CS 105 Perl: References

Nathan Clement

February 13, 2014
Agenda

We will cover references.

- References
  - Creating references
  - Dereferencing
  - Creating array and hash refs
- Autovivification
- Defined-or operator
In Perl, references are used to form nested data structures. What if I want to put a hash inside a hash?

```perl
%hoh = (   # hash of hashes
    a => ???,
    b => ???,
    c => ???,
) ;
```
Our Wish, Graphically

%hoh

a

b

c
Good Idea—But Wrong

- You might be tempted to put a hash inside the list for initializing the hash.

- What if I want to put a hash inside a hash?

```perl
%a = ( ... );
%b = ( ... );
%hoh = (   # hash of hashes
    a => %a,
    b => %b
);
```
Good Idea—But Wrong

• You might be tempted to put a hash inside the list for initializing the hash.

• What if I want to put a hash inside a hash?

```perl
%a = ( ... );
%b = ( ... );
%hoh = (   # hash of hashes
    a => %a,
    b => %b
);
```

• This is wrong. Think about how hashes flatten into lists.
One Way to Do It...

• We must use a *reference* to the hash instead.

```perl
%a = ( ... );
%b = ( ... );
%hoh = (    # hash of hashes
    a => \%a,
    b => \%b
 );
```

• This will work, but is not ideal.
Graphical Description of Previous Example
Important Clarification

• A reference (no matter what kind) is a scalar.

• This is why we depict the reference as a dot inside of a value box. An arrow is drawn to whatever it points to (array, hash, etc.).
Anonymous Hashes

To create an *anonymous hash*, use `{ }` instead of `( )`.

```ruby
%hoh = (   # hash of hashes
  a => { ... },  
  b => { ... }  
); 
```

This is better.
Anonymous Hashes

Notice that the references are *unnamed* or *anonymous.*
Nested Hash Example

• Syntactically, think of the \{ \} block like a list literal, i.e. ( ).

```
%hoh = (   # hash of hashes
  a => { foo => 1, high => 'low' },
  b => { bar => 2, fish => 'chips' },
);
```
Nested Array Example

- To create an anonymous array, use [ ] instead of ( ). This returns an array reference.

```perl
@aoa = (   # array of arrays
    [ 1, 2, 3, 4 ],
    [ qw(a b c d) ]
);  
```
Array of Arrays, Graphically
Accessing Nested Arrays and Hashes

• For a nested structure, you can just add another level of lookup with `[ ]` or `{ }`.

```perl
print "$aoa[1][0]\n";  # 'a'
print "$menu{ sidesof }{ fish}\n";  # 'chips'
```
Accessing Nested Arrays and Hashes (2)

- There are two ways to directly access an array or hash reference stored in a scalar.

- This is the first way:

```perl
$letters = $aoa[1];
${$letters}[0];       # 'a'

$sides = $menu{sidesof};
${$sides}{fish};     # 'chips'
```
Accessing Nested Arrays and Hashes (3)

- There are two ways to directly access an array or hash reference stored in a scalar.

- This is the preferred way:

```perl
$letters = $aoa[1];
$letters->[0];  # 'a'

$sides = $menu{sidesof};
$sides->{fish};  # 'chips'
```
Accessing Nested Arrays and Hashes (4)

• You can use `->` even when it’s not required, if you like.

```perl
print "$aoa[1]->[0]\n"; # 'a'
print "$hoh{b}->{fish}\n"; # 'chips'
```
Scalar References

- While not as helpful, you can also create references to scalars.

```perl
my $a = 'a';
my $b = $a;

my $c = 'b';

# dereference
${$b};
$$c;
```
Other References

• Also can have references to functions and files:

```perl
my bake_something_ref =
    sub { say “Making a $_[0]”; }
$bake_something_ref->(“Pie”);
```
Iterating Over Nested Arrays and Hashes

- You can use the `@` sigil to dereference an array reference or the `%` sigil to dereference a hash reference.

```perl
foreach (@{$aoa[0]}) {
    # ...
}

foreach (keys %{$hoh{b}}) {
    # ...
}
```
Be Aware of Copying

• If you dereference an array or hash reference and assign it to an array or hash (respectively), you have created a copy.

```perl
@a = @{ $arrayref };  # copies the array
foreach ... {
    my @a = @{ $arrayref };
    # uses $a[0] ...
}
```

• This will quickly get very expensive.

```perl
# better is $arrayref->[0]
# push @{ $arrayref }, ... etc.
```
Want to Know More?

- **perlref** - the reference
- **perlreftut** - a tutorial/introduction
my $a;  # ***
while(<>) {
    die unless /^ \s* (\d+) \s+ (\S+) \s+ \s* (\S+) \s* $/x;
    $a->[$1]->{$2} = $3;
}
Autovivification

my $a; # ***
while(<>) {
    die unless /^ \s* (\d+) \s+ (\S+) \s+ (\S+) \s* $/x;
    $a->[${1}]->{${2}} = ${3};
}

Notice how we assume that $a is an array reference, and that any value in that array is a hash reference. This is because Perl will create these data structures automatically. This process where Perl creates data structures by inferring their presence is called autovivification.
Autovivification Details

- Perl will only do this if the value that should be a reference is `undef`.

- The word `autovivification` means the process whereby something comes to life by itself. This is very appropriate since Perl brings data structures to life automatically, by itself, with no programmer effort.
Autovivification - Motivation

• Autovivification is a handy feature because it reduces the effort necessary to create data structures. (These savings are passed directly to you, the programmer.) Without this feature, we would need to write this:

```perl
my $a = []; # declaration here
while(<>) {
    die unless /^ \s* (\d+) \s+ (\S+) \s+ (\S+) \s* $/x;
    $a->[0] //= {};  # defined-or here
    $a->[0]->{$2} = $3;
}
```

• Compare this to the original example to see what code can be elided by letting autovivification do the work.
Defined-or Operator

• The previous example used a Perl 5.10 feature known as the "defined-or" operator. You don’t have to use the `use v5.10` pragma to use it, but of course it won’t work on older Perl versions.

• Defined-or (`//=`) is analogous to the regular or operator except it uses definedness (think `defined`) rather than truth.
  ```perl
  $a->[1] //= {};  
  ```

• The hash reference will only be created if that element of the array is `undef`.