CS 105 Perl:
Perl subroutines and Disciplined Perl

Nathan Clement

February 3, 2014
Agenda

We will cover Perl scoping, subroutines (user-defined functions) and then we continue on to Perl’s features for enforcing better, or at least more restrictive, programming habits.

• Subroutines
  – Parameter aliasing

• Disciplined Perl
  – warnings
  – strict
A word about nomenclature

The Perl documentation distinguishes between built-in functions and user-defined functions. Built-in functions are called *functions* and are documented in the *per1func* manual page. User-defined functions are called *subroutines* and are documented in the *per1sub* manual page.
sub introduces functions

Introduce a function with the keyword sub.

sub foo {
    print "You called foo!\n";
}
Function parameters go into @_

The parameters of a function are available in @_.

```perl
sub foo {
    print "My parameters are...\n";
    foreach my $arg (@_) {
        print " $arg\n";
    }
}
&foo(qw(a b c 1 2 3));
```
Calling a Subroutine

• Use the sigil & and the (possibly empty) parenthesis to call a subroutine
  • What will the following code do?

```perl
sub log {
    print STDERR "logging", @_, "\n";
}
log(qw(a b c 1 2 3)); # Incorrect
```

• Also get in the habit of using parenthesis, even for no-parameter functions:

```perl
&foo(); # Correct
```
Use `return` to leave the function and return values to the caller.

```perl
sub foo {
    print "My parameters are...
"
    ...
    return(scalar(@_));
}
my $a = &foo(qw(a b c 1 2 3)); # $a = 6
```
No `return`?

If the function reaches the end without having found a `return`, Perl will return the last value evaluated.

```perl
sub foo {
    print "My parameters are...\n";
    ...
}
my $a = &foo(qw(a b c 1 2 3)); # ???
```
No `return`?

If the function reaches the end without having found a `return`, Perl will return the last value evaluated.

```perl
sub foo {
    print "My parameters are...\n";
    ...
}
my $a = &foo(qw(a b c 1 2 3)); # ???
```

Use `return` if you want to be sure...
Processing arguments
Processing arguments

Perl allows you to process arguments however you like.

Several techniques are common.

• Directly access elements of @_  
• Bind a list to @_  
• shift

Be aware of slurpy lists on the left hand side.
Parameters: direct access

```perl
sub foo {
    my $first = @$_[0];
    my $last = @$_[-1];
    # ...
}
```
Parameters: direct access

sub foo {  
  my $first = $$_[0];  
  my $last = $$[-1];  
  # ...  
}

Negative indices count backwards from the end of an array
Parameters: list binding

```perl
sub foo {
    my ($a, $b, $c) = @_;
    # ...
}
```
Parameters: shift
Parameters: **shift**

```perl
sub foo {
    my $a = shift;
    # ...  
    my $b = shift;
    # ...  
}

Inside a subroutine, **shift**’s default parameter is @__ rather than @ARGV.
```
Parameters: combining techniques

sub foo {
    my $first = shift;
    my (%a) = @_;  # ...
}
foo(10, %a);
Parameters: combining techniques

sub foo {
    my $first = shift;
    my (%a) = @_;  
    # ...  
}
&foo(10, %a);

or
Parameters: combining techniques

sub foo {
    my $first = shift;
    my (%a) = @_; 
    # ...
}
&foo(10, %a);

or

sub foo {
    my ($first,%a) = @_; 
}

or

sub foo {
    my ($first,%a) = @_; 
}
The slurpy argument

sub foo {
  my ($first, %a, @b) = @_; 
}

The slurpy argument

```plaintext
sub foo {
    my ($first, %a, @b) = @_;
}

Recall ...
```
The slurpy argument

sub foo {
    my ($first, %a, @b) = @_; 
}

Recall ...
• %a is a *plural*, and can store unlimited elements
The slurpy argument

sub foo {
    my ($first,%a,@b) = @_;  
}

Recall ...

• `%a` is a *plural*, and can store unlimited elements
• the argument list is *flattened*; Perl does not maintain any distinction between elements
The slurpy argument

```
sub foo {
  my ($first,%a,@b) = @_;
}
```

Recall ...

- `%a` is a *plural*, and can store unlimited elements
- the argument list is *flattened*; Perl does not maintain any distinction between elements

Therefore, all elements **but the first** (in this example) will go into `%a`; Perl cannot logically do otherwise.
@_ aliases scalar parameters

If you modify the individual elements of @_, the changes will be reflected in the original arguments.

```perl
sub foo {
    printf "%i %i %i\n", ++$_[0], --$_[1], ++$_[2];
}
$a = 1;
@b = (2);
%c = ('foo' => 3);
&foo($a, $b[0], $c{'foo'});
print "$a $b[0] $c{'foo'}\n";
```

Take-away point: I recommend **not** using this feature if you can help it.
sub foo {
    printf "%i %i %i\n", ++$_[0], --$_[1], ++$_[2];
}
@d = qw(1 2 3);
&foo(@d);
&foo(@d[2,1,0]);
print "@d\n";
Assigning to @_ prevents aliasing

```perl
sub foo {
    @_ = ();
    @$_[0] = 'foo';
}

my $a = 'bar';
&foo($a);
print "$a\n"; # prints bar
```
Assigning to @_ prevents aliasing

```perl
sub foo {
    @_ = ();
    $_[0] = 'foo';
}

my $a = 'bar';
foo($a);
print "$a\n"; # prints bar
```

Destroys aliasing relationship between formal and actual parameters
Disciplined Perl: warnings

Perl can give you many types of warnings. For example, it can warn you about:

• Wrong sigil for `%hash{$key}`
• Troublesome string-to-numeric conversions
• Using a variable only once
• Incorrect uses of barewords

... and many, many other things.
Perl warnings, the old way

Modify the hash-bang line as follows:

```bash
#!/usr/bin/perl  -w
```

This turns on warnings for everything Perl loads and runs by default.

If someone makes reference to “-w”, they mean warnings.
Perl warnings, the old way

Modify the hash-bang line as follows:

```
#!/usr/bin/perl -w
```

This turns on warnings for everything Perl loads and runs by default.

If someone makes reference to "-w", they mean warnings.

Don’t use this for new code
Perl warnings, the old way

Modify the hash-bang line as follows:

```
#!/usr/bin/perl  -w
```

This turns on warnings for everything Perl loads and runs by default.

If someone makes reference to “-w”, they mean warnings.
Perl warnings, the new way

Enable lexical warnings with `use warnings;`. This turns on warnings for the current compilation unit (file, block, etc.). You can also turn off warnings within a block/scope using `no warnings;`.

See `per1lexwarn`. 
Stricture: `use strict`

Enable strict mode with `use strict;`.

This turns on “stricture”, three categories of runtime-enforced coding standards.

- References (`refs`)
- Variables (`vars`)
- Subroutines (`subs`)

Luckily, these three categories can be enabled and disabled independently if desired.
Stricture

1. **a.** an abnormal narrowing of a bodily passage; *also:* the narrowed part
   **b.** a constriction of the breath passage in the production of a speech sound

2. something that closely restrains or limits; restriction <moral strictures>
strict refs

use strict 'refs';.

Prohibits use of symbolic references.
$ref = \$foo;  # no problem:
print $$ref;  # hard reference

$ref = "foo";  # so far so good...
print $$ref;  # symbolic ref: runtime error
strict vars

use strict 'vars';

Forces variables to be declared.

my $a = 1;  # 'my' constitutes declaration
our $b = 1; # so does 'our'

$c = 1;     # strict will complain (fatally)
strict subs

use strict 'subs';.

Restricts use of barewords that aren’t subroutines.

See the documentation for details: some special cases like => and \{hashkey\} bareword-to-string promotion is preserved.
Documentation review

Where to learn more:

• `per1sub` - scoping and subroutines
• `perllexwarn, warnings` - warnings
• `strict` - use strict