The Scoop on Scope

Binding names to things in Perl

Names and bindings

- When you declare a variable in a program, as in

```
my $count;
```

  you are telling Perl to associate the name **`$count`** to a storage cell in memory. Henceforth, the name **`$count`** is ***bound*** to the location where its data is stored.

- You cannot write

```
my $count;
my $count;
```

  because that would create two identical names for different locations and Perl would not know which was which. ***Must names be unique in programs?***

Many moons

- In this program, there are two variables named **$user**.

```
sub DisplayGreeting
{
    my $user = $_[0];
    print "$user, Welcome to the program.\n";
}
print "Enter your name:";
chomp( my $user = <STDIN> );
DisplayGreeting($name);
```

There can be many moons ...

Separate places for each

- The first occurrence of **`$user`** is inside the block of the **`DisplayGreeting`** function. The second is outside of the block, in the main program:

```
sub DisplayGreeting
{
    my $user = $_[0];
    print "$user, Welcome to the program.\n";
}
print "Enter your name:";
chomp( my $user = <STDIN> );
DisplayGreeting($name);
```

The idea

- The same name can occur in different blocks of a program because of the concept of scope.
- Scopes are like enclosing walls around sets of names. We can declare the same name within different scopes because as soon as execution leaves the confines of one scope, it no longer "knows" the names from that scope.
- We can have scopes inside of scopes, like Russian ***matryoshka dolls***, each inside the next. Each scope can have a variable named **`$here`**.

Lexical scope

- Recall from Lesson 14 that a `my()` declaration creates a variable that can be used from that point forward in the innermost enclosing block.
- The ***lexical scope*** of a variable is the portion of the program in which that variable is accessible.
- This implies that the ***lexical scope of a variable declared with a*** `my()` ***declaration is the portion of the program from the declaration until the end of the block in which the declaration was made.***

Lexical scope

- In the block below, the lexical scope of **$user** extends from its declaration until the curly brace ending the block.

```
{
    my $user = $_[0]; # scope starts here
    print "$user, Welcome to the program.\n";
} # end of block containing declaration
# $user is out-of-scope here

# If we referred to $user here, it would not be
  the same variable.
```

Enclosing scopes

- In the code below, the scope of the first **$x** is blue; the second, pink, and the innermost, gold. Note that **$z** is visible within the innermost block.

```
my ($x, $y, $z) = (1,2,3);
{
    my $x = 4;
    {
        my $y = 6;
        print $x,$y ; # prints 4 6
    }
    print $x, $y; # prints 4 2
}
print $x, $y;  #prints 1 2
```

How lexical scopes work

- Perl maintains a ***symbol table*** for each block. The symbol table contains the names of all of the lexical variables declared within that block.
- The file containing the program is also a block, so Perl has a symbol table of the lexical variables declared at "file scope" too. The file's block is the outermost block.
- At any given time during program execution, there is a current block. This is the innermost block containing the statement being executed. If execution enters a block contained within the current block, it becomes the new current block.

Lexical scopes continued

- Each time a new block is entered, Perl pushes the symbol table for the new block on top of the table for the old block, like pushing a cafeteria tray on top of the stack of used trays.
- When a block is exited, Perl "pops" the current symbol table off of the stack of blocks that it has pushed down.
- Suppose Block A contains Block B and Block B contains Block C, and execution passed from A to B to C. Then C is "on top of" B, which is "on top of" A.
- When a lexical variable is used in a statement, Perl looks at the symbol table for the current block.

Lexical scopes continued

- When a lexical variable is used in a statement, Perl looks at the table for the current block. If it finds the variable declared in that table, it uses that variable's value. If not, it looks at the table of the next outermost block, which is "under" the current one in the stack. if it finds it there, it uses its value, otherwise it looks outward again, which means "under" that block again.
- It continues to do this until either it reaches a block whose table has the declaration, or it reaches the outermost scope and does not find it. In this case, it is a new, undeclared variable without a value, so its value is **`undef`**.

The meaning of "lexical"

- The scope created by a `my()` declaration is called a lexical scope because which declaration a variable refers to is purely determined by the program's text. The word "lexical" means ***textual***.
- Because lexical scope rules are not determined by any runtime behavior, lexical scope is also called ***static scope***.

Global scope

- The pre-defined variables, such as **`$_`** and **`@ARGV`**, are included in the main program's symbol table automatically. All variables declared in that table are called ***global variables***, because they are visible in all parts of a program.
- Variables declared using **`my()`** in file scope are not global; they are ***lexical variables with file scope***.
- You cannot create lexical versions of Perl's pre-defined variables. In other words, you cannot write **`my $_;`** anywhere in your program.

The `local()` declaration

- The `local()` declaration is used ***to hide the values of global variables temporarily***. For example,

```
@date = qw( 5 15 2009 );
local $, = "-"
print $date # this prints 5-15-2009
{
   local $, = "/";
   print $date;  # this prints 5/15/2009
}
print $date;  # this prints 5-15-2009
```

- The `local()` declaration saves the old value in the variable and restores it when the block is exited.

`local()` declarations

- The **`local()`** declaration ***cannot be used to localize a lexical variable***. For example,

```
my $name;
{
    local $name; # scope starts here
    # …. stuff here
} # end of block
```

  is illegal and is a syntax error.

- The **`local()`** declaration does not create a new copy of the variable; Perl's predefined variables have all of the "magic" they had before while they are localized.

`local()` declarations

- There is much more to **`local()`** declarations than I have said so far. Unlike lexical variables, which can only be used in the block in which they are declared, local variables exist outside of the containing block. More precisely,
- A **`local()`** declaration modifies its listed variables to be "local" to the enclosing block, and to ***any subroutine called from within that block*****.**
- This means that the localized variable is passed to called subroutines, and they can access it even though it is not in their lexical scope. The next slide illustrates.

Example

- In the example below, **foo()** calls **bar()**, implicitly handing it all localized variables, namely **$x**. **bar()** gets the value 2 for **$x**, not 1, even though the definition of **bar()** is not in the block of **foo()**.

```
use vars qw( $x ); # make $x global --
$x = 1;            # will explain soon
foo();    # bar() called from foo() uses $x=2
sub bar { print "In bar() \$x= $x\n"; }

sub foo {
          local $x= 2;
          bar();
        }
```

Warning

- You should not use `local()` definitions other than to hide the values of Perl's predefined special variables temporarily. There is absolutely no good reason to use it otherwise!
- I have showed you their behavior because you will not see it in other languages you may pick up on the way. It is not a common semantics.

User-defined global variables

- You can declare global variables in a program in one of two ways: by ***fully qualifying*** their names or by using the **`vars`** pragma.
- ***Fully qualifying*** a variable means preceding the variable name with the name of the "package" in which it is defined, followed by two colons, as in:

```
#!/usr/bin/perl -w
use strict;
$main::var = "Hello.\n";
print $main::var;
```

- **`main`** is the name of the package of your program.

The `vars` pragma

- Preceding the name of the global variable with the package name everywhere you use it is tedious; an alternative is to define it in the `vars` pragma, as follows:

```
#!/usr/bin/perl -w
use strict;
use vars ('$var', '$global');
$var = "Hello.\n";
$global = "Goodbye.\n";
print $var, $global;
```

- The `vars` pragma expects a list of variable names in single quotes.

Warning again

- It is best to avoid the use of global variables in programs. They lead to errors and difficult-to-diagnose bugs.
- Eventually you will learn about a more controlled means of limited sharing of variables among functions, via packages and namespaces.

Function declarations

- A ***function declaration*** is not the same thing as a function definition. A function declaration, sometimes called a ***prototype***, is just the word "**`sub`**" followed by the name of the function and a semicolon. The following is a function declaration:

```
sub DisplayGreeting;
```

- Notice that instead of **`{`** ***block*** **`}`** after the name, there is just a semicolon.
- Function declarations are usually placed in the beginning of the program when the definitions are at the end of the file.

Example with function declaration

- A program with a function declared in advance and defined after:

```
#!/usr/bin/perl -w
use strict;
sub DisplayGreeting; # declaration
DisplayGreeting();   # function call
# more code here
sub DisplayGreeting  # function definition
{
    print "Welcome to the program.\n";
}
```

Recursion

- Recursion is a technique for defining a function (or a set) by referring to itself. The definition must have one or more base cases that do not refer to the function.
- Here are two examples of recursive definitions:

  The natural numbers:

  0 is a natural number. If n is a natural number, so is n+1.

  Now, let ***s(x)*** be the integer that comes after ***x***.

  Addition: Let ***A(x,y)*** be called ***the sum*** of ***x*** and ***y,*** defined by

  A(x,0) = x.

  A(x,y+1) = s(A(x,y)).

Recursive functions

- The last example showed that addition of two whole numbers can be defined by defining what it means to add 0 to a number and then showing that adding anything else to a number is the same as adding the predecessor to it and adding 1. So addition is defined as long as adding 1 is defined.
- All arithmetical functions can be defined recursively. This was proved by Kurt Godel in 1931.
- Here is an obvious one:

  factorial(0) = 1

  factorial(n+1) = (n+1)* factorial(n)

Recursive Perl functions

- You can use this same idea to write functions in Perl that call themselves, provided they have an "escape clause" -- a way to stop the recursion. For example:

```
sub factorial  {
    my $x = $_[0];
    if ( 0 >= $x ) {
        return 1;
    } else {
        return $x * factorial($x-1);
    }
}
```

- You can see that `factorial(2) = 2*factorial(1) = 2*(1*factorial(0)) = 2*(1*(1)) = 2.`

Recursion is not efficient

- It is not efficient to use recursion to compute the factorial function. The following loop does the same thing:

```
sub factorial  {
    my $x = $_[0];
    my $fact = 1;
    while ( $x > 0 ) {
        $fact = $fact*$x--;
    }
    return $fact;
}
```

- Recursion is a slow means of computing, because each function call is a slow operation. Why bother then?

Why use recursion?

- Sometimes it is easier to express a solution to a problem using recursion than by any other means. When this is the case, recursion is the best choice.
- For example, many algorithms that operate on inherently recursive data structures such as directory hierarchies, are more easily expressed using recursion.
- Sometimes it is easier to see the solution to a problem using recursion.
- Sometimes recursion is used for an initial version and it is replaced by a more efficient technique later.

Rules for creating recursive functions

- A recursive function must have the following parts.
  - An argument that is used for "recursing" downwards to zero.
  - A condition to test whether the argument is zero or whatever is the lowest possible value for it. This is called the base case.
  - A return value for this base case that does not use the function itself.
  - A recursive call to the function with a value that is smaller than the current value.

Example 1

- The following function reverses a string, one character at a time.

```
sub reversestr {
    my $string  = $_[0];
    if ( 1 == length($string) ) {
        return $string;
    } else {
        my $last = chop($string);
        return $last . reversestr ($string);
    }
}
```

Example 2

- The following function computes the greatest common divisor of x and y. It could be done with iteration also.

```
sub gcd {
    my ( $x, $y ) = @_;
    ( $x, $y ) = ($y, $x ) if ( $x > $y );

    return $y if ( 0 == $x ); # gcd(0,y) is y

    return $x if ( 0 == $y ); # gcd(x,0) is x
    return gcd($x, $y % $x );
    # this algorithm was discovered by Euclid
}
```

- This has all of the required properties. See if you can identify them.

Summary

- Names in general in a program are bound to the things that they name, such as memory locations and functions.
- The same name can appear in multiple scopes.
- Variables declared with `my()` are lexical variables, which are only visible until the nearest enclosing right curly brace.
- Variables declared with `local()` have dynamic scope, and can be accessed from any subroutine called directly or indirectly from the block in which they are declared.
- Global variables are visible in all parts of the program in which they are declared.
- Recursive functions are functions that call themselves.