Subroutines

Subroutines

- aka: user-defined functions, methods, procdures, sub-procedures, etc etc etc
- We'll just say Subroutines.
  - "Functions" generally means built-in functions
- We'll attempt to start out most basic, and work our way up to complicated.

The Basics

```
sub myfunc{
  print "Hey, I'm in a function!\n";
}
…
myfunc( );
```

- Because function already declared, ( ) are optional (ie, can just say `myfunc;`)
- Can declare without defining:
  - `sub myfunc;`
  - Make sure you define it eventually….
- official name of subroutine is `&myfunc`
  - ampersand not normally necessary to call it

Parameters

- (aka Arguments, inputs, etc)
- Can call any subroutine with any number of parameters.
- Get passed in via local @_ variable.

```
sub myfunc{
  foreach $word (@_){
     print "$word ";
}
$foobar = 82
myfunc "hello", "world", $foobar;
```

- prints "hello world 82"

Passing current parameters

- Can call a function with the current value of `@_` as the parameter list by using `&`.
- `&myfunc;`
  - `myfunc`'s `@_` is alias to current `@_`
- same as saying `myfunc(@_);`
  - it's faster internally…

Squashing array parameters

- If arrays or hashes passed into a subroutine, they get 'squashed' into one flat array: `@_`

```
@a = (1, 2, 3); @b=(8, 9, 10);
myfunc (@a, @b);
```

- inside myfunc, @_ = (1, 2, 3, 8, 9, 10);
- Maybe this is what you want.
  - if not, need to use references…

References in Parameters

- To pass arrays (or hashes), and not squash them:

```
sub myfunc{
  ($ref1, $ref2) = @_;
  @x = @$ref1;  @y = @$ref2;
  …
}
@a = (1, 2, 3); @b = (8, 9, 10);
myfunc (\@a, \@b);
```

Return values

- In Perl, subroutines return last expression evaluated.

```
sub count {
  $sum = $_[0] + $_[1];
}
$total = count(4, 5);
#$total = 9
```

- Standard practice is to use return keyword

```
sub myfunc{
  …
  return $retval;
}
```

Return issues

- Can return values in list or scalar context.

```
sub toupper{
  @params = @_;
   foreach (@params) {tr/a-z/A-Z/;}
   return @params;
}
@uppers = toupper ($word1, $word2);
$upper = toupper($word1, $word2);
#$upper gets size of @params
```

Scalar vs List Returns

- wantarray function
  - Built-in function in Perl.
  - If subroutine called in list context, return true (1)
  - If subroutine called in scalar context, return false (“”)
  - If subroutine called in void context, return `undef`.
- Perhaps we want to return entire list, or first element if called in scalar context:

```
sub fctn{
  …
  return wantarray ? @params : $params[0];
}
```

Anonymous functions

- Can declare a function without giving it a name.
- call it by storing it’s return value in definition
  - `$subref = sub { print “Hello\n”; };`
- to call, de-reference the return value:
  - `&$subref;`
- works with parameters too..
  - `&$subref($param1, $param2);`

Scoping

- Up to now, we’ve used global variables exclusively.
- Perl has two ways of creating local variables
  - `local` and `my`
- what you may think of as local (from C/C++) is really achieved via `my`.

my

- my creates a new variable lexically scoped to inner most block
  - block may be subroutine, loop, or bare { }
- variables created with my are not accessible (or even visible) to anything outside scope.

```
sub fctn{
  my $x = shift(@_);
  …
}
print $x;  #ERROR!!!
```

lexical variables

- Variables declared with my are called “lexical variables” or “lexicals”
- Not only are they not visible outside block, mask globals with same name:

```
$foo = 10;
{
  my $foo = 3;
  print $foo;    #prints 3
}
print $foo;      #prints 10
```

Where’s the scope

- subroutines declared within a lexical’s scope have access to that lexical
  - this is one way of implementing static variables in Perl

```
{
  my $num = 20;
  sub add_to_num { $num++ }
  sub print_num { print “num = num\n”;
  }
}
add_to_num;
print_num;
print $num;      #ERROR!
```

local

- local does not create new variable
- instead, assigns temporary value to existing (global) variable
- has dynamic scope, rather than lexical
- functions called from within scope of local variable get the temporary value

```
sub fctn { print “a = $a, b = $b\n”; };
$a = 10; $b = 20;
{
  local $a = 1;
  my $b = 2;
  fctn();
}
#prints a = 1, b = 20
```

What to know about scope

- my is statically (lexically) scoped
  - look at code. whatever block encloses my is the scope of the variable
- local is dynamically scoped
  - scope is enclosing block, plus subroutines called from within that block
- Almost always want my instead of local
  - notable exception: cannot create lexical variables such as $_. Only ‘normal’, alpha-numeric variables
  - for built-in variables, localize them.

Prototypes

- Perl’s way of letting you limit how you’ll allow your subroutine to be called.
- when defining the function, give it the ‘type’ of variable you want it to take:
- sub f1 ($$) {…}
  - f1 must take two scalars
- sub f2($@) {…}
  - f2 takes a scalar, followed by a list
- sub f3(\@$) {…}
  - f3 takes an actual array, followed by a scalar

Prototype conversions

- sub fctn($$) { … }
- fctn(@foo, $bar)
- Perl converts @foo to scalar (ie, takes its size), and passes that into the function
- sub fctn2(\@$) {…}
- fctn2(@foo, $bar)
- Perl automatically creates reference to @foo to pass as first member of @_

Prototype generalities

if prototype char is:	Perl expects:
\$	actual scalar variable
\@	actual array variable
\%	actual hash variable
$	scalar
@	array – 'eats' rest of params and force list context
%	hash – 'eats' rest of params and forces hash context
*	file handle
&	subroutine (name or definition)

Getting around parameters

- If you want to ignore parameters, call subroutine with & character in front
- `sub myfunc (\$\$){ … }`
- `myfunc (@array); #ERROR!`
- `&myfunc (@array); #No error here`