1.1.2.8.2

Intermediate Perl Session 2

- references, continued
- function references
- dispatch tables
- `Data::Dumper`

Reference table to references

	array @a = (1,2,3)	hash %h = (one=>1,two=>2)
reference created from variable	$a_ref = \@a	$h_ref = \%h
anonymous reference	$a_ref = [1,2,3]	$h_ref = {one=>1,two=>2}
dereferencing	$a_ref->[0] $a_ref->[1] $a_ref->[2]	$h_ref->{one} $h_ref->{two}

Writing functions that return list references

- we want to make a list of the first N perfect squares

```
# function returns list



@squares = create_squares(100);

for $square (@squares) {
  print $square,"\n";
}

sub create_squares {
  $n = shift;
  @x = ();
  for $i (1..$n) {
    push @x, $i**2;
  }
  return @x;
}
```

```
# function returns reference
# but internally works with list


$squares = create_squares(100);

for $square (@$squares) {
  print $square,"\n";
}

sub create_squares {
  $n = shift;
  @x = ();
  for $i (1..$n) {
    push @x, $i**2;
  }
  return \@x;
}
```

```
# function returns list reference
# and internally works with list
# reference

$squares = create_squares(100);

for $square (@$squares) {
  print $square,"\n";
}

sub create_squares {
  $n = shift;
  $x = [];
  for $i (1..$n) {
    push @$x, $i**2;
  }
  return $x;
}
```

Writing functions that return hash references

- we want to make a hash of first N letters and their alphabet position
  - key is letter, value is order in alphabet

```
# function returns hash


%letters = create_letters(20);

for $letter (sort keys %letters) {
  print $letter,$letters{$letter},"\n"
}

sub create_letters {
  $n = shift;
  %h = ();
  $n = 26 if $n > 26;
  $i = 1;
  for $let ( (a..z)[0..$n-1]) {
    $h{$let} = $i++;
  }
  return %h;
}
```

```
a 1
b 2
c 3
d 4
e 5
f 6
g 7
h 8
i 9
j 10
...
```

```
# function returns hash
# a little tidier

%letters = create_letters(20);

for $letter (sort keys %letters) {
  print $letter,$letters{$letter},"\n"
}

sub create_letters {
  $n = shift;
  %h = ();
  $n = 26 if $n > 26;
  for $let ( (a..z)[0..$n-1]) {
    $h{$let} = 1 + scalar keys %h;
  }
  return %h;
}
```

Writing functions that return hash references

- we want to make a hash of first N letters and their alphabet position
  - key is letter, value is order in alphabet

```
# function returns hash

%letters = create_letters(20);

for $letter (sort keys %letters) {
  print $letter,$letters{$letter},"\n"
}

sub create_letters {
  $n = shift;
  %h = ();
  $n = 26 if $n > 26;
  for $let ( (a..z)[0..$n-1]) {
    $h{$let} = 1 + keys %h;
  }
  return %h;
}
```

```
a 1
b 2
c 3
d 4
e 5
f 6
g 7
h 8
i 9
j 10
...
```

```
# function returns hash reference

$letters = create_letters(20);

for $letter (sort keys %$letters) {
  print $letter,$letters->{$letter},"\n"
}

sub create_letters {
  $n = shift;
  $h = {};
  $n = 26 if $n > 26;
  for $let ( (a..z)[0..$n-1]) {
    $h->{$let} = 1 + keys %$h;
  }
  return $h;
}
```

writing functions that modify referents

- you can pass a reference to a function in order to
  - access or modify the referent

```
# reference to list of squares
$squares = create_squares(100);

# print squares
print_squares($squares);

# format list in place to add decimal
format_squares($squares);

sub print_squares {
  $sref = shift;
  for $s (@$sref) {
    print $s;
  }
}

sub format_squares {
  $sref = shift;
  for $i (0..@$sref-1) {
    $sref->[$i] = sprintf("%.1f",$sref->[$i]);
  }
}
```

```
# this also works - why?

sub format_squares {
  $sref = shift;
  for $s (@$sref) {
    $s = sprintf("%.1f",$s);
  }
}

# but not this - why?

sub format_squares {
  $sref = shift;
  for $s (@$sref) {
    $square = $s;
    $square = sprintf("%.1f",$square);
  }
}
```

diagnosing data structures with Data::Dumper

- use Dumper() to create stringified data structure
  - man Data::Dumper to learn more

```
sub create_letters {
  $n = shift;
  $h = {};
  $n = 26 if $n > 26;
  for $let ( (a..z)[0..$n-1]) {
    $h->{$let} = 1 + keys %$h;
  }
  return $h;
}
```

```
use Data::Dumper;

$letters = create_letters(20);
print Dumper($letters);

$VAR1 = {
          'e' => 5,
          'a' => 1,
          'd' => 4,
          'j' => 10,
          'c' => 3,
          'h' => 8,
          'b' => 2,
          'g' => 7,
          'f' => 6,
          'i' => 9
        };
```

```
# Sortkeys is one of the
# customizable parameters
# in the module

$Data::Dumper::Sortkeys = 1;
print Dumper($letters);

$VAR1 = {
          'a' => 1,
          'b' => 2,
          'c' => 3,
          'd' => 4,
          'e' => 5,
          'f' => 6,
          'g' => 7,
          'h' => 8,
          'i' => 9,
          'j' => 10
        };
```

diagnosing data structures with Data::Dumper

- let's start to build a more complicated structure

```
$letters = create_letters(6);

print Dumper($letters);

sub create_letters {
  $n = shift;
  $h = {};
  $n = 26 if $n > 26;
  for $let ( (a..z)[0..$n-1]) {
    # value associated with the letter
    # is now a hash reference
    $h->{$let} = {idx => 1 + keys %$h};

    # could also write this as
    # $h->{$let}{idx} = 1 + keys %$h;

  }

  return $h;
}
```

```
$VAR1 = {
          'a' => {
                   'idx' => 1
                 },
          'b' => {
                   'idx' => 2
                 },
          'c' => {
                   'idx' => 3
                 },
          'd' => {
                   'idx' => 4
                 },
          'e' => {
                   'idx' => 5
                 },
          'f' => {
                   'idx' => 6
                 },
        };
```

diagnosing data structures with Data::Dumper

- let's start to build a more complicated structure

```
$letters = create_letters(3);

for $letter (keys %$letters) {

  # ord key will store the ASCII value, e.g. a = 97
  $letters->{$letter}{ord} = ord($letter);

  # reverse key will store corresponding letter from end of alphabet
  $letters->{$letter}{reverse} = (reverse a..z)[ $letters->{$letter}{idx} -1 ];

  # prev is a list reference of all letters before $letter
  $letters->{$letter}{prev} = [ (a..z)[ 0 .. $letters->{$letter}{idx} - 2 ];

  # next is a list reference of all letters after $letter
  $letters->{$letter}{next} = [ (a..z)[ $letters->{$letter}{idx} .. 25 ];
}
```

```
'c' => {
          'idx' => 3,
          'next' => [
                      'd',
                      'e',
                      'f',
                      ...
                      'x',
                      'y',
                      'z'
                    ],
          'ord' => 99,
          'prev' => [
                      'a',
                      'b'
                    ],
          'rev' => 'x'
        }
```

diagnosing data structures with Data::Dumper

- use a temporary variable to store the current hash value

```
$letters = create_letters(3);

for $letter (keys %$letters) {
  # store the value (which is a hash reference) in a temporary variable
  $h = $letters->{$letter};
  # now use the temporary variable to make code more succinct
  $h->{ord} = ord($letter);
  $h->{reverse} = (reverse a..z)[ $h->{idx} -1 ];
  $h->{prev} = [ (a..z)[ 0 .. $h->{idx} - 2 ];
  $h->{next} = [ (a..z)[ $h->{idx} .. 25 ];
}
```

some common error messages

- here are the kinds of error messages you will see if you try to dereference non-references or the wrong references

```
$x = "sheep";

$x->[0] = 1; # nope, you can't dereference a non-reference
Can't use string ("sheep") as an ARRAY ref while "strict refs" in use at ./somescript line 10.

$x->{a} = 1; # nope, you can't dereference a non-reference
Can't use string ("sheep") as a HASH ref while "strict refs" in use at ./somescript line 10.

$x = [1];    # now $x is a list reference
$x->{a} = 1; # try to treat it like a hash reference
Can't coerce array into hash at ./somescript line 10.

$x = {sheep=>1}; # now $x is a  hash reference
$x->[0] = 1;     # try to treat it like an array reference
Not an ARRAY reference at ./somescript line 10.

$x = {sheep=>1};
$x->{sheep} = "meeh";
$x->{sheep}[0] = 2;
Can't use string ("meeh") as an ARRAY ref while "strict refs" in use at ./somescript line 11.
```

Freezing data structures with Storable

- the Storable module allows you to
  - store/retrieve data structures to/from disk

```
# script 1

use Storable;

$letters = create_letters(26);

# store(REFERENCE,FILE)
store($letters,"letters.cache");
```

```
# script 2

use Storable;

# retrieve(FILE)
$letters = retrieve("letters.cache");

# now use $letters as desired
```

```
# the script
-rwxr-xr-x   1 martink users        424 Jul 29 13:26 letterorder
# the cache file
-rw-r--r--   1 martink users        540 Jul 29 13:26 letters.cache
```

function references – the final reference frontier

- just like with lists and arrays, you can have a reference to a function
  - the sigil for functions is &
  - the dereference syntax is ->(ARGUMENTS)

```
sub square {
  my $num = shift;
  return $num**2;
}

print square(2);                4

my $square_ref = \&square;

print $square_ref;              CODE(0x818d9e4)
print $square_ref->(2);         4
```

creating dispatch tables – from strings to actions

- you can store references to functions in a hash

```
sub square {
  my $num = shift;
  return $num**2;
}

sub root {
  my $num = shift;
  return sqrt($num);
}

# create a hash of functions
# this is called a dispatch table

my $functions = { do_root   => \&root,
                  do_square => \&square };

# call functions by name (i.e. by key value)

$functions->{do_root}(2)        1.414
$functions->{do_square}(2)      4
```

anonymous function references

- sub{} actually returns the code reference to the function

```
$do_square = sub { return $_[0]**2 };
$do_root   = sub { return sqrt($_[0]) };

# create a hash of functions
# this is called a dispatch table

my $functions = { do_root   => $do_root,
                  do_square => $do_square };

# call functions by name (i.e. by key value)

$functions->{do_root}(2)        1.414
$functions->{do_square}(2)      4
```

function templates

- you can write functions that return function references
  - sometimes called "factories"
  - disregard the "my" for now – this is a variable scoping thing we'll see soon

```
sub power_factory {
  my $power = shift;
  my $fn = sub { return $_[0]**$power };
  return $fn;
}

$do_square = power_factory(2)
$do_root   = power_factory(0.5);
$do_5      = power_factory(5);

print $do_square->(2)    4
print $do_root->(3)      1.732
print $do_5->(4)         1024
```

1.1.2.8.2

Intermediate Perl
Session 2

- use `Data::Dumper` to visualize data structures
- create function references with `\&FUNCTION`
- create function references with `$ref = sub { }`