

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 `$x`. When I call `f()` from anywhere in your program, it returns the value given to it plus 1. This is true even if the variable `$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] // '';
```