

CS306: Introduction to Perl

References

U. of Alabama at Birmingham
Dept. of Computer & Information Sciences

Slide 1

Section 10: References

Introduction

Creating References

Dereferencing

Complex Data Structures

Slide 2

Why References?

- To create more complex data types than we can with scalars, arrays and hashes alone. e.g. multidimensional arrays or multilevel hashes
- To create anonymous data – data that has no name, only a reference to its place in memory

Slide 3

The Problem

- Suppose you have:

```
@joe = (43, 'brown', 180);  
@sally = (31, 'blue', 150);  
@vitals = ('joe', 'sally');
```
- Right now, if we want to keep the same “vitals” information about two different people, we'd have to use an array (or a hash would be a little better) for each user, and then store the names of those variables in a 3rd variable

Slide 4

The Indirection Problem

- When you have a value (the string “joe”) and this value is the name of another variable whose contents you want to access, you need to perform *indirection*
- Various languages solve the indirection problem differently. For instance, C uses pointers and & (get address) and * (get data at address).
- Perl uses *references*

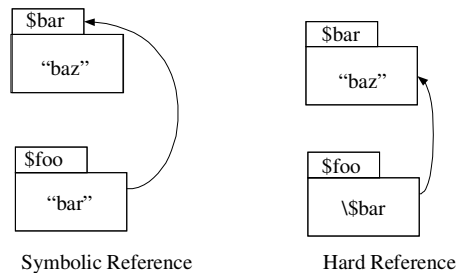
Slide 5

Two Types of Indirection

- You have a string “joe” that just happens to be the name of another variable. Perl calls this a *symbolic reference*. You can think of this as a “fake” reference.
- You have a variable that is storing a direct reference (memory address) to another variable. Perl calls this a *hard reference*. This is what we usually mean when we talk about references.

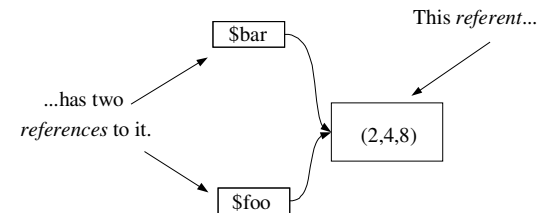
Slide 6

Two Types of References



Slide 7

References and Referents



Note that **references themselves are always scalars**, even if the referent is something other than a scalar. This turns out to be hugely important. More on that later.

Slide 8

Creating References

- Perl provides the `\` operator to create references.
- ```
$bar = \ $foo; # Reference to a scalar
$bar = \@ARGV; # Reference to an array
$bar = \%hash; # A hashref
$bar = \&foobar; # A coderef to a subroutine
$bar = \412.63; # Ref to a constant
```
- And references are just ordinary scalars, so:  

```
my $n1 = 3; my $n2 = 4; my $n5 = 5;
my @refs = (\ $n1, \ $n2, \ $n3);
```

Slide 9

## References are Scalars

- A reference is the third of the three kinds of scalar data types (along with strings and numbers).
- Thus, you can use a reference any place and manner in which you would use any other scalar
- This means that references can be array elements and hash values. If they happen to refer to other arrays - VOILA, multidimensional data!

Slide 10

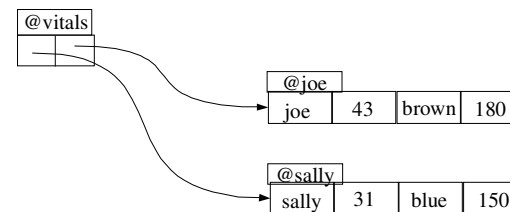
## References Allow Complex Data

- So, through the use of references you can create:
  - Arrays of Arrays
  - Hashes of Hashes
  - Arrays of Hashes
  - Hashes of Arrays
  - Arrays of Arrays of Arrays
  - etc....

Slide 11

## A Multidimensional Array

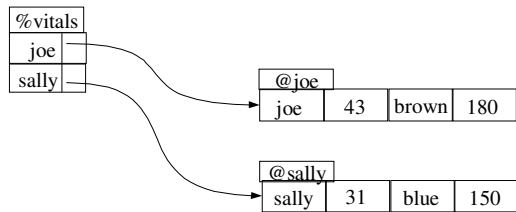
- ```
@joe   = (43, 'brown', 180);  
@sally = (31, 'blue', 150);  
@vitals = (\@joe, \@sally);
```



Slide 12

A Hash of Arrays

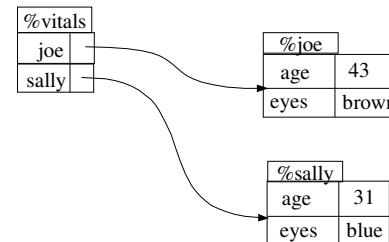
- ```
@joe = (43, 'brown', 180);
@sally = (31, 'blue', 150);
%vitals = ('joe' => \@joe, 'sally' => \@sally);
```



Slide 13

## A Hash of Hashes

- ```
%joe = (age => 43, eyes => 'brown');
%sally = (age => 31, eyes => 'blue');
%vitals = ('joe' => \%joe, 'sally' => \%sally);
```



Slide 14

Anonymous Data

- In the examples on the previous slide, we referred to data that was already stored in another variable.
- Notice that the variable names @joe, @sally, %joe and %sally aren't needed any more to get to the data that they store.

Slide 15

Anonymous Data Continued

- You can create anonymous data of arrays, hashes, etc... as well. Perl provides special syntax for this...


```
@array = (1,2,3);
$arrayref = \@array;

# compare the above to this...

$arrayref = [1,2,3]; # Now we have anon data
```
- The [] returns a reference, which is a scalar

Slide 16

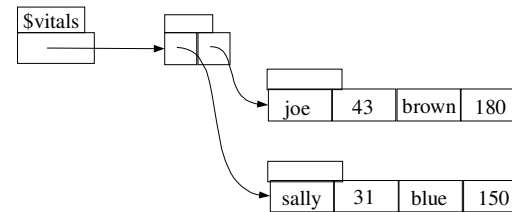
Anonymous Arrays

- Use the [] instead of () to create anonymous arrays instead of named ones.
- `my $arrayref = [1, 2, 3];`
- Remember how arrays can only store scalars?
- `my $arrayref = [1, 2, ['a', 'b', 'c'], 3];`
- Hrrmmm. What did we just do there? How many elements does that anonymous array have?

Slide 17

A Multidimensional Array

```
• $vitals = [  
  ['joe', 43, 'brown', 180],  
  ['sally', 31, 'blue', 150],  
];
```



Slide 18

Anonymous Hashes

- Use the { } to construct anonymous hashes
- ```
$hashref = { 'Mickey' => 'Donald',
 'Winnie' => 'Tigger',
 'Grinch' => anyfriends('grinch'),
 'Kyle' => ['Kenny', 'Cartman'],
};
```

Slide 19

## Anonymous Hashes, Continued

```
$vitals = {"joe" => { age => 41,
 eyes => 'brown',
 weight => 180,
 },
 "sally" => { age => 30,
 eyes => 'blue',
 weight => 150,
 },
};
```

Slide 20

## Rethinking Perl Variables

- We need to start thinking about Perl variables a little differently now.
- When you see \$foo, think of it as “the scalar value of foo”. There is an entry in memory called “foo”. The \$ says you want to look at the scalar value of it. “foo” as a name is independent of the data within.

Slide 21

## Rethinking Perl Variables

- So, if \$foo happens to be a reference (“the scalar value of foo is a reference”), you can look at what it references by putting the reference in { } and prepending the appropriate character to it.
- \${\$foo} is “the scalar value that is referred to by the scalar value of foo.” If \$foo is a *reference* to a scalar value, then \${\$foo} *is* that scalar value (referent) to which we are referring.

Slide 22

## Dereferencing

- This process is called *dereferencing*. \${\$foo} can be read as “dereferencing \$foo”.
- \${\$foo} assumes that we know:
  - \$foo is a reference
  - The thing \$foo refers to is a scalar

Slide 23

## Dereferencing Examples

- ```
$car = 'Honda';
$carref = \ $car;
$vehicle = ${$carref};
```
- ```
$arrayref = ['do','re','mi'];
push @{$arrayref}, 'fa';
foreach (@{$arrayref}) { print "$_\n"; }
```
- ```
$hashref = {mfgr => 'Dell', model => 'X200'};
@sorted = sort keys %{$hashref};
print "Model is ${$hashref}{model}\n";
```

Slide 24

The Second Way to Dereference

- Omit { } block

```
my $aref = [1,2,3];
my @sorted = sort @$aref; # {} are optional

my $href = {name => 'Fran', car => 'Honda'};
my @keys = keys %$href;
my $car = $$href{car};
```

Slide 25

The Third Way to Dereference

- Use the -> operator for arrays, hashes and subroutines

```
my $aref = ['red', 'yellow', 'blue'];
print "The sky is $aref->[2]\n";

my $hashref = {sender => 'Fran Fabrizio',
               subject => 'HW hints'};
$hashref->{subject} = 'More HW hints';
```

Slide 26

Dereferencing Comparison

```
• my $aref = ['perl','java','c'];
  $$aref[2] = 'c++';           # Method 1
  ${$aref}[2] = 'c++';        # Method 2
  $aref->[2] = 'c++';          # Method 3

• my $href = {apple => 'green', cherry => 'red'};
  print "$$href{apple}\n";     # Method 1
  print "${$href}{apple}\n";   # Method 2
  print "$href->{apple}\n";     # Method 3

• my $cref = sub {return "Hello $_\n"};
  print &$cref('Fran');        # Method 1
  print &{$cref}('Fran');       # Method 2
  print $cref->('Fran');        # Method 3
```

Slide 27

Examining References

```
• my $ref = [1,2,3];
  print $ref;                   # yields "ARRAY(0x804dc28)"
  $ref = {foo => 'bar'};
  print $ref;                   # yields "HASH(0x826d2a4)"

• You can also check if a scalar is a reference, and
  what type it is, through the ref() function

• if (ref($r) eq "HASH") {
    print "r is a reference to a hash.\n";
  }

• unless (ref($r)) {
  print "r is not a reference at all.\n";
  }
```

Slide 28

Arrays of Arrays

- my @tictactoe = (['-','-','x'],
 ['-','x','O'],
 ['o','o','-']);

- To access an element of the 2d array, you say:

```
print $tictactoe[1]->[2]; # prints '-'  
print $tictactoe[1][2];  # valid shortcut
```

Slide 29

Arrays of Arrays – More “Referency”

- my \$tictactoe = [['-','-','x'],
 ['-','x','O'],
 ['o','o','-']];

- To access an element of the 2d array, you say:

```
print $tictactoe->[1]->[2]; # prints '-'  
print $tictactoe->[1][2];  # valid shortcut  
print $tictactoe[1][2];   # BROKEN!
```

Slide 30

Array of Arrays, Continued

- Building one dynamically:

```
while (my $line = <>) {  
  my @arr = split / /, $line;  
  push @AoA, [ @arr ];  
}
```

- Shorter, same thing:

```
while (my $line = <>) {  
  push @AoA, [ split / /, $line];  
}
```

Slide 31

“Referency” AoA

- Building one dynamically:

```
my $AoA;  
while (my $line = <>) {  
  my @arr = split / /, $line;  
  push @{$AoA}, [ @arr ];  
}
```

- Shorter, same thing:

```
while (my $line = <>) {  
  push @{$AoA}, [ split / /, $line];  
}
```

Slide 32

Iterating Over Arrays of Arrays

- ```
my $tictactoe = [['-','-', 'x'],
 ['-','x','-'],
 ['o','o','-']];
```
- How would we iterate over this?
- ```
for $i (0..$#{ $tictactoe}) {  
  for $j (0..$#{ $tictactoe->[$i]}) {  
    print "$i $j has $tictactoe->[$i][$j]\n";  
  }  
}
```

Slide 33

Hash of Arrays

- ```
%HoA = (sciences => ['cs','bio','chem'],
 arts => ['music','art','theater'],
);

$HoA{engineering} = ['civil','chemical','elec'];

push @{$HoA{sciences}}, "physics";

print data
for $school (keys %HoA) {
 print "$school: @{$HoA{$school}}\n";
}
could also add inner loop to get indiv.
elements of the arrays
```

Slide 34

## Array of Hashes

- ```
@AoH = ( { mfgr => 'honda',  
          model => 'civic',  
          color => 'blue',  
        },  
        { mfgr => 'toyota',  
          model => 'camry',  
          color => 'black',  
        },  
        { mfgr => 'ford',  
          model => 'F150',  
          color => 'red',  
        },  
      );
```

Slide 35

Array of Hashes, Continued

- ```
for $href (@AoH) {
 for $feature (keys %$href) {
 print "The $feature is $href->{$feature}\n";
 }
 print "\n";
}
```

Slide 36

## Hash of Hashes

- The most flexible compound data structure

```
%tvfams = (kingofqueens => {
 husband => "doug",
 wife => "carrie",
},
 seinfeld => {
 mainchar => "jerry",
 neighbor => "kramer",
 baddancer => "elaine",
 baldguy => "george",
},
);
```

Slide 37

## Hash of Hashes, Continued

```
• $HoH{seinfeld}{mailman} = 'newman';

print everything
for $family (keys %HoH) {
 print "$family: ";
 for $role (keys %{ $HoH{$family} }) {
 print "$HoH{$family}{$role} ($role) ";
 }
 print "\n";
}
```

Slide 38

## Hash of Hashes, Continued

```
• # largest family first
use keys() in scalar context, use length of
array turned by keys()
for $family
 (sort {keys %{ $HoH{$a} } <=> keys %{ $HoH{$b} }}
 keys %HoH) {
 print "$family: ";
 for $role (keys %{ $HoH{$family} }) {
 print "$HoH{$family}{$role} ($role) ";
 }
 print "\n";
}
```

Slide 39

## Reference Counts and Cleanup

- Perl keeps count of how many things are referencing any particular piece of data.
- `$foo = 5;` # ref count for that 5 is now 1
- `$bar = \ $foo;` # Now 2 things are pointing at # memory location that has the 5
- `undef $foo;` # The memory holding the 5 lives # on, because \$bar still exists
- `undef $bar;` # The memory is freed now
- This is how Perl manages automated garbage collection

Slide 40

## Autovivifying References

- *Autovivification* - Bringing to life automatically.
- When you try to assign through an undefined reference, Perl will bring it to life for you

```
#!/usr/bin/perl
while (my $line = <>) {
 push @{$AoAref, [split / /, $line];
}
```

- Since `$AoAref` doesn't exist but is required in order to push onto this array, Perl creates it. Slide 41

## Deeper into the Perl Mind Meld

- ```
my @array;
for $i (1..10) {
    @array = somefunc();
    $AoA[$i] = @array; #WRONG
}
```
- Who can tell me why this is broken?

Slide 43

Autovivification, Continued

- Notice the behavior of this program

```
#!/usr/bin/perl
# @arr doesn't exist before this line
$arr[4][23][15][24] = 'hi';
print $arr[4][23][15][24]; # 'hi'
print $arr[4][23][15];     # ARRAY(0x804db44)
print $arr[4][23];        # ARRAY(0x804de74)
print $arr[4];            # ARRAY(0x804dd6c)
```

Slide 42

Deeper into the Perl Mind Meld

- ```
my @array;
for $i (1..10) {
 @array = somefunc();
 $AoA[$i] = @array; #WRONG
}
```
- Because you are assigning an array in scalar context, so `$AoA[$i]` is going to get the length of `@array` on each pass. You need to create references.

Slide 44

## Deeper into the Perl Mind Meld

- ```
my @array;
for $i (1..10) {
    @array = somefunc();
    $AoA[$i] = \@array;    #STILL WRONG
}
```
- Who can tell me why this is broken?

Slide 45

Deeper into the Perl Mind Meld

- ```
my @array;
for $i (1..10) {
 @array = somefunc();
 $AoA[$i] = \@array; #STILL WRONG
}
```
- @array is the same place in memory each time through the loop, so you keep grabbing a reference to the same place. Each element of \$AoA will reference the same (last) array. Slide 46

## Deeper into the Perl Mind Meld

- ```
my @array;
for $i (1..10) {
    @array = somefunc();
    $AoA[$i] = [ @array ];    #Right
}
```
- The [] create a new anonymous array into which the elements of @array are copied. We then store a reference to the anon array.

Slide 47

Deeper into the Perl Mind Meld

- ```
for $i (1..10) {
 my @array = somefunc();
 $AoA[$i] = \@array; #Also Right
}
```
- Now the lexically-scoped @array is created anew each time through the loop. (Who knows why this is better than the last option?)

Slide 48

## Deeper into the Perl Mind Meld

- for \$i (1..10) {  
  my @array = somefunc();  
  \$AoA[\$i] = \@array;   #Also Right  
}
- Because you avoid the implicit copy that's happening with the [ ]

Slide 49

## Deeper into the Perl Mind Meld

- for \$i (1..10) {  
  \$AoA[\$i] = [ somefunc() ];   #Also Right  
}
- Even better than the last slide. Why?

Slide 50

## Deeper into the Perl Mind Meld

- for \$i (1..10) {  
  \$AoA[\$i] = [ somefunc() ];   #Also Right  
}
- We've eliminated the temporary variable @array now. Now we're implicitly copying again, sure, but before we were explicitly doing a copy with the my @array = somefunc() anyhow.

Slide 51

## References Wrapup

- References enable complex data structures
- Be extra careful with references. Very easy to try to use a reference like an array/hash by accident, or vice versa. Will often get silent failures.
- Get used to reading Data::Dumper output - if you see [ ] or { } it's a reference, ( ) it's an array or hash.
- The End. Questions?

Slide 52