## Subroutines

## Subroutines

- aka: user-defined functions, methods, procdures, sub-procedures, etc etc etc
- We'll just say Subroutines.
  - "Functions" generally means built-in functions
- **`perldoc perlsub`**

## The Basics

-
  ```
  sub myfunc {
    print "Hey, I'm in a function!\n";
  }
  #…
  myfunc();
  ```
- Because the subroutine is already declared, () are optional (ie, you can just say **`myfunc;`** )
  - If you call the function before declaring it, the () are required
- You can declare a subroutine without defining it (yet):
  - **`sub myfunc;`**
  - Make sure you define it eventually….
- actual name of the subroutine is **`&myfunc`**
  - ampersand not necessary to call it
  - in fact, has (usually undesirable) side-effects

## Parameters

- (aka Arguments, inputs, etc)
- You can call any subroutine with any number of parameters.
- The parameters get passed in via local `@_` variable.

-
  ```
  my $foobar = 82;
  myfunc('hello', 'world', $foobar);
  ```

-
  ```
  sub myfunc{
    foreach my $word (@_){
      print "$word ";
    }
    print "\n";
  }
  ```
- @_ is a normal array in every way. In this subroutine, `$_[0]` = 'hello', `$_[1]` = 'world', and `$_[2]` = 82
- prints 'hello world 82 '

## Standard Procedure

- There are two "normal" ways to obtain individual parameters:

-
  ```
  sub display {
    my ($name, $addr) = @_;
    # . . .
  }
  ```
- `shift()` in a subroutine acts on @_ with no args
  - Outside of a subroutine, acts on @ARGV

-
  ```
  sub display {
    my $name = shift;
    my $addr = shift;
    # . . .
  }
  ```
- Beware that the second method destroys @_ in the process.

## Pass by value vs Pass by reference

- *All* parameters are passed by reference.
- A direct change to an element of @_ will affect the variable passed in.
- To pass by value instead, create a copy:

-
  ```
  my ($foo, $bar) = ('old', 'old');
  change ($foo, $bar);
  ```

-
  ```
  sub change {
    my ($val1, $val2) = @_;
    $_[0] = 'new';  #changes $foo
    $val2 = 'new';  #does not change $bar
  }
  ```
- $foo Ł 'old', $bar Ł 'new'

## & side effect #1

- If you use & to call a subroutine, and don't pass any arguments, the current value of @_ will be passed automatically.
- **`&myfunc;`**
  - **`myfunc`**'s `@_` is alias to current `@_`
- same as saying **`myfunc(@_);`**, but faster internally…
- In general, don't call the subroutine with &.
  - if your subroutine checks for parameters, and you don't explicitly pass parameters, @_ will not be empty as you expect.

## Squashing array parameters

- If arrays or hashes are passed into a subroutine, they get 'squashed' into one flat array: `@_`
- 
```
my @a = (1, 2, 3);
my @b = (8, 9, 10);
myfunc (@a, @b);
```
- inside myfunc, @_ Ł (1, 2, 3, 8, 9, 10);
- Same as
```
my @c = (@a, @b);
myfunc(@c);
```
- Maybe this is what you want.
  - if not, you need to use references…

## References in Parameters

- To pass arrays (or hashes), and not squash them:
- 
```
my @a = (1, 2, 3);
my @b = (8, 9, 10);
myfunc (\@a, \@b);
```
- In subroutine, @_ contains two scalar values. Each one is a reference to an array.
- 
```
sub myfunc{
   my ($ref1, $ref2) = @_;
   my @x = @{$ref1};
   my @y = @{$ref2};
   #…
}
```

## Pass by Reference, take 2

- To not change a scalar, copy the value from `@_`
- `foo($val);`
- `sub foo { $_[0] = 'new'; }`
  - Changes `$val`
- `sub foo { my $copy = $_[0]; $copy = 'new'; }`
  - Does not change `$val`

- To not change an array, you must copy the array that is referenced!
- `bar(\@vals);`
- `sub bar { push @{$_[0]}, 'new'; }`
  - Changes `@vals`
- `sub bar { my $copy = $_[0]; push @{$copy}, 'new'; }`
  - CHANGES `@vals`!!!
  - `$_[0]` and `$copy` are two different references, but they both refer to the same array
- `sub bar { my @copy = @{$_[0]}; push @copy, 'new'; }`
  - Does not change `@vals`
  - `@copy` is a different array than `@{$_[0]}`

## Return values

- All Perl blocks return last expression evaluated.
-
```
sub count {
    # . . .
    $_[0] + $_[1];
}
```
- `$total = count(4, 5);`
  - $total Ł 9
- return keyword used for explicitness, or to leave the subroutine before its lexical end
-
```
sub myfunc{
   if (!@_){
     warn "myfunc called with no args!";
     return -1;
   }
   # . . .
}
```

## Return issues

- Can return values in list or scalar context.
-
```
sub toupper{
    my @params = @_;
    tr/a-z/A-Z/ for @params;
    return @params;
}
```
- `my @uppers = toupper $word1, $word2;`
- `my $upper  = toupper $word1, $word2;`
- $upper gets size of @params
- Why not use
  `tr/a-z/A-Z/ for @_;`
  ?

## Scalar vs List Returns

- **`wantarray`** function
  - Built-in function.
  - If subroutine called in list context, wantarray returns true
  - If subroutine called in scalar context, wantarray returns false
  - If subroutine called in void context, wantarray returns **`undef`**.
- Perhaps we want to return an entire array in list context, but the first element of the array in scalar context:
- 
```
sub fctn{
   warn "fctn() called in void context"
        unless defined wantarray;
   #. . .
   return wantarray ? @params : $params[0];
}
```

## Subroutine References

- To take a reference to a subroutine, use the & and prepend a \, like you would for any other variable:
- **`my $fooref = \&foo;`**
- You can declare a reference to an anonymous subroutine
- Store the 'return value' of **`sub`** in a scalar variable
  - **`$subref = sub { print "Hello\n"; };`**
- to call, de-reference the stored value:
  - **`&$subref;`**
  - **`$subref->(); #preferred`**
- works with parameters too..
  - **`&$subref($param1, $param2);`**
  - **`$subref->($param1, $param2);`**
- Sub refs can be stored in arrays or hashes to create "dispatch tables"
- 
```
my @dispatch = (\&foo, $subref, \&baz);
my $var = <STDIN>;
$dipatch[$var]->();
```

## Scoping

- Recall that there are two distinct scopes of variables:  Package variables and Lexical variables.
  - Package variables available anywhere, without being declared
- Perl has two ways of creating 'local' variables
  - **`local`** and **`my`**
- what you may think of as local (from C/C++) is actually achieved via **`my`**.
- **`local`** is mostly a holdover from Perl 4, which did not have lexical variables.

## 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";}
  }
  ```

-
  ```
  add_to_num; #increments $num
  print_num;  #prints current val of $num
  ```
- `print $num; #ERROR!`

## `local`

- local does not create new variable
- instead, assigns temporary value to existing package variable
- has dynamic scope
  - functions called from within scope of local variable get the temporary value

-
  ```
  our ($x, $y) = (10, 20);
  sub fctn {
    print "x = $x, y = $y\n";
  }
  ```

-
  ```
  {
     local $x = 1;
     my $y = 2;
     fctn();
  }
  ```
- in fctn(), $main::x has a temporary value created by **`local`**
  - The lexical $y is not accessible to fctn
- prints "x = 1, y = 20"

## What to know about scope

- **`my`** is statically (lexically) scoped
  - Look at the actual code. Whatever block encloses **`my`** is the scope of the variable
- **`our`** is also lexically scoped
  - allows you to use a package variable without fully qualifying
- **`local`** is dynamically scoped
  - The scope is the enclosing block, plus any subroutines called from within that block
- Almost always want **`my`** instead of **`local`**
  - notable exception: cannot create lexical variables such as **`$_`**, **`$/`**, **`$"`**, **`$,`**, etc. Only ‘normal’, alpha-numeric variables
  - for built-in variables, localize them.
- See also: "Coping With Scoping"
  - http://perl.plover.com/FAQs/Namespaces.html

## Prototypes

- Perl's way of letting you limit how you'll allow your subroutine to be called.
- when declaring the subroutine, give it the 'type' of variable you want it to take:
- **`sub f1($$) {…}`**
  - f1 must take two scalar values
- **`sub f2($@) {…}`**
  - f2 takes a scalar value, followed by a list of values
    - recall a list can contain 0, 1, or any number of values
- **`sub f3(\@$) {…}`**
  - f3 takes an actual array, followed by a scalar value
- **`sub f4() {…}`**
  - f4 takes zero arguments of any kind

## Don't use prototypes

- Prototypes are almost universally considered a mistake in the language. They should NEVER be used.
- They create a false sense of security, letting you think you don't need to check your args manually.
- They frequently do NOT work as expected.

  -
    ```
    sub fctn($@) { ... }
    fctn(@foo);
    ```
  - NO ERROR! Instead, converts @foo to scalar context, and lets the second argument be an empty list.

  -
    ```
    sub avg($$) { ... }
    my @args = ($val1, $val2);
    avg(@args);
    ```
  - ERROR! Won't let you pass the array containing two values

## Even more pointless...

- The second side-effect of calling a subroutine with the & is to disable prototype checking entirely

-
  ```
  sub foo($) { ... }
  foo(@bar, %baz);    #Error
  &foo(@bar, %baz);  #No error
  ```

## Warning your users

- If something goes wrong in a subroutine, it's often helpful to know where the subroutine was called.
- 
```
sub fctn {
  warn "fctn called in void context"
    unless defined wantarray;
}
```
  - This will only tell the user an error occurred within the subroutine, line number will be the line of the warn()
- 
```
use Carp;
sub fctn {
  carp "fctn called in void context"
    unless defined wantarray;
}
```
  - Line number reported will be line on which fctn() was called.
- `croak : carp :: die : warn`