References and multidimensional data

Simon Prochnik, Dave Messina, Lincoln Stein, Steve Rozen

PfB 2011

What good are references?

Sometimes you need a more complex data structure than a list.

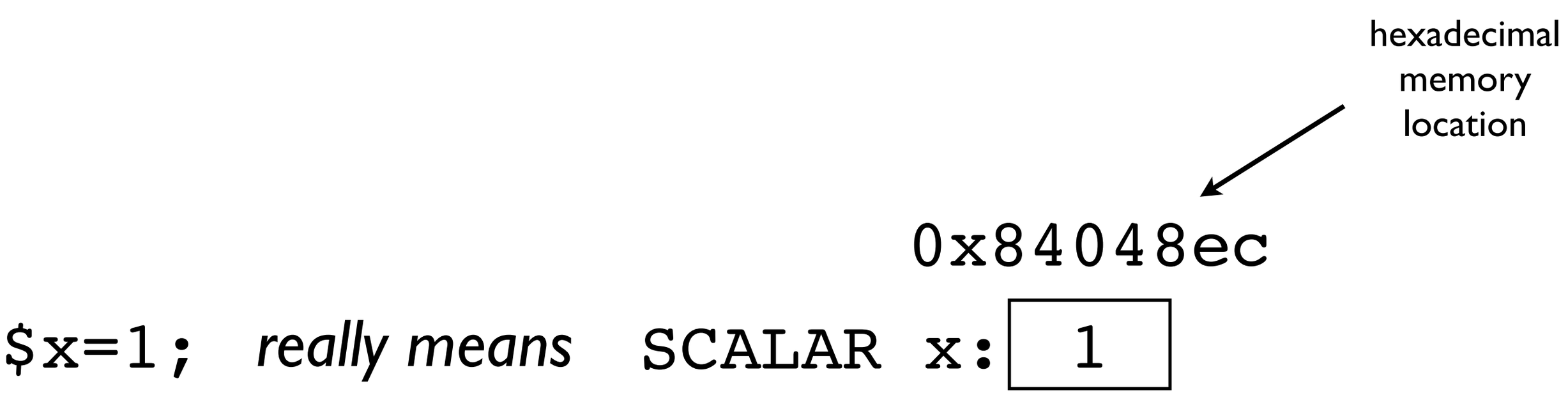
What if you want to keep together several related pieces of information?

Gene	Sequence	Organism
HOXB2	ATCAGCAATATACAATTATAAAGGCCTAAATTTAAAA	mouse
HDAC1	GAGCGGAGCCGCGGGCGGGAGGGCGGACGGAC	human

What is a reference?

Well first, what is a variable?

A variable is a labeled memory address that holds a value. The location's label is the name of the variable.



What is a list?

```
@y = (1, 'a', 23);
```

really means

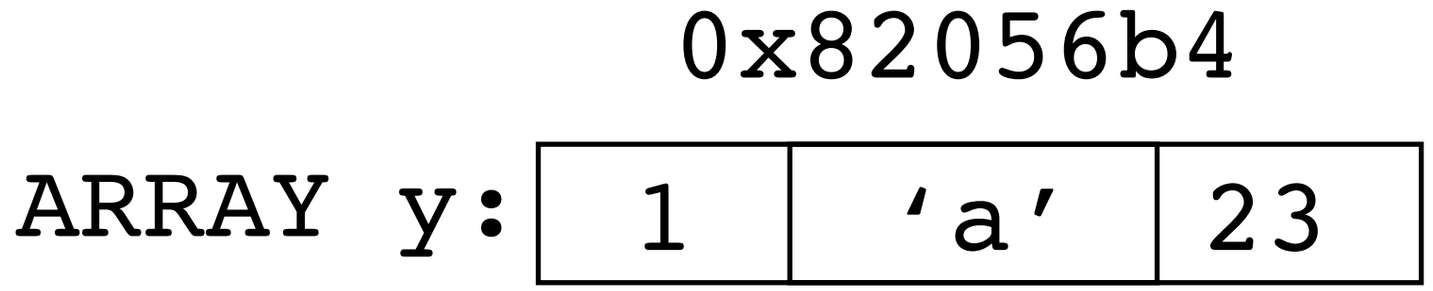
0x82056b4

ARRAY y:

1	'a'	23

A variable is a labeled memory address.

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



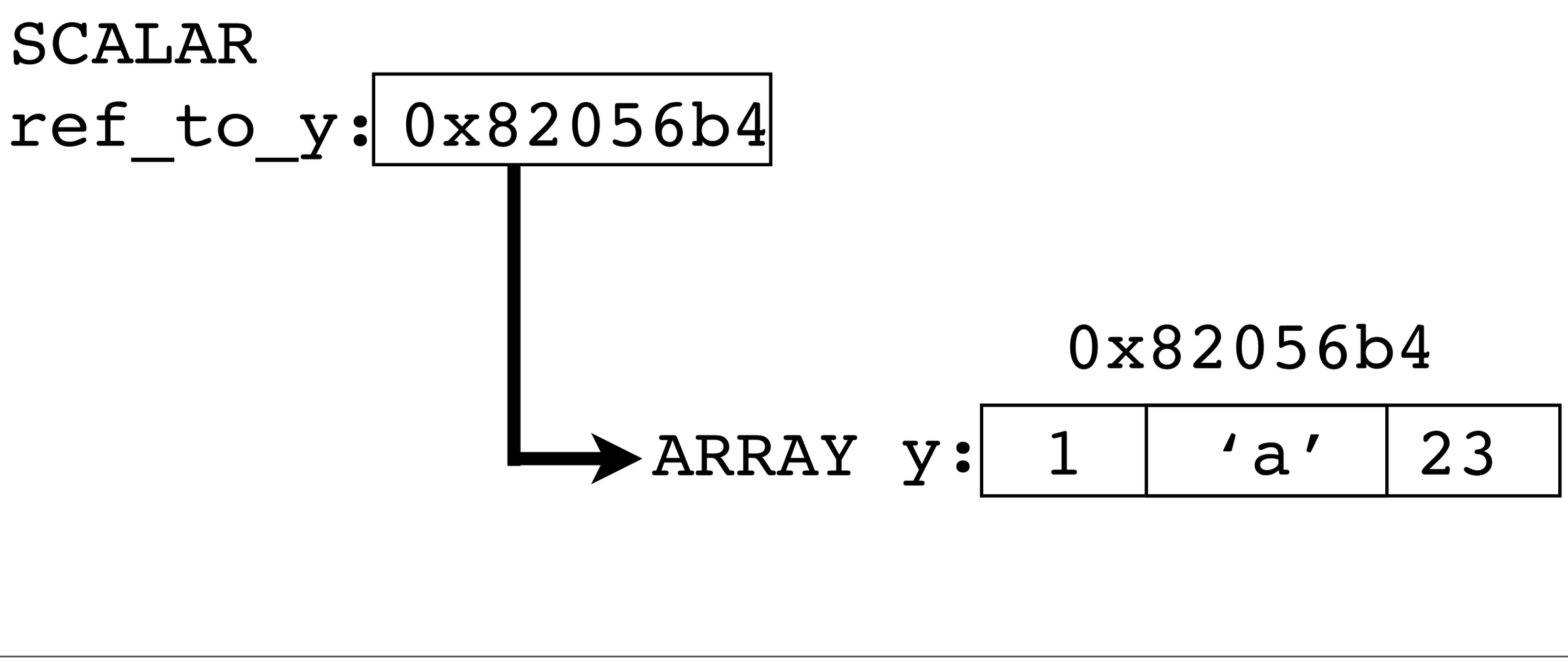
So, what is a reference?

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 some data is stored.

Making a reference to an array

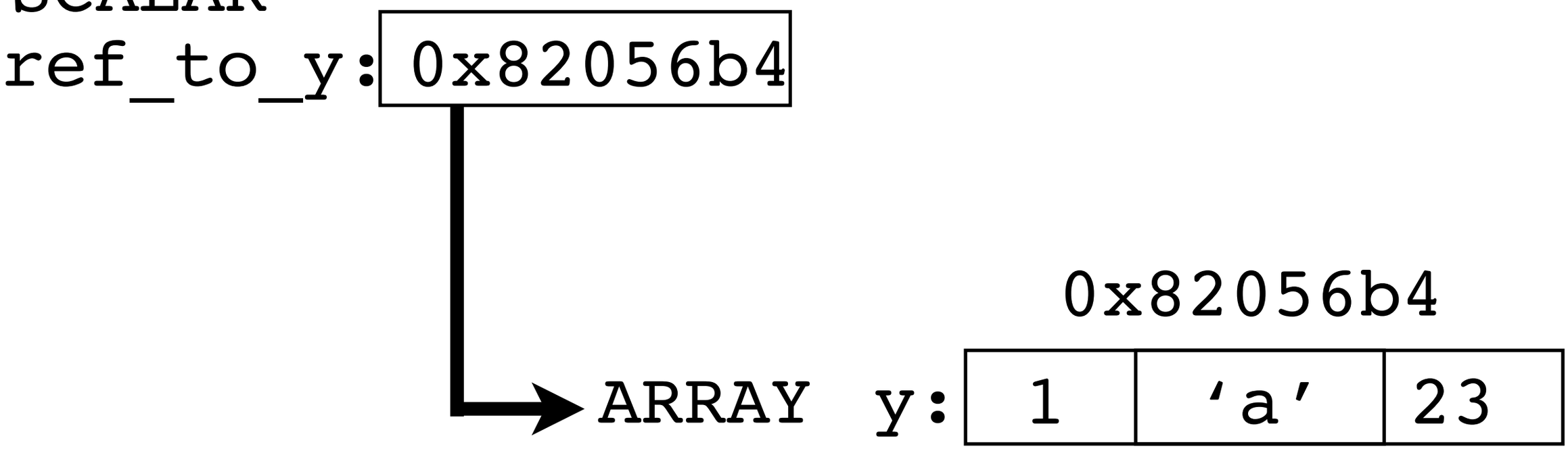
We can create a reference to named variable @y this way:



Printing a reference

If we try to print out $ref_to_y, we see the raw memory address:

```
print $ref_to_y,"\n";
ARRAY(0x82056b4)
```



To see the **contents** of what $ref_to_y **points to**, we have to **dereference** it:

```
print join ' ',@{$ref_to_y};
1 a 23
```

You can create references to scalars, arrays and hashes

```
# create some references
$scalar_ref = \$count;
$array_ref  = \@array;
$hash_ref   = \%hash;
```

To dereference a reference, place the appropriate symbol ($, @, %) in front of the reference:

```
# dereference your references:
$count_copy  = ${$scalar_ref};
@array_copy  = @{$array_ref};
%hash_copy   = %{$hash_ref};
```

References are pointers

A reference is a pointer to the data. It isn't a copy of the data.

When you make a reference to a variable, you have only created another way to get at the data.

There is still only one copy of the data.

```
@y                    = (1,'a',23);
$ref_to_y             = \@y;
print join ' ',@{$ref_to_y};
```
1 a 23

```
push @{$ref_to_y},'new1','new2';

print join ' ',@y;
```
1 a 23 new1 new2

This is in contrast to doing a direct copy from one variable to another, which creates a new data structure in a new memory location.

```
@y    = (1,'a',23);
@z    = @y;
push @y,'new1','new2';

print join ' ',@y;
```
1 a 23 new1 new2

```
print join ' ',@z;
```
1 a 23

If you have a reference to an array or a hash, you can access any element.

```
$value    = $y[2];
```

directly access the 3rd element in @y

```
$value    = ${$ref_to_y}[2];
```

dereference the reference, then access the 3rd element in @y

```
${$ref_to_y}[2] = 'new';
print join ' ',@y;
```
1 a new

change the value of the 3rd element in @y

```
%z          = (‘dog’     => 'animal',
               ‘potato’ => 'vegetable',
               ‘quartz’ => 'mineral',
               ‘tomato’ => 'vegetable');

$ref_to_z = \%z;

$value    = $z{‘dog’};
```

directly access the value associated with the key ‘dog’ in the hash %z

```
$value    = ${$ref_to_z}{‘dog’};
```

dereference the reference, then get the value associated with the key ‘dog’ in the hash %z

```
${$ref_to_z}{‘tomato’} = 'fruit';
print join ' ', values %z;
```

animal vegetable mineral fruit

change the value associated with the key ‘tomato’ in the hash %z

Anonymous Hashes and Arrays

You will not usually make references to existing variables. Instead you will create anonymous hashes and arrays. These have a memory location, but no symbol or name, i.e. you can't write @my_data. The reference is the only way to address them.

To create an anonymous array use the form:

```
$ref_to_arry = ['item1','item2'...]
```

To create an anonymous hash, use the form:

```
$ref_to_hash =
{key1=>'value1',key2=>'value2',...}
```

Remember [ ] goes with arrays `$a[0]` etc and { } goes with hashes `$hash{$key}` etc

```
$y_gene_families = ['DAZ', 'TSPY', 'RBMY', 'CDY1',
'CDY2' ];

$y_gene_family_counts = { 'DAZ'  => 4,
                          'TSPY' => 20,
                          'RBMY' => 10,
                          'CDY2' => 2 };
```

```
$third_item_of_arry = $y_gene_families->[2];
$daz_count          = $y_gene_family_counts->{DAZ};
```

$y_gene_families gets (i.e. is assigned) a reference to an array, and $y_gene_family_counts gets a reference to a hash.

Multidimensional Data: Making a Hash of Hashes

The beauty of anonymous arrays and hashes is that you can nest them:

```
my %y_gene_data = ( ‘DAZ’  => {‘family_size’ => 4,
          ‘description’ => 'deleted in azoospermia' },
                  ‘TSPY’ => {‘family_size’ => 20,
          ‘description’ => 'testis specific protein Y-
linked' },
                  ‘RBMY’ => {‘family_size’ => 10,
          ‘description’ => 'RNA-binding motif Y'},
                    ‘CDY2’ => {‘family_size’ => 2,
          ‘description’ => 'chromodomain protein, Y-linked' }
                  );

# what is the size of the RBMY family?
my $size = $y_gene_data{‘RBMY’}{‘family_size’};

# what is the description of TSPY?
my $desc = $y_gene_data{‘TSPY’}{‘description’};
```

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]
                );
my $cell_1_0  = $spotarray[1][0];
print $cell_1_0;

0.113
```

Examining References

Inside a Perl script, the `ref` function tells you what kind of value a reference points to:

```
print ref($y_gene_data), "\n";
HASH

print ref($spotarray), "\n";
ARRAY

$x = 1;
print ref($x), "\n";
(empty string)
```

Examining complex data structures in the debugger

Inside the Perl debugger, the "x" command will pretty-print the contents of a complex reference:

```
DB<3> x $y_gene_data
0  HASH(0x8404bb0)
   'CDY2' => HASH(0x8404b80)
      'description' => 'chromodomain protein, Y-linked'
      'family_size' => 2
   'DAZ' => HASH(0x84047fc)
      'description' => 'deleted in azoospermia'
      'family_size' => 4
   'RBMY' => HASH(0x8404b50)
      'description' => 'RNA-binding motif Y'
      'family_size' => 10
   'TSPY' => HASH(0x8404b20)
      'description' => 'testis specific protein Y-linked'
      'family_size' => 20
```

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 %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
...
```