References
&
Multi-Dimensional Data Structures

Sofia Robb

What good are references?

Sometimes you need a more complex data structure than just an array or just a hash.

What if you want to keep together several related pieces of information?

Gene	Sequence	Organism
HOXB2	ATCAGCAATATACAATTATAAAGGCCTAAATTTAAAA	mouse
HDAC1	GAGCGGAGCCGCGGGCGGGAGGGCGGACGGAC	human

References are only addresses.

Multi-dimensional data structures are just hashes and arrays inside of hashes and arrays.

References

- References are pointers, or the address of the data
  - All data has an address in memory
  - Humans have no need to know the address

- References are useful because they are a scalar variable.
  - Arrays and hashes are not scalar variables.
  - The only kind of data that you can store in an array or hash is scalar.

We can now store hashes and arrays in hashes and arrays by storing the address!!!!

What is a reference, what do you mean by an address?

Well first, what is a variable?

A variable is a label for the location in memory of some data. This location has an address.

Scalar

`$x=1;`

really means

address

`0x84048ec`

SCALAR x:	1

Array

```
@y = (1, ‘a’, 23);
```

really means

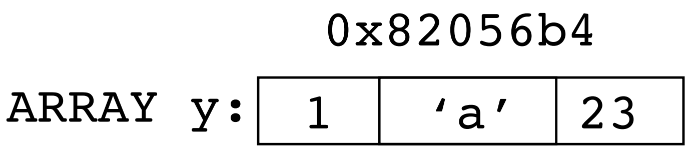
`0x82056b4`

ARRAY y:	1	‘a’	23

How do I find you, what’s your address?

A variable is a labeled memory address.

When we read the contents of the variable, we are reading the contents of the memory address.



0x82056b4

ARRAY y: 1 ‘a’ 23

So, what is a reference?

A reference is a variable that contains the memory address of some data.

> It does not contain the data itself.
>
> It contains the memory address where data is stored.

Creating a Reference

- Every time a variable is created it gets an address
- To retrieve the address or in other words, create a reference, use '\'

Creating a Reference to an Array

```
# codons for my favorite gene: HDAC
my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC TAC TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);


my $address = \@codons;
print "$address\n";
```

Output:

```
%% ./references.pl
ARRAY(0x100812e30)
```

$address is now a reference to the array.

Creating a Reference to a Hash

```
my %HDAC;

$HDAC{seq}      = "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";
$HDAC{function} = "Histone Deacetylase";
$HDAC{symbol}   = "HDAC";


my $address = \%HDAC;
print "$address\n";
```

Output:

```
%% ./references.pl
HASH(0x10081e538)
```

Storing References

Now that we have a way to retrieve the address we can store (assign) an array or a hash in an array or hash.

- Arrays are a list of scalars
- Hashes are key/value pairs of scalars
- References are scalars

Storing a Reference as a Hash Value

```
use Data::Dumper;

my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT
TAC TAC TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);

my $codons_address = \@codons;

my %HDAC;
$HDAC{seq}= "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";
$HDAC{function} = "Histone Deacetylase";
$HDAC{symbol}   = "HDAC";
$HDAC{codons}   = $codons_address;

## using Data::Dumper to print our data structure
print Dumper \%HDAC;
```

Notice the hash reference.

output:

```
$VAR1 = {
          'symbol'   => 'HDAC',
          'function' => 'Histone Deacetylase',
          'seq'      => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',
          'codons'   => [
                        'ATG',
                        'GCG',
                        'CAG',
                        'ACG',
                        'CAG',
                        'GGC',
                        'ACC',
                        'CGG',
                        'AGG',
                        'AAA',
                        'GTC',
                        'TGT',
                        'TAC',
                        'TAC',
                        'TAC',
                        'GAC',
                        'GGG',
                        'GAT',
                        'GTT',
                        'GGA',
                        'AAT',
                        'TAC',
                        'TAT',
                        'TAT'
                      ]
        };
```

Data::Dumper is a nice way to view the contents of your data structures without complicated print statements.

Or you could use the debugger.

Altering the data

Addresses/References are like Short Cuts/Aliases

- References are NOT copies of the data. They are addresses or pointers to the data
- Since a reference is like a short cut (windows) or alias (mac), when the original data changes, the change can be seen when using the reference to access the data.
- So, if @codons is changed, the hash also changes, because the hash contains only the address of the array, not a copy of the array.

Altering the Original Array affects the reference

```
my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC TAC
TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);

my $codons_address = \@codons;

my %HDAC;
$HDAC{seq}= "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";
$HDAC{function} = "Histone Deacetylase";
$HDAC{symbol}   = "HDAC";
$HDAC{codons}   = $codons_address;

#Replacing the contents of @codons with only 2 codons
@codons = qw(ATG GCG);

#Printing the unaltered %HDAC
print Dumper \%HDAC;
```

Output:

```
$VAR1 = {
          'symbol' => 'HDAC',
          'function' => 'Histone Deacetylase',
          'seq' => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',
          'codons' => [
                        'ATG',
                        'GCG'
                      ]
        };
```

Only @codons was altered but the hash also changed

Anonymous Data structures

- You do not always need to retrieve the address of data to store/assign in a variable.
- You can create an anonymous array or hash on the fly.
  - It is anonymous because it is unnamed.
  - It only has an address, no name, no label.
- We use the [ ] in the anonymous array assignment
- We use the {} in the anonymous hash assignment.

Creating and Storing an Anonymous Array

the array is never given a name.

```
Before:
my @codons = qw(ATG GCG);
my @address = \@codons;
$HDAC{codons} = $address;
```

```
Now:Creating and storing an anonymous array
$HDAC{codons}    = [ "ATG" , "GCG" ] ;
```

Notice the [ ] instead of ().

Storing an Anonymous (unnamed) Array as a Hash Value

~~#my @codons = qw(ATG GCG);~~

```
my %HDAC;
$HDAC{seq}= "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";
$HDAC{function} = "Histone Deacetylase";
$HDAC{symbol}   = "HDAC";
$HDAC{codons}   = [ "ATG" , "GCG" ] ;

print Dumper \%HDAC;
```

the array is never given a name.

Output:

```
$VAR1 = {
          'symbol' => 'HDAC',
          'function' => 'Histone Deacetylase',
          'seq' => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',
          'codons' => [
                        'ATG',
                        'GCG'
                      ]
        };
```

Storing an Anonymous (unnamed) Hash as a Hash Value

```
my %HDAC;
$HDAC{seq}= "MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...";
$HDAC{function}   = "Histone Deacetylase";
$HDAC{symbol}     = "HDAC";
$HDAC{codons}     = [ "ATG" , "GCG" ] ;
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;

print Dumper \%HDAC;
```

Notice the {} instead of ().

Output:

```
$VAR1 = {
          'symbol' => 'HDAC',
          'function' => 'Histone Deacetylase',
          'expression' => {
                            'heart' => '1.3',
                            'liver' => '2.1'
                          },
          'seq' => 'MAQTQGTRRKVCYYYDGDVGNYYYGQGHPMKPHRIR...',
          'codons' => [
                        'ATG',
                        'GCG'
                      ]
        };
```

Storing an Anonymous (unnamed) Hash as a Hash Value

```
$HDAC{expression} = { "liver" => 2.1 ,
                       "heart" => 1.3
                     } ;
```

Same As:

```
$HDAC{expression}{"liver"} = 2.1 ;
$HDAC{expression}{"heart"} = 1.3 ;
```

Looks Like:

```
$HDAC{symbol}     = "HDAC";
```

Now, all the data is in the data structure, how to you get it out?

Whole chunks of data or pieces of data can be retrieved from the multidimensional structures by using the address.

A.K.A. **Dereferencing**

3 Easy Steps to Dereference

Dereference === retrieve data from address

1. Get the address, or reference: $ADDRESS
2. Wrap the address, or reference in {}: {$ADDRESS}
3. Put the symbol of the data type out front @: @{$ADDRESS}

Dereference a reference to an array

```
my @codons = qw(ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC
TAC TAC GAC GGG GAT GTT GGA AAT TAC TAT TAT);

my $codons_address = \@codons;

print "address of the array:\n$codons_address\n\n";
print "array from a dereferenced reference:\n @{$codons_address}\n";
```

Output:

```
address of the array:
ARRAY(0x7fd89c016b90)

array from a dereferenced reference:
ATG GCG CAG ACG CAG GGC ACC CGG AGG AAA GTC TGT TAC TAC TAC GAC GGG GAT GTT GGA
AAT TAC TAT TAT
```

Dereference an anonymous array that is a hash value

```
           Key                  Value
$HDAC{codons}    = [ "ATG" , "GCG" ] ;

my $codons_address = $HDAC{codons};

print "address of the array:\n$codons_address\n\n";
print "array from a dereferenced reference:\n @{$codons_address}\n";
```

Output:

```
address of the array:
ARRAY(0x7f97db822958)

array from a dereferenced reference:
ATG GCG
```

Regular hash

```
$hash{key} = "value";
my $value = $hash{key};
```

Did you notice that dereferencing an array and an anonymous array are the same?

Dereference an anonymous hash that is a hash value

```
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;

my $hash_address = $HDAC{expression};

print "address of the hash:\n$hash_address\n\n";
print "keys from a dereferenced reference:\n";

my @keys = keys %{$hash_address};

print "keys from a dereferenced reference:\n@keys\n";
```

Output:

```
address of the hash:
HASH(0x7f94e38226d0)

keys from a dereferenced reference:
heart liver
```

Regular hash

```
my @keys = keys %hash;
```

It is not always needed to explicitly retrieve the address

```
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;
```

~~my $hash_address = $HDAC{expression};~~
~~my @keys = keys %{$hash_address};~~

```
my @keys = keys %{ $HDAC{expression}  };
```

This evaluates to an address

```
print "keys from a dereferenced reference:\n@keys\n";
```

Output:

```
keys from a dereferenced reference:
heart liver
```

Regular hash

```
my @keys = keys %hash;
```

Dereferencing to access every element from the anonymous array that is a hash value

```
$HDAC{codons}   = [ "ATG" , "GCG" ] ;

##my @codons = @{ $HDAC{codons} };
##my $zeroth_element =  ${ $HDAC{codons} }[0];

foreach my $codon ( @ { $HDAC{codons} } ){
```

This evaluates to an address

```
       print "codon: $codon\n";
}
```

Output:

```
codon:  ATG
codon:  GCG
```

Regular array:

```
foreach my $codon ( @codons )
{
  print "codon: $codon\n";
}
```

Dereferencing to access a piece of the anonymous array that is a hash value.

```
$HDAC{codons}    = [ "ATG" , "GCG" ] ;
##my @codons = @{ $HDAC{codons} };

my $zeroth_element =  ${ $HDAC{codons} }[0];
```

This evaluates to an address

```
print "the 0th element = $zeroth_element\n";
```

Output:

```
the 0th element = ATG
```

Regular array

```
$array[1] = "value";
my $value = $array[1]
```

Dereferencing to access every key/value pair from the anonymous hash in a hash

```
$HDAC{expression} = { "liver" => 2.1 , "heart" => 1.3 } ;

foreach my $tissue ( keys % { $HDAC{expression} } ){
  my $level = ${ $HDAC{expression} }{$tissue};
  print "$tissue: $level\n";
}
```

Output:

```
heart: 1.3
liver: 2.1
```

Regular Hash

```
foreach my $key (keys %hash){
  my $value = $hash{$key};
}
```

The ref() function

ref(*REF*)
returns the data type in which the reference points

```
my %hash;

$hash{codons}= [“ATG” , “TTT”];
my $address = $hash{codons};

ref ( $address );          ## returns ARRAY
ref ( $hash{codons} );     ## returns ARRAY
```

both $address and $hash{codons} evaluate to the address of the array

Extra fun stuff to look over later.

- Array of arrays
- Another Scripting Example:
  - Creating a Hash of Hashes

Multidimensional Data: Making an Array of Arrays

```
my @spotarray = (
                 [0.124, 43.2, 0.102, 80.4],
                 [0.113, 60.7, 0.091, 22.6],
                 [0.084, 112.2, 0.144, 35.3]
                );

#  two ways to get the value of the inner index
# my $cell_1_0  = ${$spotarray[1]}[0];
my $cell_1_0  = $spotarray[1][0];

print $cell_1_0;
```

Output:

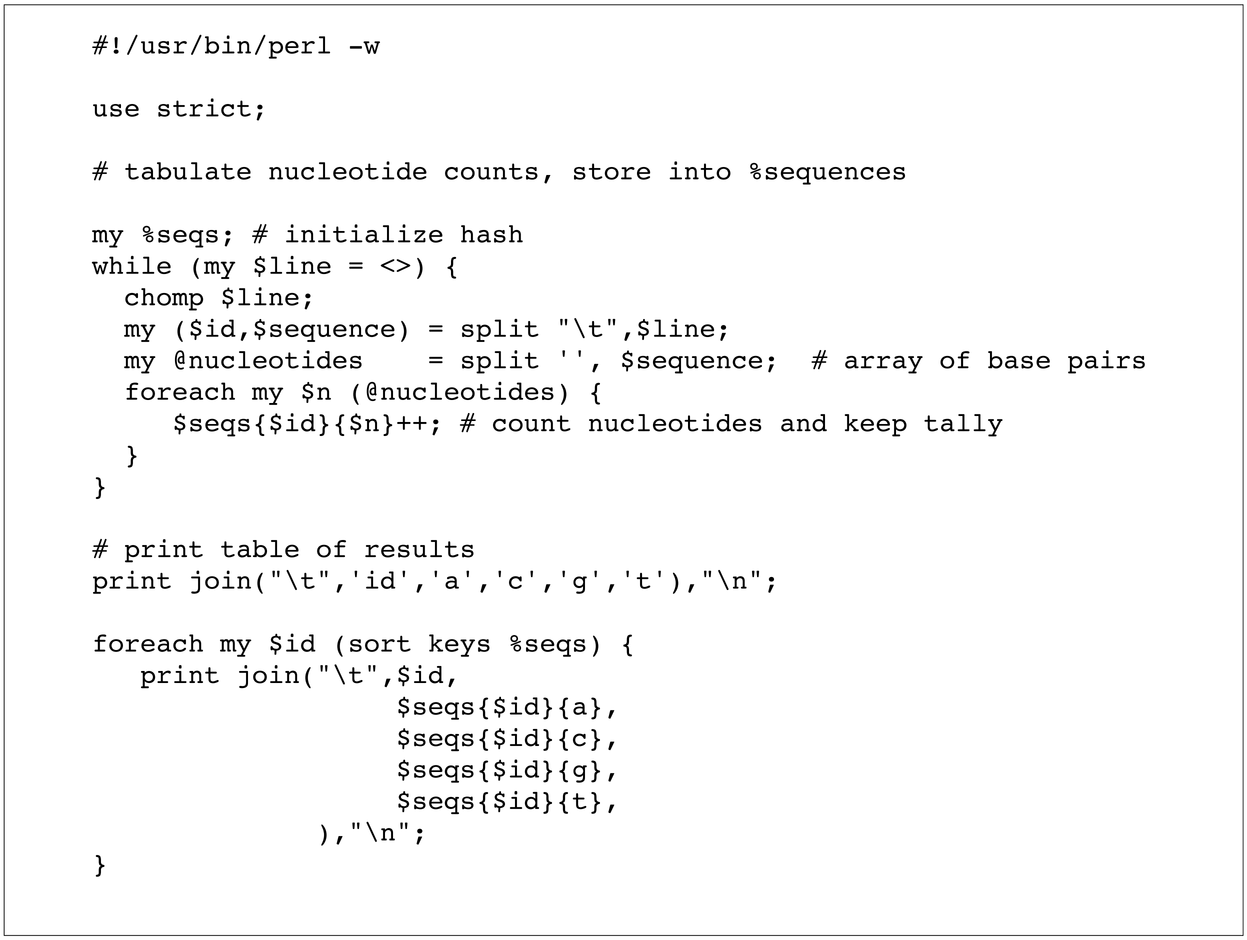
```
0.113
```

Scripting Example: Creating a Hash of Hashes

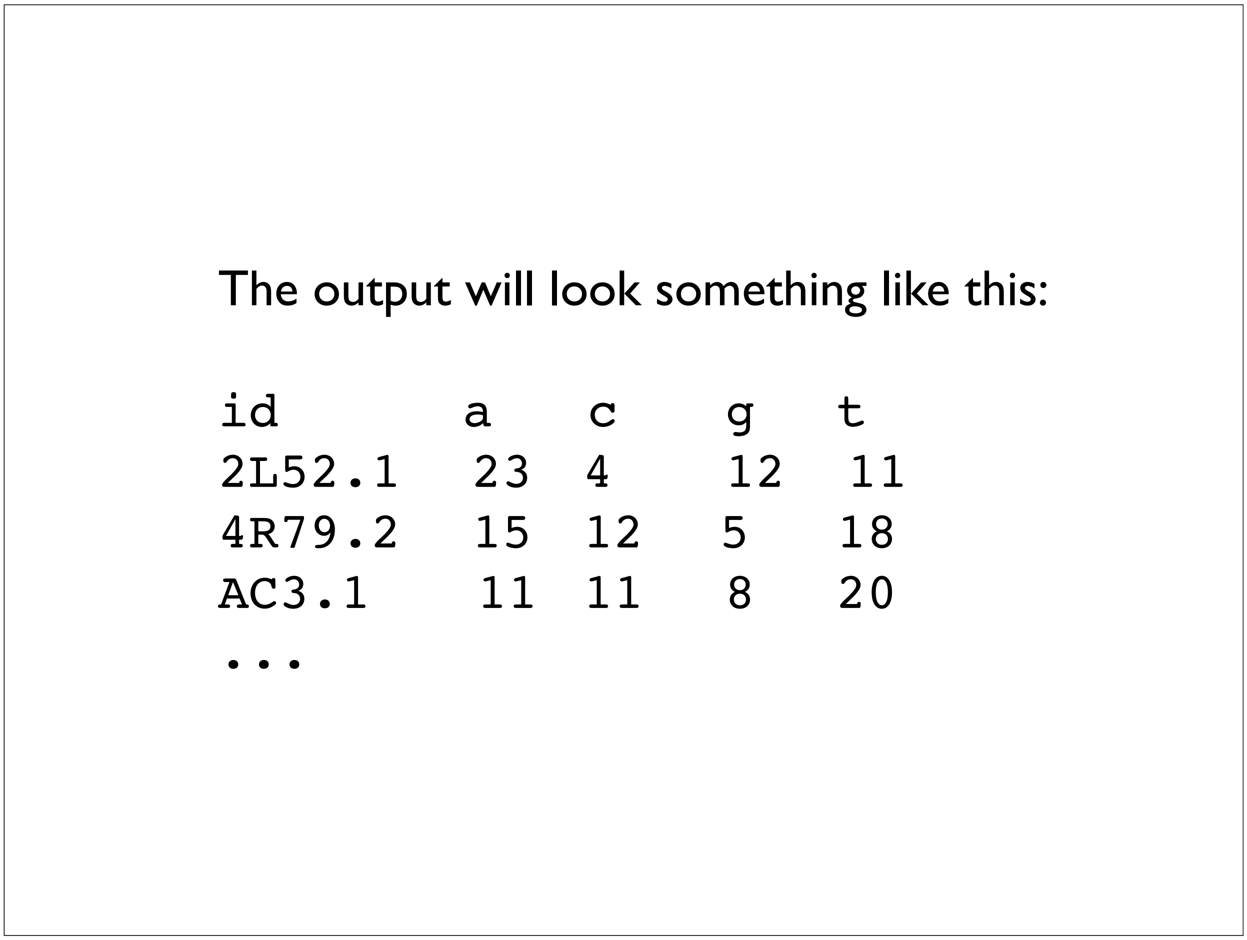
We are presented with a table of sequences in the following format: the ID of the sequence, followed by a tab, followed by the sequence itself.

```
2L52.1      atgtcaatggtaagaaatgtatcaaatcagagcgaaaaattggaagtaag...
4R79.2      tcaaatacagcaccagctcctttttttatagttcgaattaatgtccaact...
AC3.1       atggctcaaactttactatcacgtcatttccgtggtgtcaactgttattt...
...
```

For each sequence calculate the length of the sequence and the count for each nucleotide. Store the results into hash of hashes in which the outer hash's key is the ID of the sequence, and the inner hashes' keys are the names and counts of each nucleotide.



```
#!/usr/bin/perl -w

use strict;

# tabulate nucleotide counts, store into %sequences

my %seqs; # initialize hash
while (my $line = <>) {
  chomp $line;
  my ($id,$sequence) = split "\t",$line;
  my @nucleotides    = split '', $sequence;  # array of base pairs
  foreach my $n (@nucleotides) {
     $seqs{$id}{$n}++; # count nucleotides and keep tally
  }
}

# print table of results
print join("\t",'id','a','c','g','t'),"\n";

foreach my $id (sort keys %seqs) {
   print join("\t",$id,
                   $seqs{$id}{a},
                   $seqs{$id}{c},
                   $seqs{$id}{g},
                   $seqs{$id}{t},
              ),"\n";
}
```



The output will look something like this:

```
id      a   c    g   t
2L52.1   23  4    12  11
4R79.2   15  12   5   18
AC3.1    11  11   8   20
...
```