# References and Extended Data Structures

## Aims

- Gain a grasp of references
- Multi-dimensional data structures

## 2 Dimensional Data

- Often we are going to have rows/sets of data, rather than just a single set
- The current variables we have only cope with one set
  - single value – scalar
  - list – array
  - named data - hash
- How do we deal with a table of data for example?

## Data Structures

| Name | Age | Height | Hair | Eyes |
|---|---|---|---|---|
| Andy | 36 | 175 | Brown | Blue |
| James | 42 | 180 | Black | Brown |
| Cameron | 5 | 120 | Blonde | Blue |

- Here is a table of information
- You are limited to AN array or A hash per row with what we currently know

## References

- However, we can use references to help with this problem
- A reference is a pointer to the memory location of the data
- It is stored in a scalar, and we access by dereferencing

## References - Scalar

- bin/01-references.pl

```
my $original_scalar = 'original';
my $copy_of_original_scalar = $original_scalar;
say $original_scalar . ' ' . \$original_scalar;
say $copy_of_original_scalar . ' ' . \
$copy_of_original_scalar;
```

- Both of these pieces of information use different memory locations, even though one is a copy.
- \ tells perl to give us the memory reference

## References - Scalar

```
$copy_of_original_scalar = 'new';
say $copy_of_original_scalar . ' ' . \
$copy_of_original_scalar;
```

- What you see here is that we have overwritten the value in the copy, but the memory location is the same

## References - Scalar

```
my $reference_to_original_scalar = \
$original_scalar;
say $reference_to_original_scalar . ' ' . \
$original_scalar;
```

- We can assign the reference to another variable

## References - Scalar

- To get it back:

```
say ${ $reference_to_original_scalar };
```

- By putting it into ${ } we tell perl to dereference the reference to a scalar

## References - Scalar

- Now a bit of Magic:

```
$original_scalar = 'another string';
```

- What do you expect to see if we get the dereferenced value?

```
say $original_scalar . ' ' . $
{ $reference_to_original_scalar };
```

## References - Scalar

- And what about changing the value of the dereferenced reference?

```
${ $reference_to_original_scalar } = 'original
again';
say $original_scalar . ' ' . $
{ $reference_to_original_scalar }
```

## References - Scalar

- Why faff with references to scalars?
  - Can reduce memory (do you want two copies of War and Peace?)
- What we see is the principal. We can take references of any data structure.

## References - Array

```
my @original_array = 1..200;
my @copy_of_original_array = @original_array;
say \@original_array;
say \@copy_of_original_array;
```

- Here we are, doing the copy of an array again, and seeing the memory locations of the two copies

## References - Array

- Lets take a reference of the original array

```
my $array_ref = \@original_array;
```

- Note: We are storing the reference in a scalar, not an array!

```
say $array_ref . ' ' . \@original_array;
```

- Same reference to place in memory
- We can get an element with the - > operator

```
say $array_ref->[0];
```

## References - Array

- We can do similar magic:

```
splice @copy_of_original_array, 100,10;
say scalar @copy_of_original_array;
say scalar @original_array;
say scalar @{ $array_ref };
```

- Whats going to be the results here?

(splice is removing 10 elements, starting at index 100, scalar returns the number of elements)

## References - Array

```
push @original_array, @original_array;
say scalar @copy_of_original_array;
say scalar @original_array;
say scalar @{ $array_ref };
```

- How about here?

(This is a way of doubling an array)

## References - Array

```
pop @{ $array_ref };
say scalar @copy_of_original_array;
say scalar @original_array;
say scalar @{ $array_ref };
```

- And here?

(pop removes the last element)

## References - Array

- Basically, we have only altered the array stored in memory
- You can do anything you want with an arrayref that you can do with an array
- You just need to dereference with
  - @{ $array_ref } for the full array
  - $array_ref->[index] for individual element

## References - Hash

```
my %original_hash = (
  band => 'Queen',
  'lead vocals' => 'Freddie Mercury',
  'lead guitar' => 'Brian May',
  'bass guitar' => 'John Deacon',
  'drums' => 'Roger Taylor',
);
```

- Here I have a hash, anyone want to suggest how I take a reference?

## References - Hash

```
my %copy_of_original_hash = %original_hash;
say \%original_hash;
say \%copy_of_original_hash;
my $hash_ref = \%original_hash;
say $hash_ref . ' ' . \%original_hash; # same reference to place in memory
```

- Again - same reference to place in memory

## References - Hash

```
$copy_of_original_hash{band} = 'Queen + Paul Rodgers';
$copy_of_original_hash{'lead vocals'} = 'Paul Rodgers';
delete $copy_of_original_hash{'bass guitar'};
say %original_hash;
say %copy_of_original_hash;
```

- Here, we have modified the copy band

## Instantiate Anonymity

- References are often referred to as anonymous, and can be directly created

```
my $anon_array_ref = [ ];
say $anon_array_ref;
my $anon_hash_ref = { };
say $anon_hash_ref;
```

- This will be very important for later work

## Multi-Dimensional Data

- So an array or hash can be referenced and that can be stored in a scalar 'box'
- As arrays or hashes contain scalar boxes, we can put references in those boxes as well

*bin/02-multi_dimensional_data_structures.pl*

## Multi-Dimensional Data

```
my %amino_acid_3_letter_codes = (
  Alanine => 'Ala',
  Arginine => 'Arg', ...
);
my %amino_acid_bases = (
  TTT => 'Phenylalanine',
  TCT => 'Serine', ...
);
```

## Multi-Dimensional Data

- The first thing to note is that there is lots of repeated data inside the 2nd hash
- But, we could change that around, so that the codons are in an array, which match the name, using arrayrefs

## Multi-Dimensional Data

```
my %codes_per_amino_acid = (
  Alanine => [ 'GCA', 'GCC', 'GCG','GCT],
  Arginine => [ 'AGA', 'AGG', 'CGA', 'CGC', 'CGG', 'CGT' ], ...
);
```

- we are instantiating an anonymous array into the scalar that would be at $codes_per_amino_acid{Alanine}

## Multi-Dimensional Data

- Could we get the two hashes down to one?

  Yes
- Because hashes can also be referenced.

```
my %genetic_code_data = (
  'Alanine' => {
    '3_letter_code' => 'Ala',
    'codons' => [ 'GCA', 'GCC', 'GCG', 'GCT' ]
  }, ...
);
```

## Multi-Dimensional Data

- How do we access the data in a multidimensional structure?

```
say $genetic_code_data{Methionine}{'3_letter_code'};
say $genetic_code_data{Methionine}{codons}[0];
```

- We chain together the keys/indexes that would be used in each structure to get the final value.

## Multi-Dimensional Data

- We can just have arrays of course

```
my @multiplication_table = (
  [ qw{ 0 0 0 0 0 0 0 0 0 0 0 } ],
  [ 0..10 ],
  [ qw{ 0 2 4 6 8 10 12 14 16 18 20 } ],
  [ qw{ 0 3 6 9 12 15 18 21 24 27 30 } ],
);
say $multiplication_table[3][5];
```

## Summary

- By using references, we can store multidimensional data (tables)
- A scalar can contain a reference to a memory location of an array or hash
- Each of these items can also contain memory references to further arrays and hashes
- We can retrieve the data using a combination of indices and keys, and the ->