References and multidimensional data

Simon Prochnik,
with input from Dave Messina, Lincoln Stein, Steve Rozen

Why do we need references?

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

What if you want to work with several related pieces of information? How would you represent this data in perl?

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

Working with related data

To represent the table of data below, you could imagine working with 4 arrays (one for each column of data)

```
@gene, @seq, @organism, @expression
```

or perhaps 3 hashes (with a common key of the gene name)

```
%sequence, %organism, %expression
```

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

Representing tables of data in perl

Perl lets you work with multidimensional arrays and hashes very easily. You just string keys or indices together.

For data with named columns, hashes are the most natural way to work

```
my %gene;
$gene{HOXB2}{sequence} = 'ATCAGCAATATTT';
$gene{HOXB2}{org} = 'mouse';
$gene{HOXB2}{expr} = 45.33;
$gene{HDAC1}{sequence} = 'GAGCGGAGCCGGGC';
$gene{HDAC1}{org} = 'human';
$gene{HDAC1}{expr} = 8.91;
```

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

Two-dimensional arrays

You could also represent this table with a two-dimensional array. The first index will be the row number, the second index will be the column number (both starting with 0). It would look like this

```
my @data;
$data[0][0] = 'HOXB2';
$data[0][1] = 'ATCAGCAATATTT';
$data[0][2] = 'mouse';
$data[0][3] = 45.33;
$data[1][0] = 'HDAC1';
$data[1][1] =  'GAGCGGAGCCGCGG';
$data[1][2] = 'human';
$data[1][3] = '8.91';
```

Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

More complex data structures

You can use more dimensions

```
$data[0][3][45] = 'human';
$expression{human}{BRCA1}{liver} = 45.98;
```

and you can mix and match hashes and arrays

```
$assay{HDAC1}[0]{human}{liver}[3] = 62.95;
```

How does perl store two-dimensional data?

You can only store one item of data in a scalar or an element of a hash or an element of array. So to make a two-dimensional array, perl stores a **reference** to an array in an element of the first array.

The debugger can help you explore and understand this

```
  DB<100>$data[0][0] = 'HOXB2'

  DB<101> p $data[0][0]
HOXB2
  DB<102> p $data[0]
ARRAY(0x7fd02a245490)
```

This is how perl displays a **reference** to an array

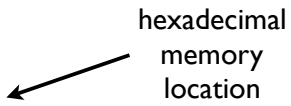
Gene	Sequence	Organism	Expression
HOXB2	ATCAGCAATATACAATTATAAAGG CCTAAATTTAAAA	mouse	45.33
HDAC1	GAGCGGAGCCGCGGGCGGGAG GGCGGACGGAC	human	8.91

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.



0x84048ec

`$x=1;` *really means* `SCALAR x:` | 1 |

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

really means

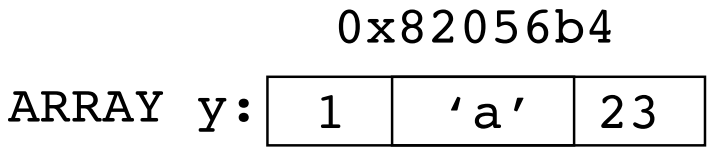
0x82056b4

`ARRAY y:` | 1 | 'a' | 23 |

How is a reference different from a variable?

A variable is a labeled memory address.

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



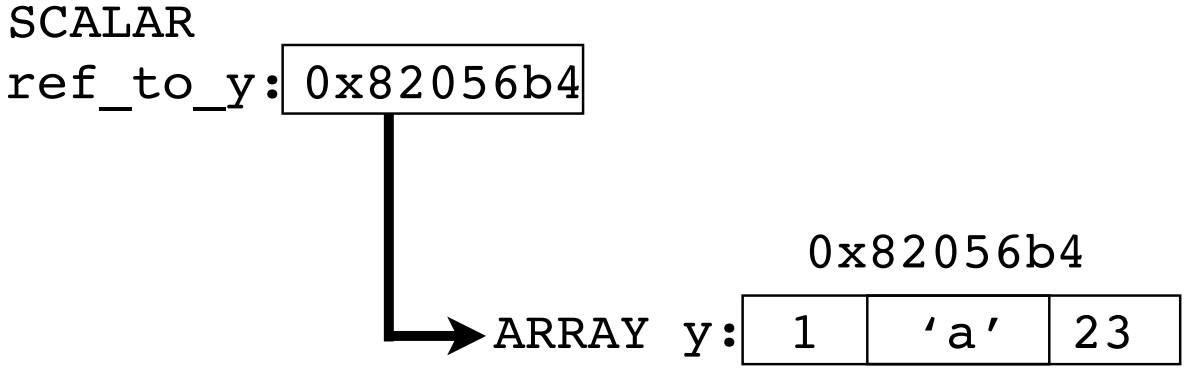
In contrast, a **reference** contains the memory address where some data is stored; it does not contain the data itself.

Creating references yourself

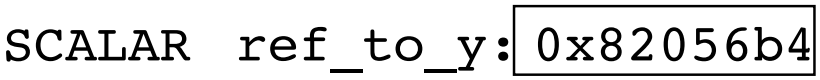
Because a reference is a scalar, it is useful to create a reference to a more complicated data structure if you want to pass it to a subroutine.

We can create a reference to named variable @y. We use a backslash character '\' to say 'a reference to' like this:

```
my $ref_to_y = \@y;
```



What's stored in a reference



If we print out $ref_to_y, we see the raw hexadecimal memory address where the array @y is stored:

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


Dereferencing = the opposite of making a reference

You can create references to scalars, arrays and hashes

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

To dereference a reference, place the appropriate symbol ($ for scalar references, @ for array references, % for hash references) in front of the reference.

This makes a new scalar, array or hash that is a copy of the one the reference pointed to.

```
# dereference your references:
my $count  = ${$scalar_ref};
my @new_array  = @{$array_ref};
my %new_hash   = %{$hash_ref};
```

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.

```
my @y                 = (1,'a',23);
my $ref_to_y          = \@y;
print join ' ',@{$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 assigning a variable to be equal to another, which creates a new data structure in a new memory location.

```
my @y   = (1,'a',23);
my @z   = @y;              # copy @y into @z
push @y,'new1','new2';     # add to @y only

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.

```
my $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


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

my $ref_to_z = \%z;

my $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:

```
my $array_ref = [‘item1','item2'...];
```

To create an anonymous hash, use the form:

```
my $hash_ref = {key1=>’value1',key2=>'value2',...};
```



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

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

my $third_item_of_arry = $y_gene_families->[2];
my $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.

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'};
```


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]
                 );
print $spotarray[1][0];
0.113

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:

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

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

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


Examining complex data structures in the debugger

Inside the Perl debugger, the "x" command will print the contents of a complex reference nicely formatted like so:

```
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      tcaaatacagcaccagctccttttttttatagttcgaattaatgtccaact...
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
use warnings;
use strict;

# tabulate nucleotide counts, store into %sequences

my %seqs; # initialize hash

while (my $line = <>) { # shortcut , reads a line from a file
  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
...
```


When do you really need references? For passing complex data to subroutines

```
#!/usr/bin/perl
use warnings;
use strict;

my @scores = (1,2,3,4);
my @students = qw(bob karen emily john);

# you can't use this next form. Why not?
#my $smartest_student=see_who_is_best(@scores,@students); #WRONG

my $smartest = see_who_is_best(\@scores,\@students);
print "$smartest\n";

sub see_who_is_best {
  # this next line doesn't work
  # my (@scores,@students) = @_; # WRONG!
  # you have to use this
  # can you see why?
   my ($score_ref,$student_ref) = @_;
  my @scores = @{$score_ref};
  my @students = @{$student_ref};
   # some more code goes here
   #
   #
}
```