Perl References & Objects

Brian O'Connor
UCLA
boconnor@ucla.edu

Suggested Reading

- Perl Cookbook, Chapter 11, “References and Records”
- perlref man page
- perlobj man page

Overview

- **References**
  - **Why References?**
  - **What is a Reference?**
  - **Making References**
  - **Manipulating data with References**
  - **Examining/Debugging References**
- Objects
  - Why Objects?
  - Using Objects

Why References

- How do you return two arrays from a subroutine?
  `my (@foo, @bar) = random_fxn();`
  won't work very well...
- How do you efficiently return large structures from subroutines?
- You can create an array, how do you create an array of arrays?
- How do you nest an array within a hash?

Why References

- How do you return two arrays from a subroutine?
  `my (@foo, @bar) = random_fxn();`
  won't work very well... **use references!**
- How do you efficiently return large structures from subroutines? **use references!**
- You can create an array, how do you create an array of arrays... **use references!**
- How do you nest an array within a hash... **use references!**

Why References

- How do you return two arrays from a subroutine?

```
my ($foo, $bar) = random_fxn();
```

$foo and $bar “point” to two distinct arrays produced by random_fxn

Why References

- What about returning a very large array? Is this efficient?

```
my @foo_array = random_fxn();
```

- Inside random_fxn, a large array is created. Passing this back is an expensive process

```
my $foo_array_ref = random_fxn();
```

Array of Arrays

- my data looks like:

Spot_num	Ch1-BKGD	Ch1	Ch2-BKGD	Ch2
000	0.124	43.2	0.102	80.4
001	0.113	60.7	0.091	22.6
002	0.084	112.2	0.144	35.5

- `my @spotarray = ([0.124, 43.2, 0.102, 80.4], [0.113, 60.7, 0.091, 22.6], [0.084, 112.2, 0.144, 35.5]);`

Hash of Arrays

- my data looks like:

Accession	Ch1-BKGD	Ch1	Ch2-BKGD	Ch2
AW10021	0.124	43.2	0.102	80.4
BE52002	0.113	60.7	0.091	22.6
W20209	0.084	112.2	0.144	35.5

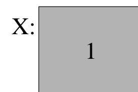
- `my %spothash = ('AW10021' => [0.124, 43.2, 0.102, 80.4], 'BE52002' => [0.113, 60.7, 0.091, 22.6], 'W20209' => [0.084, 112.2, 0.144, 35.5]);`

Complex Example

- Your nested data structures can be very complex and include many layers of hashes, arrays... even objects
- `my %complexhash = ('foo' => [('red' => [3, 6, 8], 'blue' => 65, 'green' => 38), 43.2], 'bar' => [('red' => [3, 6, 8], 'blue' => 65, 'green' => 38), 60.7], 'car' => [('red' => [3, 6, 8], 'blue' => 65, 'green' => 38), 35.5]);`

What is a Reference

- A variable... that's it... just like any other variable
- So for example: `$x` is a regular variable and it has a value of 1... this is just an ordinary variable, think of it as a box that holds a value
- `$x = 1;`

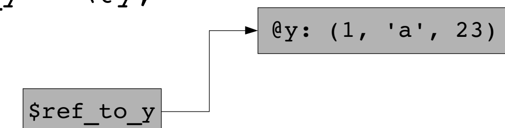


So References “Point” to a Value

- A reference is just like an ordinary variable (it holds a value) but the value is actually the address of another variable
- for example:

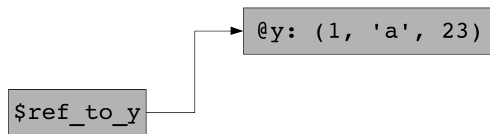
```
$ref_to_y = \@y;
```

- looks like:



So References "Point" to a Value

- printing `@y` yields `1a23`
- printing `$ref_to_y` yields `ARRAY (0x80cd6ac)`
- this is just the address of @y!



Making a Reference

- `$list_ref = \@array;`
- `$map_ref = \%hash;`
- `$c_ref = \$count;`
- `$sub_ref = \&subroutine`
- just use the "\" character... think of it as saying "points to"

Getting at the Value

- So who cares about the address... what you really want is the value that it points to
- This process is called 'de-referencing'

```
print $ref_to_y;   (gives address of y)
ARRAY (0x80cd6ac)

print @{$ref_to_y}; (gives value of y)
1a23
```

Getting at the Value

- Since a pointer "points to" anything you can dereference a hash, array, or scalar

```
@{$ref_to_array}
%{$ref_to_hash}
${$ref_to_scalar}
```

Getting at the Value

- You can also dereference into the structure:

```
print ${$ref_to_y}[0]; (first element)
or
print $ref_to_y->[0]; (my preference!)

if $ref_to_y is a ref to a hash
print $ref_to_y->{$key};
print ${$ref_to_y}{$key};
```

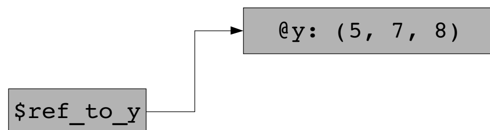
OK, That's Cool... Now What?

- Shown why you want to use references:
  - let us create complex structures
  - let us pass information easily between subroutines, objects, etc
- Shown we can easily create a reference or get to the value a reference points to.
- Can references be used to modify what it points to?

Manipulating References

- Can change their value...

```
@y = (5, 7, 8)
$ref_to_y = \@y;
```



Manipulating References

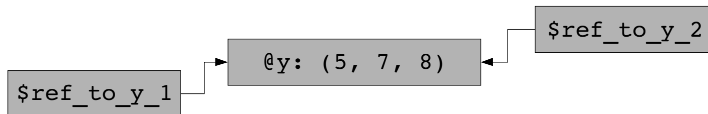
- Can change their value...

```
@y = (5, 7, 8)
$ref_to_y = \@y;
$ref_to_y->[0] = 6;

print $y[0];              6
print $ref_to_y->[0];     6 also... why?
```

Manipulating References

- Can change their value...

```
@y = (5, 7, 8)
$ref_to_y_1 = \@y;
$ref_to_y_2 = $ref_to_y_1;
$ref_to_y_1->[0] = 6;
```



Manipulating References

- Can change their value...

```
@y = (5, 7, 8)
$ref_to_y_1 = \@y;
$ref_to_y_2 = $ref_to_y_1;
$ref_to_y_1->[0] = 6;

print $ref_to_y_1->[0];        6
print $ref_to_y_2->[0];        6
```

Manipulating References

- What about copying the contents?

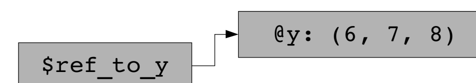
```
@y = (5, 7, 8)
$ref_to_y = \@y;
@y2 = @{$ref_to_y};
$ref_to_y->[0] = 6;

print $ref_to_y_1->[0];    6
print $y[0]                6
print $y2[0];              5
```

Manipulating References

- What about copying the contents?

```
@y = (5, 7, 8)
$ref_to_y = \@y;
@y2 = @{$ref_to_y};
$ref_to_y->[0] = 6;
```

@y2: (5, 7, 8)



Summary of Dereferencing

- **Arrays**
  - `my @new_array = @{$array_ref};`
  - `my $list_element = ${$array_ref}[1];`
  - `my $list_element = $array_ref->[1];`
- **Hashes**
  - `my %hash_copy = %{$hash_ref};`
  - `my $hash_value = ${$hash_ref}{'some key'};`
  - `my $hash_value = $hash_ref->{'some key'};`
- **Subroutines**
  - `my $result = &{$my_subroutine}($arg1, $arg2);`
  - `my $result = $my_subroutine->($arg1, $arg2);`

Anonymous Hashes/Arrays

- What if I just want the reference? You don't always want to create an array and then create a reference... solution... Anonymous Hashes/Arrays
- `$y_gene_families = ['DAZ', 'TSPY', 'RBMY'];`
- ```
  $y_gene_family_counts = {
                'DAZ'    => 4,
                'TSPY'   => 20,
                'RBMY'   => 10 };
  ```
- Then do whatever you want with the ref

How to Use these References

- ... just like you've seen before
- ```
  for (keys
  %{$y_gene_family_counts})
  {
    print "$_\n";
  }
  ```
- `my @a = @{$y_gene_families};`

How to Use these References

- I like the “arrow” notation myself...
- `$y_gene_families->[0];`
  value is 'DAZ'
- `$y_gene_family_counts->{'DAZ'};`
  value is '4'

One Small Problem...

- `$returned_reference = complex_subroutine();`
- what is `$returned_reference`?
- `print ref($y_gene_families);` ARRAY
- `print ref($y_gene_family_counts);` HASH

OK, Two Small Problems...

- `$returned_reference = some_complex_undocumented_subroutine();`
- `print ref($returned_reference);` HASH
- But this could be a nested hash of hashes, arrays, objects etc...

Two Solutions

- The debugger is awesome, use the 'x' command to show the contents of a reference
- However, I'm partial to Data::Dumper, a module designed to dump the contents of a complex structure to human-readable text

Break for Examples

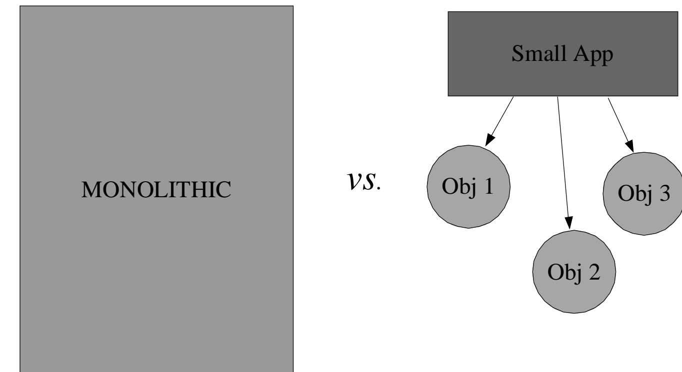
hope this works...

- hash-of-hashes example
- debugger example
- Data::Dumper example
- more from the wonderful world of hashes...

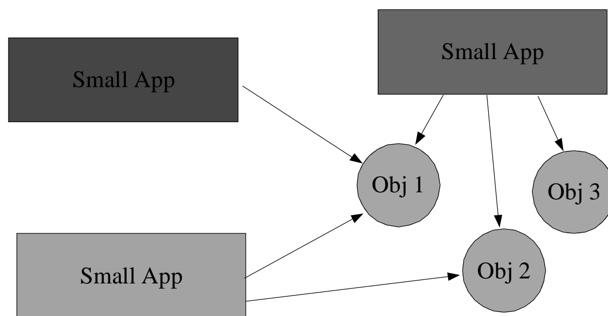
Overview

- References
  - Why References?
  - What is a Reference?
  - Making References
  - Manipulating data with References
  - Examining/Debugging References
- **Objects**
  - **Why Objects?**
  - **Using Objects**

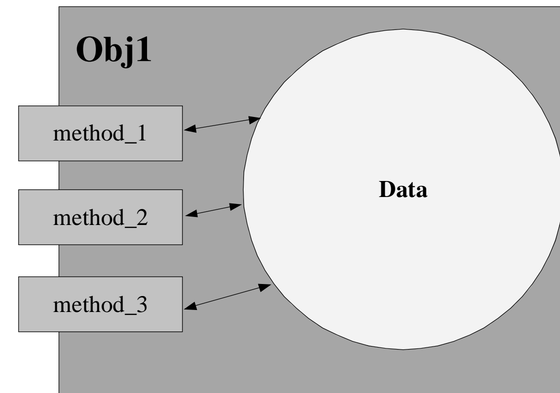
Why Objects



Why Objects



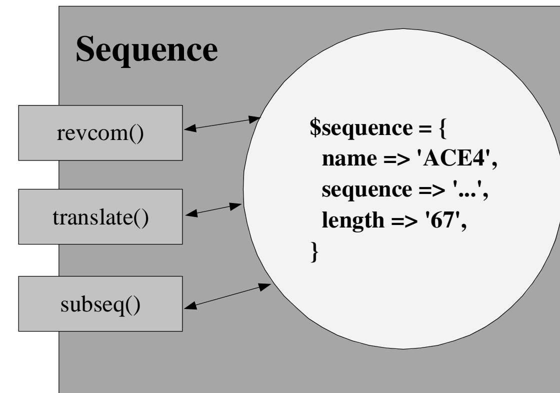
Look Inside an Object



Perl Objects

- are special references that come bundled with subroutines (called “methods”) and data
- i.e. BioPerl Sequence Object

Perl Objects



Using Objects

- Before you create/use an object you need to load them... just like loading a module!
- use “use”

```
#!/usr/bin/perl -w
use strict;
use Bio::PrimarySeq;
```

Creating an Instance

- The next step is to create a reference to a new object
- Most object oriented modules have a `new()` method

```
#!/usr/bin/perl -w
use strict;
use Bio::PrimarySeq;
my $sequence =
    Bio::PrimarySeq->new('gattaca');
```

Creating an Instance

- The new method returns an object that belong to the `Bio::PrimarySeq` class
- Another way to call `new()`

```
#!/usr/bin/perl -w
use strict;
use Bio::PrimarySeq;
my $sequence =
   new Bio::PrimarySeq('gattaca');
```

Use the Sequence Object

- A Sequence object is stored in the scalar `$sequence`
- Call methods like this:

```
$reverse_complement = $sequence->revcom();
$first_10_bases = $sequence->subseq(1,10);
$protein = $sequence->translate;
```

It's an Object... It's a Hashref...

- When it comes down to it the `$sequence` object is really just a hash ref
- you can get its keys using `keys %{$sequence}`
- you can peek inside using `$sequence->{seq}`
- This is a little different from languages such as Java where the internals of an object can be hidden

Passing Arguments to Methods

- You can pass information in... i.e. `$sequence->subseq(1,10);`
- but what about methods that take many arguments?
- use the "named parameter" style of arguments

```
my $result = $obj->method(arg1 => $value1,
                          arg2 => $value2,
                          arg3 => $value3);
```

Passing Arguments to Methods

- example with `Bio::PrimarySeq->new()`

```
#!/usr/bin/perl -w
use strict;
use Bio::PrimarySeq;
my $sequence = new Bio::PrimarySeq(
           -seq      => 'gattaca',
           -id       => 'oligo234',
           -alphabet => 'dna'
           );
```

Object Oriented Programming

- So you've seen:
  - why objects are useful
  - how to create objects
  - how to call object methods
- What about creating you're own?
  ... tomorrow

Break for Demo

- Show how to use `Bio::PrimarySeq`
- Maybe look at POD for this object