Far More Than I Ever Wanted To Know About Closures

by Johan Lodin
lodin@cpan.org

Closures are a wonderful thing. In this article I'll walk through the cases where Perl's closures behave differently from one another, and sometimes display an at first surprising behavior.

■ What's a closure?

There are different ideas of what a closure actually is. The most common definition is that it's a function that has deep bindings to its lexical context. A broader definition is that it's any data structure that has deep bindings to its lexical context.

■ A brief introduction

In Perl, the subroutine is the only data type that deeply binds its lexical context. I can define a subroutine in the same scope as a lexical variable. When the lexical variable goes out of scope, the subroutine still binds to it:

```
{
    my $x = 1;

    sub f {
        my ($y) = @_;
        return $x + $y;
    }

    print f(4); # 5
}

print f(4); # 5
```

`f()` is a named closure that deeply binds the lexical variable `$x`. When I call `f()` from anywhere in you program, it returns the value given to it plus 1. This is true even if the variable `$x` isn't in scope where I call `f()`, or even if nothing else is holding a reference to `$x`.

A typical example of this is a counter subroutine that remembers its value:

```
{
    my $c = 0;
    sub counter {
        return ++$c;
    }
}

print counter();    # 1
print counter();    # 2
print counter();    # 3
```

`counter()` is a named closure that deeply binds the lexical variable `$c`. Even though `$c` has gone out of scope, `counter()` has a reference to `$c`, so Perl does not garbage collect `$c`.

■ References

It's important to realize that a variable represents more than just a value. A variable is an instance of a data structure behind the scenes, and one of the bits of information in it is the value I get when I print the variable:

```
my $foo = 1;

my $r;
{
    my $foo = 2;
    $r = \$foo;
}

print $$r;         # 2
```

The `print` statement outputs 2 instead of 1 because it sees the inner `$foo` instead of the one in scope when I call `print`. `$foo` is just a name and the underlying data structure it represents is the thing of interest. The \ operator returns a reference to that very instance and doesn't much care what the actual name is.

When I declared `$foo` the second time with `my`, I masked the old `$foo` definition, including its value. Understanding this is crucial for understanding the more esoteric cases for Perl closures. For more about this, see the *perlref* documentation.

`my` has both compile time and runtime behavior. `my` creates a new instance for a variable when its surrounding is defined and when it is executed, except for the first time where it reuses the instance created at its surrounding's definition time. Without

Perl 5.10 Feature

The defined-or operator, `//`, tests for definedness instead of truth. It returned the first defined value:

```
my $value = $ARGV[0] // '';
```

this, the whole reference system in Perl wouldn't work.

Compare the following, in which I create a new $bar in every iteration of the for. I get what I expect, which is each number as an element of @foo:

```
my @foo;

for (1 .. 3) {
    my $bar = $_;
    push @foo, \$bar;
}

print join ' ', map { $$_ } @foo;   # 1 2 3
```

That worked as I wanted, but the next doesn't. If I declare $bar outside the loop, when I take a reference to it in each iteration of the loop, I'm actually getting the same reference over and over again:

```
my @foo;
my $bar;

for (1 .. 3) {
    $bar = $_;
    push @foo, \$bar;
}

print join ' ', map { $$_ } @foo;   # 3 3 3
```

This time I get the wrong output. Since @foo has three copies of the reference the same variable, I get the same value back for each element.

■ Anonymous closures

A closure can be anonymous just as any subroutine can. I can modify the earlier counter() subroutine so that it can count in specified steps. I'll make a generator function I'll call make_counter() that takes the step value as its argument, defaulting to 1:

```
sub make_counter {
    my ($step) = shift || 1;

    my $x = 0;

    return sub {
        $x += $step;
        return $x;
    };
}
```

Notice the close similarity between the named and the anonymous counters. The bare block from before has become the subroutine make_counter(). The assignment to $x now gets its value from my call to make_counter(), and I return the anonymous subroutine instead of using a named subroutine.

```
my $counter = make_counter(3);
print $counter->();    # 3
print $counter->();    # 6
print $counter->();    # 9
```

The anonymous subroutine in make_counter() binds (that is, references) a new instance of $x each time I call make_counter(). This means that I can get separate counters that operate independently:

```
my $counter1 = make_counter(1);
my $counter7 = make_counter(7);

print $counter1->();    #  1
print $counter7->();    #  7

print $counter1->();    #  2
print $counter7->();    # 14
```

■ Nested, named subroutines

When I have nested, named subroutines, I need to take into account the runtime effect of my and the time of definition of the subroutines. As the subroutines initially share a lexical context that can be redefined, yet the subroutines are only defined once, some peculiarities arise. Here's a subroutine outer() in which I define another subroutine, inner().

```
sub prettify {
 defined $_[0] ? $_[0] : 'undefined'
}

my $x = 'initial';

sub outer {
    my $val = $x;

    print prettify($val), "\n";

    sub inner {
        print prettify($val), "\n";
    }
}
```

With that code there are a couple of scenarios. I could call inner() before I call outer(). Since I've set $x to initial, that's what I expect inner() to output, although it doesn't do that until I call outer() first:

```
inner();           # undefined
outer();           # initial
inner();           # initial
```

The value of $val in inner() isn't defined in the first call to inner() since the assignment of $val doesn't happen until

I call outer(), at which point $val gets its value. When I call outer(), $val in inner() gets defined. So far so good. This is comparable to trying to use a variable in a BEGIN block, which runs at compile time, but not defining it until runtime:

```
  my $foo = 1;

  BEGIN {
      print $foo ? 'true' : 'false';   # false
  }

  print $foo ? 'true' : 'false';       # true
```

Now, I continue the outer()/inner() example by changing $x and calling outer() and inner() again. outer() recognizes the change in $x, but inner() doesn't:

```
  $x = 'second';

  outer();            # second
  inner();            # initial <--- Unchanged!
```

The value of $val in inner() doesn't change because inner() is defined once and only once. When I call outer() the second time, Perl creates a new $val, but only for that call to outer(). inner() is already defined and inner()'s binding to $val was done when inner() was defined.

Another illustration of this idea is using a BEGIN block inside the subroutine.

```
  my @all;

  sub outer {
      my $x;

      print "runtime:      ", \$x, "\n";

      push @all, \$x;

      BEGIN { print "compile time: ", \$x,
  "\n" }
  }

  outer() for 1 .. 4;
```

The output show that the BEGIN block connects with the first definition of $x that the runtime sees, but every other call to outer() creates another variable which each has a different reference address:

```
  compile time: SCALAR(0x18241a0)
  runtime:      SCALAR(0x18241a0)
  runtime:      SCALAR(0x274ff4)
  runtime:      SCALAR(0x275018)
  runtime:      SCALAR(0x27503c)
```

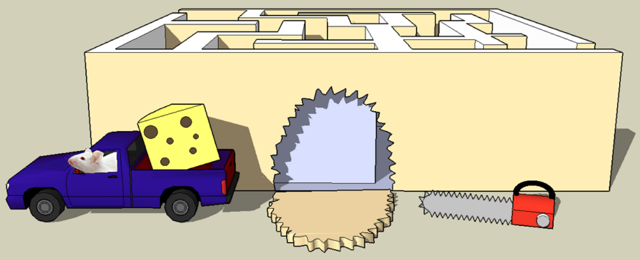
Note that I used @all just to prevent Perl from using an optimization that allows it to reuse memory slots. Since each of the references sticks around as an element of @all new instances of $x must be created. Otherwise, Perl may reuse the same memory and I wouldn't be able to see the effect with this trick.

■ Named closures in anonymous subroutines ---------

I change the example from the last section to make outer() an anonymous subroutine instead, and assign it to $outer. Everything else is the same:

```
  sub prettify {
   defined $_[0] ? $_[0] : 'undefined'
  }

  my $x = 'initial';

  my $outer = sub {
      my $val = $x;

      print prettify($val), "\n";

      sub inner {
          print prettify($val), "\n";
```

Sometimes you just need a little help to get ahead.



Perl trainings by author Peter Scott
All levels
On-site in North America

Pacific Systems Design Technologies
http://www.psdt.com/
250.474.0976
info@psdt.com

```
    }
};
```

First, I call it in the same sequence that I did before. I'll call `inner()` first, and I expect it to output `undefined` just as it did before. Then I'll dereference `$outer`, and call `inner()` again:

```
inner();            # undefined
$outer->();         # initial
inner();            # undefined
```

This time `inner()` outputs `undefined` again! The anonymous subroutine is defined at runtime (instead of compile time for named subroutines) which means that the scope of `$val` is **`redefined`** at runtime, and thus creates a new instance of `$val` the first time it's executed. This is different from the named `outer()` subroutine which reused the `$val` defined at compile time. This time, dereferencing `$outer` creates a different `$val` so I don't have a chance to assign a value to the `$val` that `inner()` can see. `&$outer` and `inner()` will therefore **`never`** share `$val`.

I can see this by using the same `BEGIN` block trick I used in the previous section:

```
my @all;

my $outer = sub {
    my $x;
    print "runtime:      ", \$x, "\n";
    push @all, \$x;

    BEGIN {
      print "compile time: ", \$x, "\n"
    }
};

$outer->() for 1 .. 4;
```

This time, none of the output lines show the same reference address:

```
compile time: SCALAR(0x1bbffbc)
runtime:      SCALAR(0x1bc006c)
runtime:      SCALAR(0x1bbfd7c)
runtime:      SCALAR(0x1bc007c)
runtime:      SCALAR(0x1bbfdec)
```

Also see our earlier articles (TPR 4.1, Winter 2007) on one of Perl's new features:

State Variables in Perl 5.10

■ Recursive, lexical anonymous closures

With "recursive lexical", I mean that the reference to an anonymous subroutine is stored in a lexical variable, and I use that variable to call the subroutine from within itself. If I have a lexical anonymous subroutine that I want to make recurse I might write it like this:

```
my $fac; # The factorial of a non-negative
integer.

$fac = sub {
    my ($n) = @_;
    return 0 if $n <= 1;
    return $n * $fac->($n - 1);
};
```

Typically I want to do this because the anonymous subroutine already is a closure and not suited for moving out to the global namespace.

This, however, creates a closure that is a circular reference. `$fac` references the closure which references `$fac`. This particular circular reference will undoubtedly be a noticeable memory leak if it's inside something that's called repeatedly. A fix for this is to not have a lexical variable to hold the subroutine for the recursive call:

```
use vars qw/ $REC /;

my $_fac = sub {
    my ($n) = @_;
    return 0 if $n <= 1;

    # $REC instead of $fac
    return $n * $REC->($n - 1);
};

my $fac = sub {
    local $REC = $_fac;
    $_fac->(@_);
};
```

This solution solves the memory leak issue and is functionally equivalent to the leaking solution except for stack traces due to the extra initial call in `&$fac`. It is, however, slightly overkill if I'm not going to use the subroutine outside its lexical scope. For that, a simple localization will fix this:

```
use vars qw/ $fac /;

local $fac = sub {
    my ($n) = @_;
    return 0 if $n <= 1;

    return $n * $fac->($n - 1);
};
```

This assumes I don't use `$fac` anywhere else in the dynamic scope (*i.e.* in the closure directly or indirectly by a subroutine call).

I've written the `Sub::Recursive` module to provide a convenient work-around using pure Perl. With `Sub::Recursive` I simply replace `sub` with `recursive` and use `$REC` for the recursive call:

```
use Sub::Recursive;

my $fac = recursive {
    my ($n) = @_;
    return 0 if $n <= 1;

    # $REC instead of $fac
    return $n * $REC->($n - 1);
};
```

`Devel::Caller` is another alternative that uses more low-level access to the interpreter to retrieve the currently executing subroutine. The relatively new XS module `Sub::Current` is yet another alternative.

■ The `state` keyword --

In Perl 5.10, there new keyword `state` lets me easily create persistent variables that are private to the subroutine. Instead of my earlier tricks that define a variable outside of the subroutine, I use `state` inside the subroutine. Perl defines the variable only once:

```
sub counter {
    state $c = 0;
    return ++$c;
}
```

I can even use `state` as part of an expression. The initialization effects only take place on the first use. Here I collapse everything in the last code example into a single expression:

```
sub counter { ++state $c }
```

Of course, I can use `state` in anonymous subroutines as well. However, there's a difference between the closure and the subroutine using `state`. The assignment using `state` is done during the first execution of the subroutine, but as I've shown, for the closures I made I can execute that subroutines before their surrounding lexical scope runs, leaving their variables uninitialized.

This becomes significant when the assignment has side effects or is time consuming. Which behavior I prefer depends on the particular problem at hand.

■ Conclusion --

The more esoteric uses of Perl's closures are as you see full of pitfalls. Hopefully I've helped you avoid them. Don't let this scare you away from closures. They're wonderful and can improve your code quality in many ways.

■ References--

The *perlref* documentation explains Perl references.

The *perlsub* documentation explains the details of lexical variables in subroutines as well as the new keyword `state`.

perlfaq7 has an answer to "What is a closure?", which has more closure examples.

perldiag on the closure warnings category. The *perldiag* lists both the short warnings and the longer explanations.

Wikipedia has general information of closures: *http://en.wikipedia.org/wiki/Closure_%28computer_science%29*

`Sub::Recursive` is available at CPAN: *http://search.cpan.org/dist/Sub-Recursive/*

■ About the author --

Johan Lodin is a student from Sweden that's been using Perl for fun and laziness over the last eight years.

Perl Trick

Sometimes you want to export to a package that isn't the package that called you. For instance, I want to make a single module that loads all the modules that I need in everywhere else. Anywhere I need all of those modules, I just load my single module:

```
#!/usr/bin/perl
# in package main:: by default
use AllMyModules;
```

The problem is that anything that call import in `AllMyModules` puts the symbols in `AllMyModules`, not in main. The trick is to not use `import`, which means avoiding `use`. Instead, use `require` and then call `export_to_level`:

```
package AllMyModules;

BEGIN {
  require Socket;
  Socket->export_to_level(
     1, '', qw(:DEFAULT) ); # up one level
  }
```

Using `1` as the first argument means I'm exporting to one level above me. This trick won't work if the module doesn't use `Exporter` because it wants to make a custom import method without the rest of the `Exporter` interface.