference to an IO handle (filehandle or dirhandle) using the backslash operator. The most you can get is a reference to a typeglob, which is actually a complete symbol table entry. But see the explanation of the `*foo{THING}` syntax below. However, you can still use type globs and globrefs as though they were IO handles.

2. A reference to an anonymous array can be created using square brackets:

   ```
   $arrayref = [1, 2, ['a', 'b', 'c']];
   ```

   Here we’ve created a reference to an anonymous array of three elements whose final element is itself a reference to another anonymous array of three elements. (The multidimensional syntax described later can be used to access this. For example, after the above, `$arrayref->[2][1]` would have the

value “b”.)

Taking a reference to an enumerated list is not the same as using square brackets — instead it’s the same as creating a list of references!

```
@list = (\$a, \@b, \%c);
@list = \($a, @b, %c); # same thing!
```

As a special case, `\(@foo)` returns a list of references to the contents of `@foo`, not a reference to `@foo` itself. Likewise for `%foo`, except that the key references are to copies (since the keys are just strings rather than full-fledged scalars).

3. A reference to an anonymous hash can be created using curly brackets:

```
$hashref = {
'Adam' => 'Eve',
'Clyde' => 'Bonnie',
};
```

Anonymous hash and array composers like these can be intermixed freely to produce as complicated a structure as you want. The multidimensional syntax described below works for these too. The values above are literals, but variables and expressions would work just as well, because assignment operators in Perl (even within *local()* or *my()*) are executable statements, not compile-time declarations.

Because curly brackets (braces) are used for several other things including BLOCKs, you may occasionally have to disambiguate braces at the beginning of a statement by putting a `+` or a `return` in front so that Perl realizes the opening brace isn’t starting a BLOCK. The economy and mnemonic value of using curlies is deemed worth this occasional extra hassle.

For example, if you wanted a function to make a new hash and return a reference to it, you have these options:

```
sub hashem { { @_ } } # silently wrong
sub hashem { +{ @_ } } # ok
sub hashem { return { @_ } } # ok
```

On the other hand, if you want the other meaning, you can do this:

```
sub showem { { @_ } } # ambiguous (currently ok,
# but may change)
sub showem { {; @_ } } # ok
sub showem { { return @_ } } # ok
```

The leading `+{` and `{;` always serve to disambiguate the expression to mean either the HASH reference, or the BLOCK.

4. A reference to an anonymous subroutine can be created by using `sub` without a subname:

```
$coderef = sub { print "Boink!\n" };
```

Note the semicolon. Except for the code inside not being immediately executed, a `sub {}` is not so much a declaration as it is an operator, like `do{}` or `eval{}`. (However, no matter how many times you execute that particular line (unless you’re in an `eval("...")`), `$coderef` will still have a reference to the *same* anonymous subroutine.)

Anonymous subroutines act as closures with respect to *my()* variables, that is, variables lexically visible within the current scope. Closure is a notion out of the Lisp world that says if you define an anonymous function in a particular lexical context, it pretends to run in that context even when it’s called outside the context.

In human terms, it’s a funny way of passing arguments to a subroutine when you define it as well as when you call it. It’s useful for setting up little bits of code to run later, such as callbacks. You can even do object-oriented stuff with it, though Perl already provides a different mechanism to do

that — see perlobj.

You might also think of closure as a way to write a subroutine template without using *eval()*. Here's a small example of how closures work:

```
sub newprint {
my $x = shift;
return sub { my $y = shift; print "$x, $y!\n"; };
}
$h = newprint("Howdy");
$g = newprint("Greetings");

# Time passes...

&$h("world");
&$g("earthlings");
```

This prints

```
Howdy, world!
Greetings, earthlings!
```

Note particularly that `$x` continues to refer to the value passed into *newprint() despite* "my `$x`" having gone out of scope by the time the anonymous subroutine runs. That's what a closure is all about.

This applies only to lexical variables, by the way. Dynamic variables continue to work as they have always worked. Closure is not something that most Perl programmers need trouble themselves about to begin with.

5. References are often returned by special subroutines called constructors. Perl objects are just references to a special type of object that happens to know which package it's associated with. Constructors are just special subroutines that know how to create that association. They do so by starting with an ordinary reference, and it remains an ordinary reference even while it's also being an object. Constructors are often named `new()`. You *can* call them indirectly:

   ```
   $objref = new Doggie( Tail => 'short', Ears => 'long' );
   ```

   But that can produce ambiguous syntax in certain cases, so it's often better to use the direct method invocation approach:

   ```
   $objref = Doggie->new(Tail => 'short', Ears => 'long');

   use Term::Cap;
   $terminal = Term::Cap->Tgetent( { OSPEED => 9600 });

   use Tk;
   $main = MainWindow->new();
   $menubar = $main->Frame(-relief => "raised",
   -borderwidth => 2)
   ```

6. References of the appropriate type can spring into existence if you dereference them in a context that assumes they exist. Because we haven't talked about dereferencing yet, we can't show you any examples yet.

7. A reference can be created by using a special syntax, lovingly known as the *foo{THING} syntax. *foo{THING} returns a reference to the THING slot in *foo (which is the symbol table entry which holds everything known as foo).

```
$scalarref = *foo{SCALAR};
$arrayref = *ARGV{ARRAY};
$hashref = *ENV{HASH};
$coderef = *handler{CODE};
$ioref = *STDIN{IO};
$globref = *foo{GLOB};
$formatref = *foo{FORMAT};
$globname = *foo{NAME}; # "foo"
$pkgname = *foo{PACKAGE}; # "main"
```

Most of these are self-explanatory, but `*foo{IO}` deserves special attention. It returns the IO handle, used for file handles (‘‘open’’ in perlfunc), sockets (‘‘socket’’ in perlfunc(1) and ‘‘socketpair’’ in perlfunc), and directory handles (‘‘opendir’’ in perlfunc). For compatibility with previous versions of Perl, `*foo{FILEHANDLE}` is a synonym for `*foo{IO}`, though it is discouraged, to encourage a consistent use of one name: IO. On perls between v5.8 and v5.22, it will issue a deprecation warning, but this deprecation has since been rescinded.

`*foo{THING}` returns undef if that particular THING hasn't been used yet, except in the case of scalars. `*foo{SCALAR}` returns a reference to an anonymous scalar if `$foo` hasn't been used yet. This might change in a future release.

`*foo{NAME}` and `*foo{PACKAGE}` are the exception, in that they return strings, rather than references. These return the package and name of the typeglob itself, rather than one that has been assigned to it. So, after `*foo=*Foo::bar`, `*foo` will become ‘‘*Foo::bar’’ when used as a string, but `*foo{PACKAGE}` and `*foo{NAME}` will continue to produce ‘‘main’’ and ‘‘foo’’, respectively.

`*foo{IO}` is an alternative to the `*HANDLE` mechanism given in ‘‘Typeglobs and Filehandles’’ in perldata(1) for passing filehandles into or out of subroutines, or storing into larger data structures. Its disadvantage is that it won't create a new filehandle for you. Its advantage is that you have less risk of clobbering more than you want to with a typeglob assignment. (It still conflates file and directory handles, though.) However, if you assign the incoming value to a scalar instead of a typeglob as we do in the examples below, there's no risk of that happening.

```
splutter(*STDOUT); # pass the whole glob
splutter(*STDOUT{IO}); # pass both file and dir handles

sub splutter {
my $fh = shift;
print $fh "her um well a hmmm\n";
}

$rec = get_rec(*STDIN); # pass the whole glob
$rec = get_rec(*STDIN{IO}); # pass both file and dir handles

sub get_rec {
my $fh = shift;
return scalar <$fh>;
}
```

Using References

That's it for creating references. By now you're probably dying to know how to use references to get back to your long-lost data. There are several basic methods.

1. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a simple scalar variable containing a reference of the correct type:

```
$bar = $$scalarref;
push(@$arrayref, $filename);
$$arrayref[0] = "January";
$$hashref{"KEY"} = "VALUE";
&$coderef(1,2,3);
print $globref "output\n";
```

It's important to understand that we are specifically *not* dereferencing `$arrayref[0]` or `$hashref{"KEY"}` there. The dereference of the scalar variable happens *before* it does any key lookups. Anything more complicated than a simple scalar variable must use methods 2 or 3 below. However, a ''simple scalar'' includes an identifier that itself uses method 1 recursively. Therefore, the following prints ''howdy''.

```
$refrefref = \\\"howdy";
print $$$$refrefref;
```

2. Anywhere you'd put an identifier (or chain of identifiers) as part of a variable or subroutine name, you can replace the identifier with a BLOCK returning a reference of the correct type. In other words, the previous examples could be written like this:

```
$bar = ${$scalarref};
push(@{$arrayref}, $filename);
${$arrayref}[0] = "January";
${$hashref}{"KEY"} = "VALUE";
&{$coderef}(1,2,3);
$globref->print("output\n"); # iff IO::Handle is loaded
```

Admittedly, it's a little silly to use the curlies in this case, but the BLOCK can contain any arbitrary expression, in particular, subscripted expressions:

```
&{ $dispatch{$index} }(1,2,3); # call correct routine
```

Because of being able to omit the curlies for the simple case of `$$x`, people often make the mistake of viewing the dereferencing symbols as proper operators, and wonder about their precedence. If they were, though, you could use parentheses instead of braces. That's not the case. Consider the difference below; case 0 is a short-hand version of case 1, *not* case 2:

```
$$hashref{"KEY"} = "VALUE"; # CASE 0
${$hashref}{"KEY"} = "VALUE"; # CASE 1
${$hashref{"KEY"}} = "VALUE"; # CASE 2
${$hashref->{"KEY"}} = "VALUE"; # CASE 3
```

Case 2 is also deceptive in that you're accessing a variable called `%hashref`, not dereferencing through `$hashref` to the hash it's presumably referencing. That would be case 3.

3. Subroutine calls and lookups of individual array elements arise often enough that it gets cumbersome to use method 2. As a form of syntactic sugar, the examples for method 2 may be written:

```
$arrayref->[0] = "January"; # Array element
$hashref->{"KEY"} = "VALUE"; # Hash element
$coderef->(1,2,3); # Subroutine call
```

The left side of the arrow can be any expression returning a reference, including a previous dereference. Note that `$array[$x]` is *not* the same thing as `$array->[$x]` here:

```
$array[$x]->{"foo"}->[0] = "January";
```

This is one of the cases we mentioned earlier in which references could spring into existence when in an lvalue context. Before this statement, `$array[$x]` may have been undefined. If so, it's automatically defined with a hash reference so that we can look up `{"foo"}` in it. Likewise `$array[$x]->{"foo"}` will automatically get defined with an array reference so that we can look up `[0]` in it. This process is called *autovivification*.

One more thing here. The arrow is optional *between* brackets subscripts, so you can shrink the above down to

```
$array[$x]{"foo"}[0] = "January";
```

Which, in the degenerate case of using only ordinary arrays, gives you multidimensional arrays just like C's:

```
$score[$x][$y][$z] += 42;
```

Well, okay, not entirely like C's arrays, actually. C doesn't know how to grow its arrays on demand. Perl does.

4. If a reference happens to be a reference to an object, then there are probably methods to access the things referred to, and you should probably stick to those methods unless you're in the class package that defines the object's methods. In other words, be nice, and don't violate the object's encapsulation without a very good reason. Perl does not enforce encapsulation. We are not totalitarians here. We do expect some basic civility though.

Using a string or number as a reference produces a symbolic reference, as explained above. Using a reference as a number produces an integer representing its storage location in memory. The only useful thing to be done with this is to compare two references numerically to see whether they refer to the same location.

```
if ($ref1 == $ref2) { # cheap numeric compare of references
print "refs 1 and 2 refer to the same thing\n";
}
```

Using a reference as a string produces both its referent's type, including any package blessing as described in perlobj(1), as well as the numeric address expressed in hex. The *ref()* operator returns just the type of thing the reference is pointing to, without the address. See ''ref'' in perlfunc(1) for details and examples of its use.

The *bless()* operator may be used to associate the object a reference points to with a package functioning as an object class. See perlobj.

A typeglob may be dereferenced the same way a reference can, because the dereference syntax always indicates the type of reference desired. So `${*foo}` and `${\$foo}` both indicate the same scalar variable.

Here's a trick for interpolating a subroutine call into a string:

```
print "My sub returned @{[mysub(1,2,3)]} that time.\n";
```

The way it works is that when the `@{...}` is seen in the double-quoted string, it's evaluated as a block. The block creates a reference to an anonymous array containing the results of the call to `mysub(1,2,3)`. So the whole block returns a reference to an array, which is then dereferenced by `@{...}` and stuck into the double-quoted string. This chicanery is also useful for arbitrary expressions:

```
print "That yields @{[$n + 5]} widgets\n";
```

Similarly, an expression that returns a reference to a scalar can be dereferenced via `${...}`. Thus, the above expression may be written as:

```
print "That yields ${\($n + 5)} widgets\n";
```

Circular References

It is possible to create a ''circular reference'' in Perl, which can lead to memory leaks. A circular reference occurs when two references contain a reference to each other, like this:

```
my $foo = {};
my $bar = { foo => $foo };
$foo->{bar} = $bar;
```

You can also create a circular reference with a single variable:

```
my $foo;
$foo = \$foo;
```

In this case, the reference count for the variables will never reach 0, and the references will never be garbage-collected. This can lead to memory leaks.

Because objects in Perl are implemented as references, it's possible to have circular references with objects as well. Imagine a TreeNode class where each node references its parent and child nodes. Any node with a parent will be part of a circular reference.

You can break circular references by creating a ''weak reference''. A weak reference does not increment the reference count for a variable, which means that the object can go out of scope and be destroyed. You can weaken a reference with the `weaken` function exported by the Scalar::Util module.

Here's how we can make the first example safer:

```
use Scalar::Util 'weaken';

my $foo = {};
my $bar = { foo => $foo };
$foo->{bar} = $bar;

weaken $foo->{bar};
```

The reference from `$foo` to `$bar` has been weakened. When the `$bar` variable goes out of scope, it will be garbage-collected. The next time you look at the value of the `$foo->{bar}` key, it will be `undef`.

This action at a distance can be confusing, so you should be careful with your use of weaken. You should weaken the reference in the variable that will go out of scope *first*. That way, the longer-lived variable will contain the expected reference until it goes out of scope.

Symbolic references

We said that references spring into existence as necessary if they are undefined, but we didn't say what happens if a value used as a reference is already defined, but *isn't* a hard reference. If you use it as a reference, it'll be treated as a symbolic reference. That is, the value of the scalar is taken to be the *name* of a variable, rather than a direct link to a (possibly) anonymous value.

People frequently expect it to work like this. So it does.

```
$name = "foo";
$$name = 1; # Sets $foo
${$name} = 2; # Sets $foo
${$name x 2} = 3; # Sets $foofoo
$name->[0] = 4; # Sets $foo[0]
@$name = (); # Clears @foo
&$name(); # Calls &foo()
$pack = "THAT";
${"${pack}::$name"} = 5; # Sets $THAT::foo without eval
```

This is powerful, and slightly dangerous, in that it's possible to intend (with the utmost sincerity) to use a hard reference, and accidentally use a symbolic reference instead. To protect against that, you can say

```
use strict 'refs';
```

and then only hard references will be allowed for the rest of the enclosing block. An inner block may countermand that with

```
no strict 'refs';
```

Only package variables (globals, even if localized) are visible to symbolic references. Lexical variables (declared with *my()*) aren't in a symbol table, and thus are invisible to this mechanism. For example:

```
local $value = 10;
$ref = "value";
{
my $value = 20;
print $$ref;
}
```

This will still print 10, not 20. Remember that *local()* affects package variables, which are all ‘‘global’’ to the package.

Not-so-symbolic references

Brackets around a symbolic reference can simply serve to isolate an identifier or variable name from the rest of an expression, just as they always have within a string. For example,

```
$push = "pop on ";
print "${push}over";
```

has always meant to print ‘‘pop on over’’, even though push is a reserved word. This is generalized to work the same without the enclosing double quotes, so that

```
print ${push} . "over";
```

and even

```
print ${ push } . "over";
```

will have the same effect. This construct is *not* considered to be a symbolic reference when you’re using strict refs:

```
use strict 'refs';
${ bareword }; # Okay, means $bareword.
${ "bareword" }; # Error, symbolic reference.
```

Similarly, because of all the subscripting that is done using single words, the same rule applies to any bareword that is used for subscripting a hash. So now, instead of writing

```
$array{ "aaa" }{ "bbb" }{ "ccc" }
```

you can write just

```
$array{ aaa }{ bbb }{ ccc }
```

and not worry about whether the subscripts are reserved words. In the rare event that you do wish to do something like

```
$array{ shift }
```

you can force interpretation as a reserved word by adding anything that makes it more than a bareword:

```
$array{ shift() }
$array{ +shift }
$array{ shift @_ }
```

The `use warnings` pragma or the **-w** switch will warn you if it interprets a reserved word as a string. But it will no longer warn you about using lowercase words, because the string is effectively quoted.

Pseudo-hashes: Using an array as a hash

Pseudo-hashes have been removed from Perl. The ’fields’ pragma remains available.

Function Templates

As explained above, an anonymous function with access to the lexical variables visible when that function was compiled, creates a closure. It retains access to those variables even though it doesn’t get run until later, such as in a signal handler or a Tk callback.

Using a closure as a function template allows us to generate many functions that act similarly. Suppose you wanted functions named after the colors that generated HTML font changes for the various colors:

```
print "Be ", red("careful"), "with that ", green("light");
```

The *red()* and *green()* functions would be similar. To create these, we'll assign a closure to a typeglob of the name of the function we're trying to build.

```
@colors = qw(red blue green yellow orange purple violet);
for my $name (@colors) {
no strict 'refs'; # allow symbol table manipulation
*$name = *{uc $name} = sub { "<FONT COLOR='$name'>@_</FONT>" };
}
```

Now all those different functions appear to exist independently. You can call *red()*, *RED()*, *blue()*, *BLUE()*, *green()*, etc. This technique saves on both compile time and memory use, and is less error-prone as well, since syntax checks happen at compile time. It's critical that any variables in the anonymous subroutine be lexicals in order to create a proper closure. That's the reasons for the `my` on the loop iteration variable.

This is one of the only places where giving a prototype to a closure makes much sense. If you wanted to impose scalar context on the arguments of these functions (probably not a wise idea for this particular example), you could have written it this way instead:

```
*$name = sub ($) { "<FONT COLOR='$name'>$_[0]</FONT>" };
```

However, since prototype checking happens at compile time, the assignment above happens too late to be of much use. You could address this by putting the whole loop of assignments within a BEGIN block, forcing it to occur during compilation.

Access to lexicals that change over time — like those in the `for` loop above, basically aliases to elements from the surrounding lexical scopes — only works with anonymous subs, not with named subroutines. Generally said, named subroutines do not nest properly and should only be declared in the main package scope.

This is because named subroutines are created at compile time so their lexical variables get assigned to the parent lexicals from the first execution of the parent block. If a parent scope is entered a second time, its lexicals are created again, while the nested subs still reference the old ones.

Anonymous subroutines get to capture each time you execute the `sub` operator, as they are created on the fly. If you are accustomed to using nested subroutines in other programming languages with their own private variables, you'll have to work at it a bit in Perl. The intuitive coding of this type of thing incurs mysterious warnings about ''will not stay shared'' due to the reasons explained above. For example, this won't work:

```
sub outer {
my $x = $_[0] + 35;
sub inner { return $x * 19 } # WRONG
return $x + inner();
}
```

A work-around is the following:

```
sub outer {
my $x = $_[0] + 35;
local *inner = sub { return $x * 19 };
return $x + inner();
}
```

Now *inner()* can only be called from within *outer()*, because of the temporary assignments of the anonymous subroutine. But when it does, it has normal access to the lexical variable `$x` from the scope of *outer()* at the time outer is invoked.

This has the interesting effect of creating a function local to another function, something not normally supported in Perl.

WARNING

You may not (usefully) use a reference as the key to a hash. It will be converted into a string:

```
$x{ \$a } = $a;
```

If you try to dereference the key, it won't do a hard dereference, and you won't accomplish what you're attempting. You might want to do something more like

```
$r = \@a;
$x{ $r } = $r;
```

And then at least you can use the *values()*, which will be real refs, instead of the *keys()*, which won't.

The standard Tie::RefHash module provides a convenient workaround to this.

Postfix Dereference Syntax

Beginning in v5.20.0, a postfix syntax for using references is available. It behaves as described in ''Using References'', but instead of a prefixed sigil, a postfixed sigil-and-star is used.

For example:

```
$r = \@a;
@b = $r->@*; # equivalent to @$r or @{ $r }

$r = [ 1, [ 2, 3 ], 4 ];
$r->[1]->@*; # equivalent to @{ $r->[1] }
```

In Perl 5.20 and 5.22, this syntax must be enabled with `use feature 'postderef'`. As of Perl 5.24, no feature declarations are required to make it available.

Postfix dereference should work in all circumstances where block (circumfix) dereference worked, and should be entirely equivalent. This syntax allows dereferencing to be written and read entirely left-to-right. The following equivalencies are defined:

```
$sref->$*; # same as ${ $sref }
$aref->@*; # same as @{ $aref }
$aref->$#*; # same as $#{ $aref }
$href->%*; # same as %{ $href }
$cref->&*; # same as &{ $cref }
$gref->**; # same as *{ $gref }
```

Note especially that `$cref->&*` is *not* equivalent to `$cref->()`, and can serve different purposes.

Glob elements can be extracted through the postfix dereferencing feature:

```
$gref->*{SCALAR}; # same as *{ $gref }{SCALAR}
```

Postfix array and scalar dereferencing *can* be used in interpolating strings (double quotes or the `qq` operator), but only if the `postderef_qq` feature is enabled.

Postfix Reference Slicing

Value slices of arrays and hashes may also be taken with postfix dereferencing notation, with the following equivalencies:

```
$aref->@[ ... ]; # same as @$aref[ ... ]
$href->@{ ... }; # same as @$href{ ... }
```

Postfix key/value pair slicing, added in 5.20.0 and documented in the Key/Value Hash Slices section of perldata(1), also behaves as expected:

```
$aref->%[ ... ]; # same as %$aref[ ... ]
$href->%{ ... }; # same as %$href{ ... }
```

As with postfix array, postfix value slice dereferencing *can* be used in interpolating strings (double quotes or the `qq` operator), but only if the `postderef_qq` feature is enabled.

Assigning to References

Beginning in v5.22.0, the referencing operator can be assigned to. It performs an aliasing operation, so that the variable name referenced on the left-hand side becomes an alias for the thing referenced on the right-hand side:

```
\$a = \$b; # $a and $b now point to the same scalar
\&foo = \&bar; # foo() now means bar()
```

This syntax must be enabled with `use feature 'refaliasing'`. It is experimental, and will warn by default unless `no warnings 'experimental::refaliasing'` is in effect.

These forms may be assigned to, and cause the right-hand side to be evaluated in scalar context:

```
\$scalar
\@array
\%hash
\&sub
\my $scalar
\my @array
\my %hash
\state $scalar # or @array, etc.
\our $scalar # etc.
\local $scalar # etc.
\local our $scalar # etc.
\$some_array[$index]
\$some_hash{$key}
\local $some_array[$index]
\local $some_hash{$key}
condition ? \$this : \$that[0] # etc.
```

Slicing operations and parentheses cause the right-hand side to be evaluated in list context:

```
\@array[5..7]
(\@array[5..7])
\(@array[5..7])
\@hash{'foo','bar'}
(\@hash{'foo','bar'})
\(@hash{'foo','bar'})
(\$scalar)
\($scalar)
\(my $scalar)
\my($scalar)
(\@array)
(\%hash)
(\&sub)
\(&sub)
\($foo, @bar, %baz)
(\$foo, \@bar, \%baz)
```

Each element on the right-hand side must be a reference to a datum of the right type. Parentheses immediately surrounding an array (and possibly also `my`/`state`/`our`/`local`) will make each element of the array an alias to the corresponding scalar referenced on the right-hand side:

```
\(@a) = \(@b); # @a and @b now have the same elements
\my(@a) = \(@b); # likewise
\(my @a) = \(@b); # likewise
push @a, 3; # but now @a has an extra element that @b lacks
\(@a) = (\$a, \$b, \$c); # @a now contains $a, $b, and $c
```

Combining that form with `local` and putting parentheses immediately around a hash are forbidden

(because it is not clear what they should do):

```
\local(@array) = foo(); # WRONG
\(%hash) = bar(); # wRONG
```

Assignment to references and non-references may be combined in lists and conditional ternary expressions, as long as the values on the right-hand side are the right type for each element on the left, though this may make for obfuscated code:

```
(my $tom, \my $dick, \my @harry) = (\1, \2, [1..3]);
# $tom is now \1
# $dick is now 2 (read-only)
# @harry is (1,2,3)

my $type = ref $thingy;
($type ? $type eq 'ARRAY' ? \@foo : \$bar : $baz) = $thingy;
```

The `foreach` loop can also take a reference constructor for its loop variable, though the syntax is limited to one of the following, with an optional `my`, `state`, or `our` after the backslash:

```
\$s
\@a
\%h
\&c
```

No parentheses are permitted. This feature is particularly useful for arrays-of-arrays, or arrays-of-hashes:

```
foreach \my @a (@array_of_arrays) {
frobnicate($a[0], $a[-1]);
}

foreach \my %h (@array_of_hashes) {
$h{gelastic}++ if $h{type} eq 'funny';
}
```

CAVEAT: Aliasing does not work correctly with closures. If you try to alias lexical variables from an inner subroutine or `eval`, the aliasing will only be visible within that inner sub, and will not affect the outer subroutine where the variables are declared. This bizarre behavior is subject to change.

SEE ALSO

Besides the obvious documents, source code can be instructive. Some pathological examples of the use of references can be found in the *t/op/ref.t* regression test in the Perl source directory.

See also perldsc(1) and perllol(1) for how to use references to create complex data structures, and perlootut(1) and perlobj(1) for how to use them to create objects.