# 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?!?!? Multi-dimensional data structures?!?!?

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*

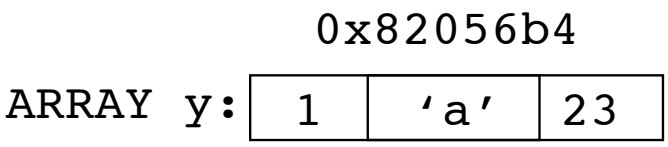
0x82056b4

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.






## 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 = ('ATG' , 'GCG' , 'CAG');

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

$address is now a reference to the array.

Output:

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

## Creating a Reference to a Hash

```
my %HDAC;

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


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

$address is now a reference to the hash.

Output:

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



## Storing References

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

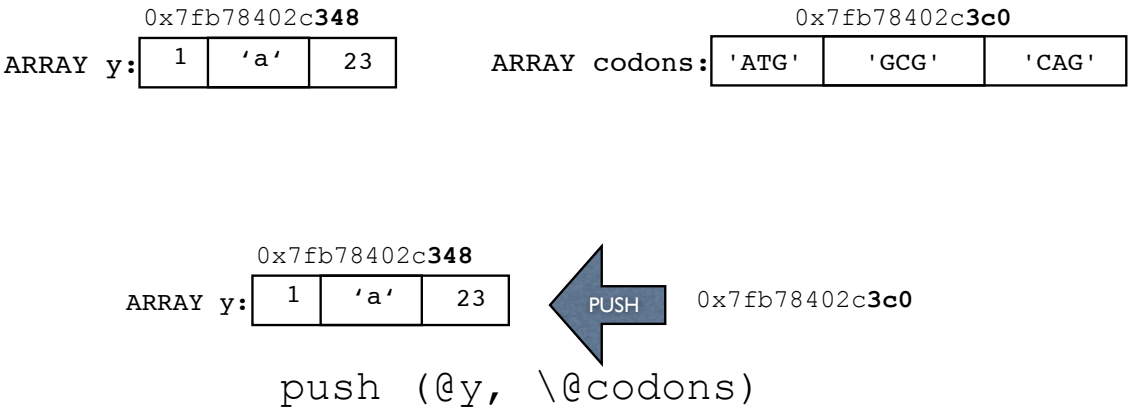
- Arrays are a list of scalars
- Hashes are key/value pairs of scalars
- References are scalars

# Storing an array reference in an array

```
my @y = (1, 'a' , 23 );     ##regular array
my $y_array_address = \@y; ##create a reference
print 'address of @y      : ' ,"$y_array_address\n";

my @codons = ('ATG' , 'GCG' , 'CAG'); #regular array
my $codons_array_address = \@codons;  #create a reference
print 'address of @codons : ', "$codons_array_address\n";

##store ref in regular array
push (@y , $codons_array_address);

## yeilds same as above
# push (@y, \@codons);
# $y[3] = \@codons;

print 'contents of @y     : ' , "@y\n";
print 'address of @y      : ' , \@y , "\n";
```

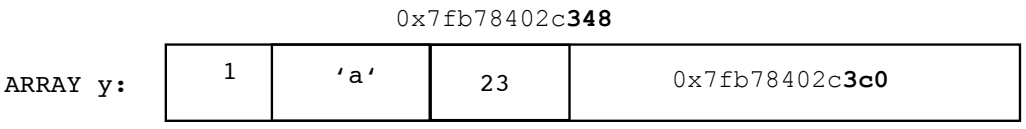
```
address of @y      : ARRAY(0x7fb78402c348)
address of @codons : ARRAY(0x7fb78402c3c0)
contents of @y     : 1 a 23 ARRAY(0x7fb78402c3c0)
address of @y      : ARRAY(0x7fb78402c348)
```






```
push (@y, \@codons)
```

```
## add the address of @codons (0x7fb78402c3c0) to the end of @y
```



# Storing a Reference as a Hash Value

```
use Data::Dumper;

my @codons = ('ATG' , 'GCG' , 'CAG');

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:

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

Or you could use the debugger.

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

# 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 = ('ATG','GCG','CAG');

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 = ('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 an Anonymous Array

**Before:**

```
my @codons = ('ATG' , 'GCG');
my $address = \@codons;
```

**Now:**

```
my $address = [ 'ATG' , 'GCG' ] ;
```

evaluates to an address

Notice the [ ] instead of ().

!!! the array is never given a name.
!!! it only has an address.

'Before' and 'Now' look different but are functionally the same.

**Check it out:**

```
print ['ATG','GCG'] , "\n";

Output:
ARRAY(0x7f9cf302bb08)
```

## Storing an Anonymous (unnamed) Array as a Hash Value

```
#my @codons =('ATG' , 'GCG');
#my $address = \@codons;

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

Notice the {} instead of ().

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

print Dumper \%HDAC;
```

### Output:

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

Regular hash

```
Set all at once:
%hash = (
  'key1' => 'value1',
  'key2' => 'value2',
 );
```

## Storing an Anonymous (unnamed) Hash as a Hash Value

All at once:

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

Notice the {} instead of ().

One at a time:

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

Regular hash

```
All at once:
%hash = (
  'key1' => 'value1',
  'key2' => 'value2',
 );
One at a time:
$hash{'key1'}="value1";
$hash{'key2'}="value2";
```



# 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 =('ATG' , 'GCG' , 'CAG' );

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
```

# Dereference an anonymous array that is a hash value

```
$HDAC{codons} = [ "ATG" , "GCG" ] ; #anony array is a hash value
                                    #anony array is an address
my $codons_address = $HDAC{codons};
print "address of the array: " , $HDAC{codons} , "\n";
print "address of the array: $codons_address\n\n";

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

Key

Value

This evaluates to an address

## Output:

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

array from a dereferenced reference:
ATG GCG
```

Did you notice that dereferencing an array and an anonymous array are the same? Check out the dereferencing in the last slide and compare to this one.

Regular hash

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



# 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";

my @keys = keys %{$hash_address};

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

This evaluates to an address

**Check it out:**

```
print {"liver"=>2.1, "heart"=>1.3},"\n";

Output:
HASH(0x7fadf082bbb0)
```

## Output:

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

keys from dereferenced reference:
heart liver
```

Regular hash

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

## It is not always needed to explicitly retrieve the address

This evaluates to an 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 of the anonymous array that is a hash value

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

#my @codons = @{ $HDAC{codons} };


foreach my $codon ( @ { $HDAC{codons} } ){
                      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];
                      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 a piece of the anonymous array that is a hash value.

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

my $zeroth_element =  ${ $HDAC{codons} }[0];
                      evaluates to an address

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


$last_element = pop @ { $HDAC{codons} };
print "the last element = $last_element\n";

## pop actually changes the array
```

## Output:

```
the 0th element  = ATG
the last element = GCG
```

Regular array

```
$array[1] = "value";
my $value = $array[1];
my $last = pop @array;
```

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

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

my $level = ${ $HDAC{expression} }{ "heart" };
print "heart: $level\n";

my $tissue = "liver";
$level = ${ $HDAC{expression} }{ $tissue };
print "liver: $level\n";
```

## Output:

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

Regular Hash

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



# 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};
}
```

Lets draw out what a hash of hashes would look like?



regular %genes hash

| key | value |
|---|---|
| gene1 | ATG |
| gene2 | CTT |
| gene3 | ATT |

multidimensional %genes hash

| key | value |
|---|---|
| gene1 | 0x543 |
| gene2 | 0x234 |
| gene3 | 0x152 |

Each key has a value that is an address to an anonymous hash

0x543

{anonymous hash}

| key | value |
|---|---|
| seq | TATGCC |
| desc | something |
| len | 6 |

Each key has a string as a value.

0x234

{anonymous hash}

| key | value |
|---|---|
| seq | CAAATG |
| desc | something |
| len | 6 |

Each key has a string as a value.

0x152

{anonymous hash}

| key | value |
|---|---|
| seq | TATACG |
| desc | something |
| len | 6 |

Each key has a string as a value.

```
$VAR1 = {
          'gene1' => {
                       'seq' => "TATGCC",
                       'desc' => 'something',
                       'len' => 6,
                     },

          'gene2' => {
                       'seq' => "CAAATG",
                       'desc' => 'something',
                       'len' => 6,

                     },
   };
```

# What about a hash of hashes?



multidimensional
%genes hash

| key | value |
|---|---|
| gene1 | 0x543 |
| gene2 | 0x234 |
| gene3 | 0x152 |

Each key has a value that is an address to an anonymous hash

{anonymous hash}

| key | value |
|---|---|
| seq | TATGCC |
| desc | something |
| len | 6 |
| nt_comp | 0x759 |

{anonymous hash}

| key | value |
|---|---|
| A | 1 |
| T | 2 |
| G | 1 |
| C | 2 |

Each key has a string as a value.

{anonymous hash}

| key | value |
|---|---|
| seq | CAAATG |
| desc | something |
| len | 6 |
| nt_comp | 0x191 |

{anonymous hash}

| key | value |
|---|---|
| A | 3 |
| T | 1 |
| G | 1 |
| C | 1 |

Each key has a string as a value.

```
$VAR1 = {
          'gene1' => {
                       'seq' => "TATGCC",
                       'desc' => 'something',
                       'len' => 6,
                       'nt_comp' => {
                                      'A' => 1,
                                      'T' => 2,
                                      'G' => 1,
                                      'C' => 2,
                                    }
                     },

          'gene2' => {
                       'seq' => "CAAATG",
                       'desc' => 'something',
                       'len' => 6,
                       'nt_comp' => {
                                      'A' => 3,
                                      'T' => 1,
                                      'G' => 1,
                                      'C' => 1,
                                    }
                     }
```

## 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      tcaaatacagcaccagctccttttttatagttcgaattaatgtccaact...
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 $infile = shift @ARGV;
open IN , '<' , $infile or die "Can't open $infile $!\n";

my %seqs;
while (my $line = <IN>) {
  chomp $line;
  my ($id,$sequence) = split "\t",$line;
  my @nucleotides    = split '', $sequence;  # array of nts
  foreach my $n (@nucleotides) {
     $seqs{$id}{$n}++; # count nts 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
...
```

### Data::Dumper Output:

```
$VAR1 = {
          '2L52.1' => {
                        'c' => 4,
                        'a' => 23,
                        'g' => 12,
                        't' => 11
                      },
          '4R79.2' => {
                        'c' => 12,
                        'a' => 15,
                        'g' => 5,
                        't' => 18
                      }
        };
```