CS306: Introduction to Perl

Section #10: References

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

Section 10: References

Introduction
Creating References
Dereferencing
Complex Data Structures

A Quick Note About my

For clarity during lecture, I may omit the my keyword before variable declarations.

However, in your programs you should always use my and avoid global variables.

The Problem We Want To Solve

- Suppose you have:

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

- Given only the string "joe", we want to change the contents of @joe to set Joe's eyes from brown to hazel.

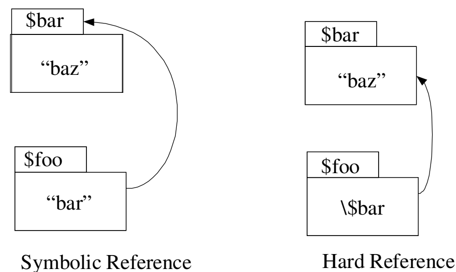
The Indirection Problem

- 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*

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.

Two Types of References

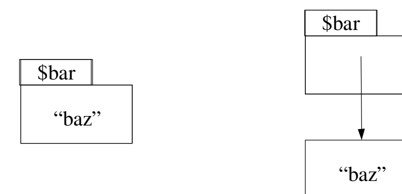


Symbolic Reference

Hard Reference

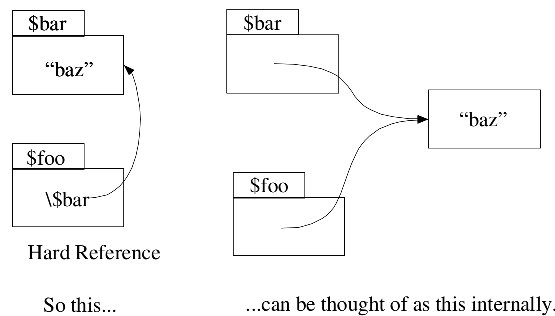
A Perl Secret

All Perl variables are really just hard references...



This... ...can be thought of as ... ...this.

A Perl Secret, Continued



So this... ...can be thought of as this internally.

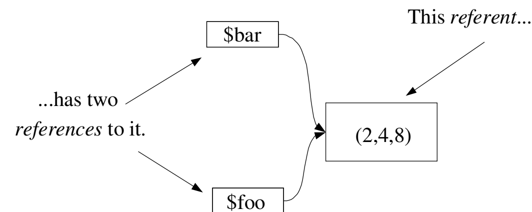
A Perl Secret, Continued

- But, when you explicitly name a variable, you can't use it like a reference. So:

```
$foo = "baz";   # Can't use $foo as a reference
$bar = \$foo;   # $bar is a true hard reference
```

- However, internally to Perl, there are two things ($foo and $bar) that refer to the string "baz". This is important for garbage collection - Perl only cleans up memory when nothing is referring to it any more.

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.

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
```

Anonymous Data

- In most of the examples on the previous slide, we referred to data that was already stored in another variable.
- However, `$bar = \412.63` created a reference to a piece of data that was not named by any variable. This is called *anonymous data*, because it doesn't have a name and is only reachable through the reference that refers to it.

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

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];`
- Hrmmm. What did we just do there? How many elements does that anonymous array have?

References are Scalars

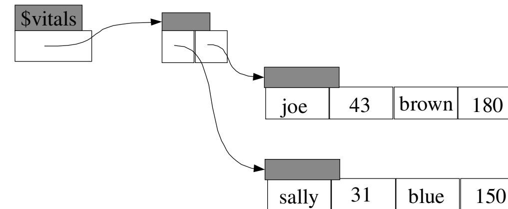
- A reference is the third of the three kinds of scalar data types (along with strings and numbers).
- Thus, you can use a reference any place and manner in which you would use any other scalar
- This means that references can be array elements and hash values. If they happen to refer to other arrays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

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....

A Multidimensional Array

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



Anonymous Hashes

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


Anonymous Hashes, Continued

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

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.

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 prepending the appropriate character to it.
- $$foo is “the scalar value that is referred to by the scalar value of foo.” If $foo is a reference to a scalar, $$foo is that scalar (referent) to which we are referring.

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

Dereferencing Examples

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

The Second Way to Dereference

- Use a { } block

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

# constructing hash of anon sub references
# sub {...}; returns a ref to an anon subroutine
# don't confuse it with sub NAME { ... }
my $hash= ( sayhi => sub {"hello"}; ,
            saybye => sub {"bye"}; ,
          );
print &{ $hash{sayhi} }; # {} not optional here!
# &$hash{sayhi} Perl treats $hash as the ref
```


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';

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
```


Style Notes

- Using Method 1 ($$ref) can get tricky

```
• my ($foo, $bar) = ("foo", "bar");
  my @a = (\$foo, \$bar);

  # We want to dereference the 2nd element of @a
  print $$a[1];  # WRONG - tries $a->[1]
  print ${$a[1]};  # Right
```

- For this reason, many programmers always use the { } or -> notation.

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";
}
```


Arrays of Arrays

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

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

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


Array of Arrays, Continued

- Building one dynamically:

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

- Shorter, same thing:

```
while (<>) {
  push @AoA, [ split ];
}
```


Sidetrack: Autovivifying References

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

```
while (<>) {
  push @$AoAref, [ split ];
}
```

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

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)
```


Back to 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 "Element $i $j is $tictactoe[$i][$j]\n";
 }
}
```


Reference to an Array of Arrays

```
my $tttref = [ ['-','-','x'],
               ['-','x','-'],
               ['o','o','-'] ];

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

- What does the broken line try to do?

Deeper into the Perl Mind Meld

```
for $i (1..10) {
  @array = somefunc();
  $AoA[$i] = @array;     #WRONG
}
```

- Who can tell me why this is broken?

Deeper into the Perl Mind Meld

```
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.

Deeper into the Perl Mind Meld

```
for $i (1..10) {
  @array = somefunc();
  $AoA[$i] = \@array;     #STILL WRONG
}
```

- Who can tell me why this is broken?

Deeper into the Perl Mind Meld

```
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.

Deeper into the Perl Mind Meld

```
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.

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. As a bonus, it's more efficient than the other way. (Who knows why? Anyone? Bueller? Anyone get that joke?)

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 [ ]

Deeper into the Perl Mind Meld

```
for $i (1..10) {
  $AoA[$i] = [ somefunc() ];     #Also Right
}
```

- Even better than the last slide. Why?

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.

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
```


Array of Hashes

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


Array of Hashes, Continued

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


Hash of Hashes

- The most flexible compound data structure

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

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";
  }
```


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";
  }
```


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?