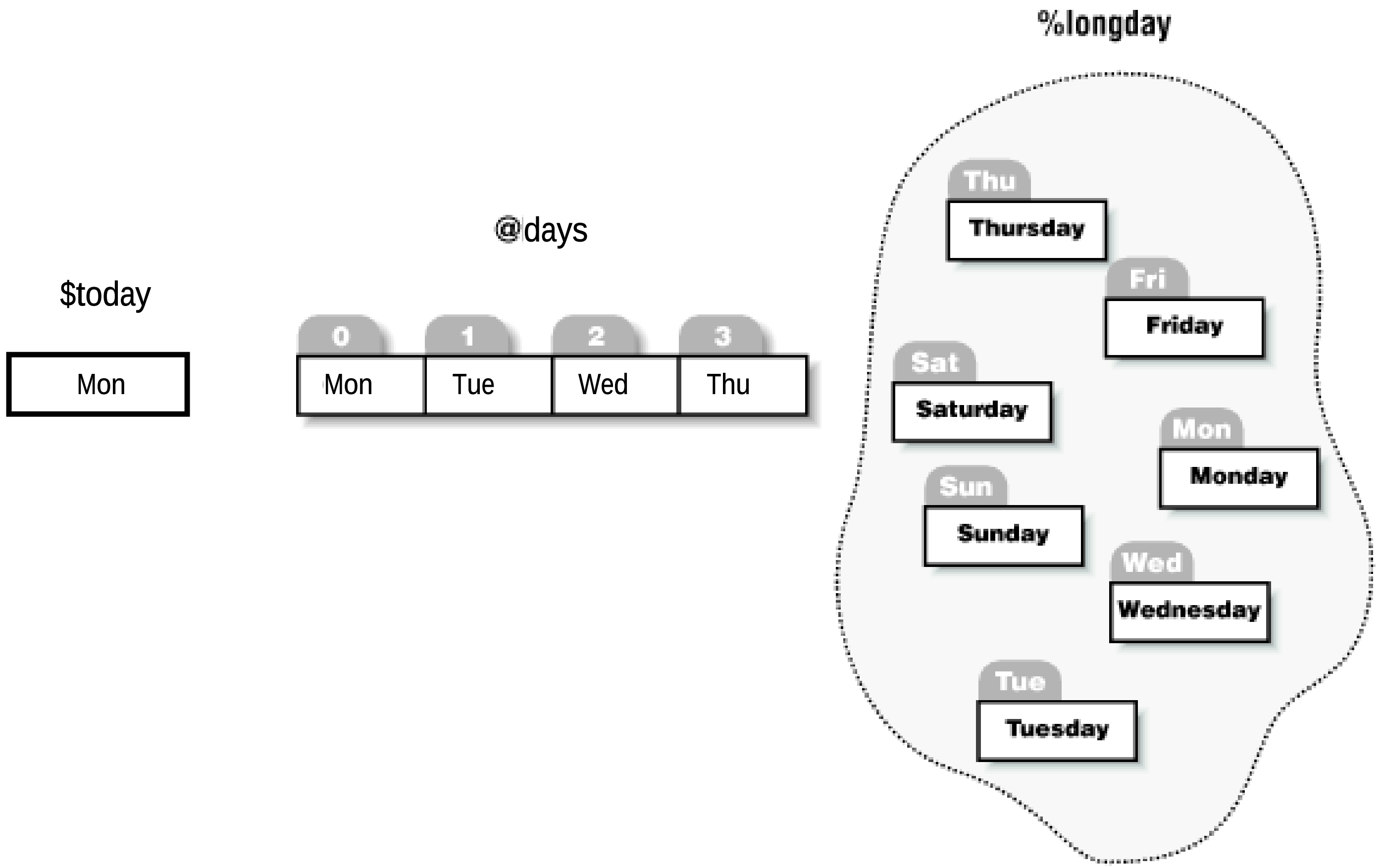
Lecture 2

Torgeir R. Hvidsten
Assistant professor in Bioinformatics
Umeå Plant Science Center (UPSC)
Computational Life Science Centre (CLiC)

This lecture

- Go through Lab 1
- Introduction to Perl 2
  - hashes
  - data structures
  - subroutines and modules
  - references

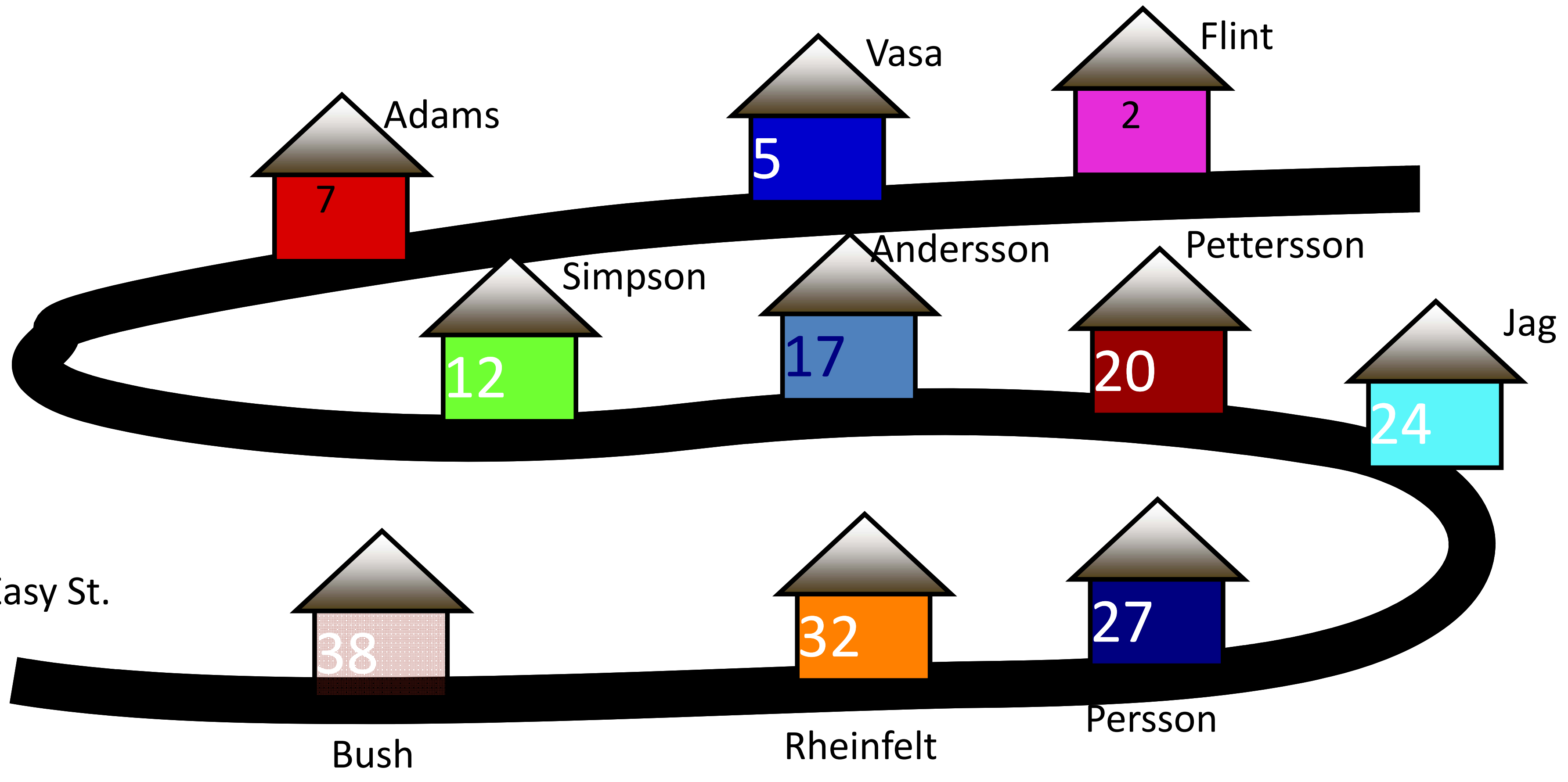
The three fundamental datatypes in Perl



Scalar Array Hash

- The *sigills* $,@,% must always be used.
- You can use different datatypes with the same name in the same program.

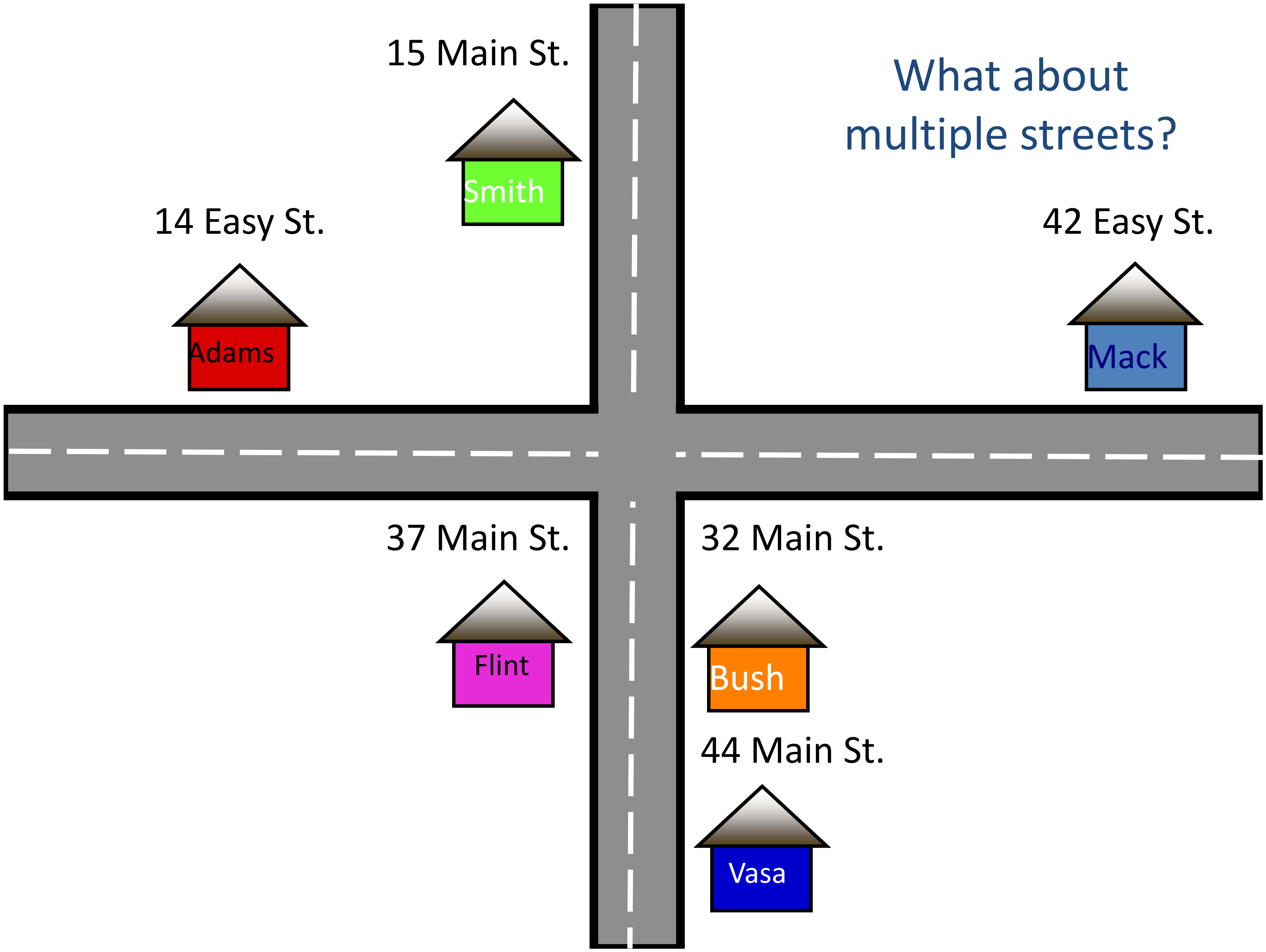
Arrays to look up addresses on the same street

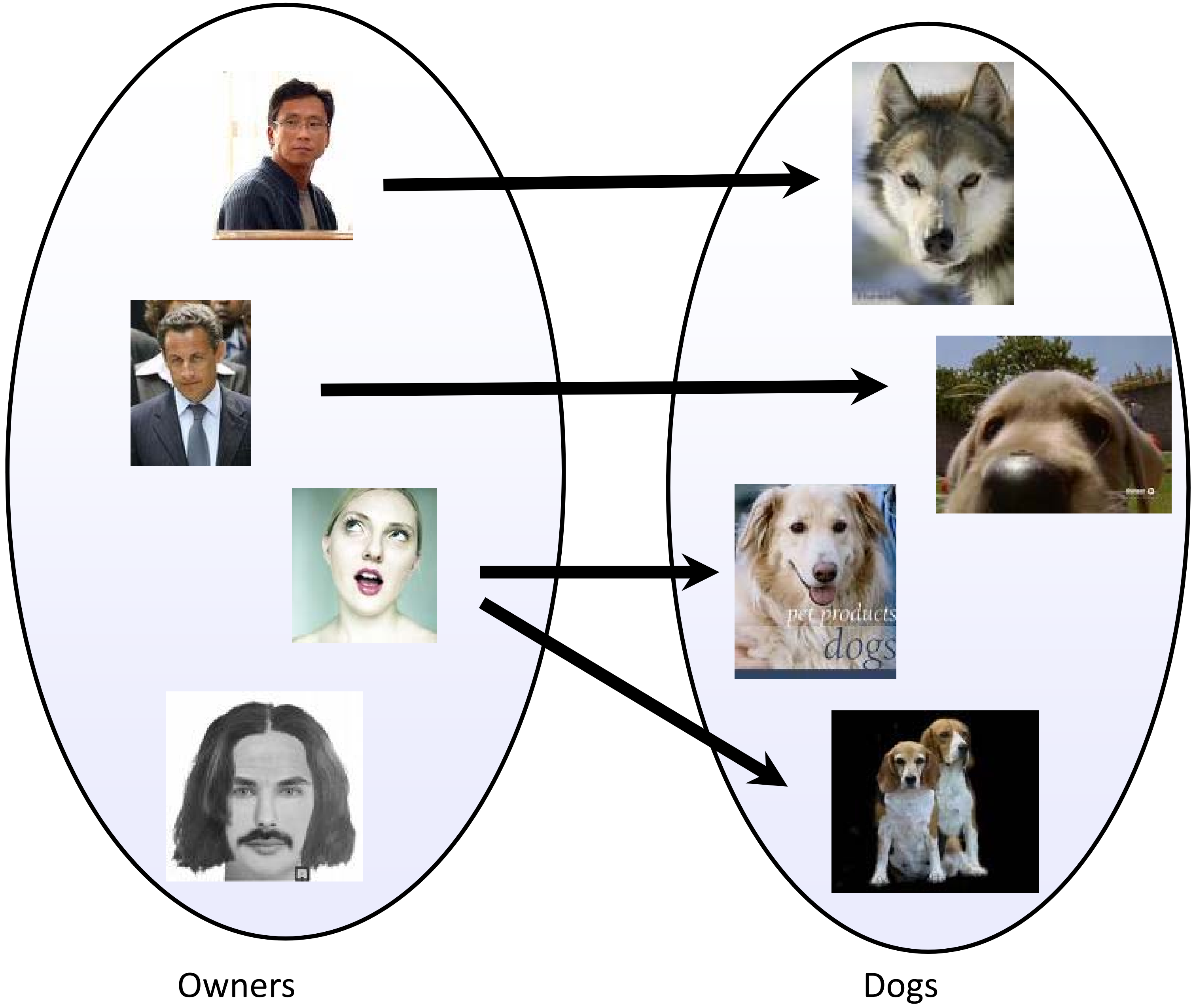


@owners_on_easy:

0	1	2	3	4	5	6	7	
undef	undef	"Flint"	undef	undef	"Vasa"	undef	"Adams"	...

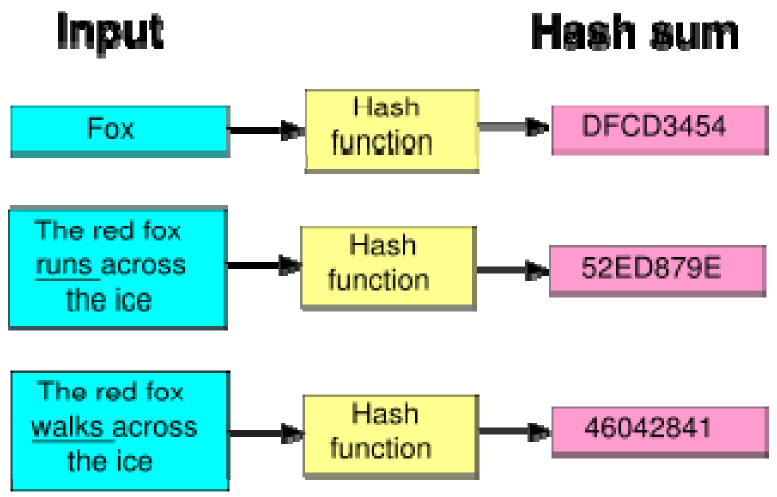
What about multiple streets?




pet products
dogs
Owners
Dogs

Hashing

- Hash algorithms convert strings of any length into reasonably small numbers; these numbers may be used to index an array.
- The same string must always give the same hash, but different strings can give the same hash. This is called a *collision* and is handled by Perl in a way that is invisible to you.
- Well-mixed hash-functions don't preserve the similarity of their input. *Hash functions do not sort their input.*



Perl hashes

- *Hashes* hold multiple, unordered pairs of keys and values. Each is a scalar.
- Hashes are written with a leading %, like: %favorite_color
- Hashes can be initialized by lists of keys and values using the "Big Arrow" => :

```
my %favorite_color = (dave => 'green', jim => 'blue', fred => 'red');
```

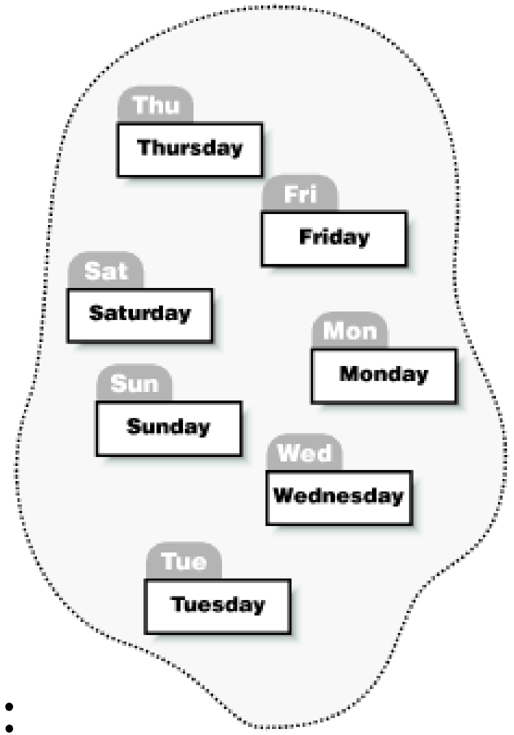
- Hashes are indexed by their keys. Notice the curly brackets!

```
my %fc = (dave => 'green', jim => 'blue', fred => 'red');
print "Daves favorite color is $fc{dave}\n";
Daves favorite color is green
```

- *Each key in a hash must be unique!* Reuse of a key causes reassignment:

```
my %fc = (dave => 'green', dave => 'blue');
print "Daves fave color is $fc{dave}\n";
Daves favorite color is blue
```

Accessing Hashes and Hash Slices



%days

- You access hashes by key in curly brackets:

```
my $fave = “Fri”;
my ($today,$tomorrow,$favorite) = ($days{Mon},$days{Tue},$days{$fave});
print "$today $tomorrow $favorite\n";
Monday Tuesday Friday
```

- You can access a slice of a hash by a list:

```
my ($today,$tomorrow,$favorite) = @days{"Mon","Tue",$fave};
print "$today $tomorrow $favorite\n";
Monday Tuesday Friday
```

Iterating over hashes

- The keys function is the most common way to iterate over a hash:

```
my %fc = (dave => 'green', jim => 'blue', fred => 'red');
foreach (keys %fc) {
	print "$_\'s favorite color is $fc{$_}\n";
}
```

```
jim's favorite color is blue
dave's favorite color is green
fred's favorite color is red
```

- The each function is less common, returning key-value pairs

```
while (my ($key, $value) = each %fc) {
	print "$key\'s favorite color is $value\n";
}
```

Iterating over hashes

- Sorting by keys

```
my %fc = (dave => 'green', jim => 'blue', fred => 'red');
foreach (sort keys %fc) {
        print "$_\'s favorite color is $fc{$_}\n";
}
```

```
dave's favorite color is green
fred's favorite color is red
jim's favorite color is blue
```

- Sorting by value:

```
my %fc = (dave => 'green', jim => 'blue', fred => 'red');
foreach (sort {$fc{$a} cmp $fc{$b}} keys %fc) {
        print "$_\'s favorite color is $fc{$_}\n";
}
```

```
jim's favorite color is blue
dave's favorite color is green
fred's favorite color is red
```

Existence and definedness

Use exists to check for the presence of a key in a hash, not defined

```
my %age;
$age{"Toddler"} = 3;
$age{"Unborn"} = 0;
$age{"Phantasm"} = undef;

foreach my $thing ("Toddler", "Unborn", "Phantasm", "Relic") {
	print "$thing: ";
	print "Exists " if exists $age{$thing};
	print "Defined " if defined $age{$thing};
	print "\n";
}
```

```
Toddler: Exists Defined
Unborn: Exists Defined
Phantasm: Exists
Relic:
```

Hashes as sets

- The uniqueness of keys in hashes make hashes useful models of sets, and you can easily do set operations on hashes:

```
my %hash1 = (a => 1, b => 1, d => 1);
my %hash2 = (a => 1, c => 1, d => 1);

my @common = ();
foreach (keys %hash1) {
  push @common, $_ if exists $hash2{$_};
}
print "@common\n";
a d
```

- Write pseudo-code that solves the same problem with arrays!

Nested data structures

- Scalars, arrays and hashes are not enough! We want to nest data structures to create e.g. tables (arrays of arrays).
- Perl cannot do arrays of arrays, however, it can do arrays of references to arrays:
  ```
  my @players = ("Maldini","Giggs","Inzaghi");
  my $ref  = \@players;
  ```
- References are scalars that point to an address in memory
  ```
  print "$ref\n";
  ARRAY(0x23affd4)
  ```
- Accessing values from references is called *dereferencing*.
  ```
  print "$ref->[2]\n";
  Inzaghi
  print "@$ref\n";
  Maldini Giggs Inzaghi
  ```

References

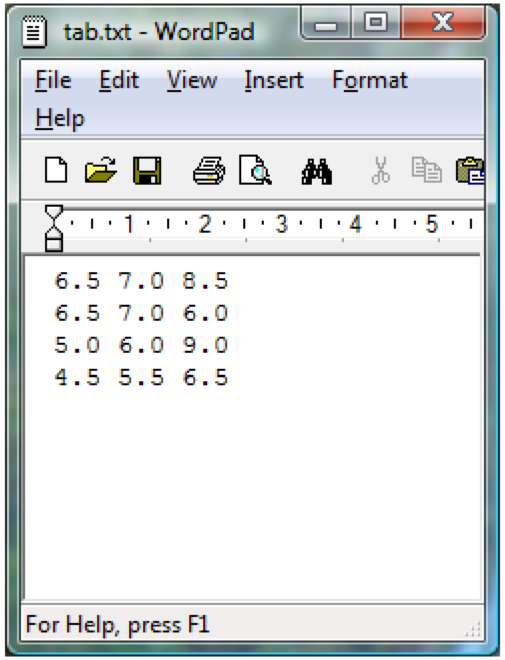
- This:
  ```
  my @players = ("Maldini","Giggs","Inzaghi");
  my $ref = \@players;
  ```
  is equivalent to this:
  ```
  my $ref = ["Maldini","Giggs","Inzaghi"];
  ```
- And this:
  ```
  my %players = (Maldini => 1, Giggs => 1, Inzaghi => 1);
  my $ref = \%players;
  ```
  is equivalent to this
  ```
  my $ref = {Maldini => 1, Giggs => 1, Inzaghi => 1};
  ```
- $ref is called an anonymous array or hash.

Reading a table from file

```
my @tab;

open (T, "tab.txt");
while (<T>) {
      chomp;
      my @row = split /\s/;
      push @tab, \@row;
}
close (T);

print "$tab[0]->[1]\n";
print "$tab[0][1]\n";
print "@{$tab[2]}\n";
7.0
7.0
5.0 6.0 9.0
```



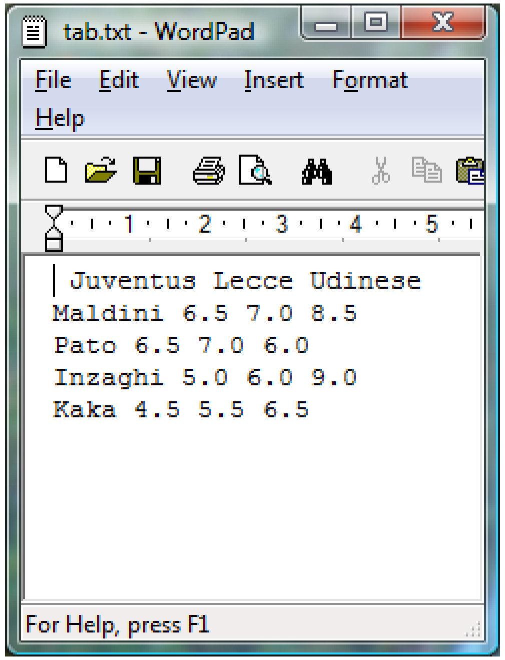
```
6.5 7.0 8.5
6.5 7.0 6.0
5.0 6.0 9.0
4.5 5.5 6.5
```

Reading a table from file stored as a hash of arrays

```
my %ratings;

open (T, "tab.txt");
my @teams = split /\s/, readline *T;
while (<T>) {
  chomp;
  my @row = split /\s/;
  my $player = shift @row;
  $ratings{$player} = \@row;
}
close (T);

print "$ratings{Maldini}->[1]\n";
print "$ratings{Maldini}[1]\n";
print "@{$ratings{Inzaghi}}\n";
7.0
7.0
5.0 6.0 9.0
```



```
 Juventus Lecce Udinese
Maldini 6.5 7.0 8.5
Pato 6.5 7.0 6.0
Inzaghi 5.0 6.0 9.0
Kaka 4.5 5.5 6.5
```

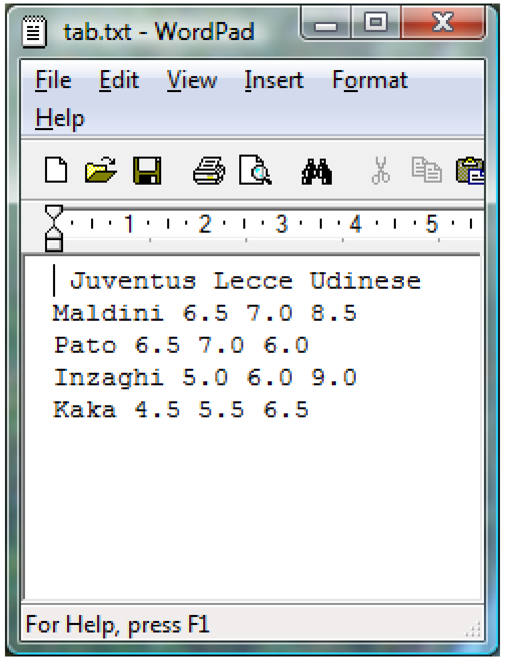
Reading a table from file stored as a hash of hashes

```
my %ratings;

open (T, "tab.txt");
my @teams = split /\s/, readline *T;
while (<T>) {
        chomp;
        my @row = split /\s/;
        my $player = $row[0];
        for (1..$#row) {
                $ratings{$player}{$teams[$_]} = $row[$_];
        }
}
close (T);

print "$ratings{Maldini}->{Juventus}\n";
print "$ratings{Maldini}{Juventus}\n";
print "Inzaghi\n";
foreach (keys %{$ratings{Inzaghi}}) {
        print " $_: $ratings{Inzaghi}{$_}\n";
}
```

```
6.5
6.5
Inzaghi
 Udinese: 9.0
 Juventus: 5.0
 Lecce: 6.0
```



```
 Juventus Lecce Udinese
Maldini 6.5 7.0 8.5
Pato 6.5 7.0 6.0
Inzaghi 5.0 6.0 9.0
Kaka 4.5 5.5 6.5
```

Syntax summary

- Scalers:

  $player

- Arrays:

  @players, Element: $players[1]

- Hashes:

  %players, Value: $players{Maldini}

Syntax summary

- Array of arrays:

  @{$players[1]}, Element: $players[1][5]
- Hash of hashes:

  %{$players{Maldini}}, Value: $players{Maldini}{Udinese}
- Hash of arrays:

  @{$players{Maldini}}, Element: $players{Maldini}[5]
- Array of hashes:

  %{$players[1]}, Value: $players[1]{Udinese}

Subroutines and modules

- Modularizing code makes programming easier
  - allows shorter and more easily maintainable code
  - allows reuse of code
- Subroutines are functions
- Modules are collections of subroutines

Subroutines

```
my $m1 = mean(1.2, 1.5, 1.7, 4.5, 6.7);
print "$m1\n";

my $m2 = mean(3.3, 1.8, 1.9, 4.5, 10);
print "$m2\n";

sub mean {

    my @vector = @_;

    my $sum = 0;
    foreach (@vector) {
        $sum += $_;
    }
    my $mean = $sum/@vector;

    return $mean;
}
```

```
3.12
4.3
```

- The default array @_ has a similar function and use as the default scalar $_, but for subroutines
- return returns a scalar or an array

Subroutine

Pass by value

```
my @vector = (1,4,3,8,9);

multiply_by_n(\@vector, 2);
print "@vector\n";

sub multiply_by_n {

    my @vector = @{$_[0]};
    my $n = $_[1];

    foreach (@vector) {
        $_ *= $n;
    }
}
```

```
1 4 3 8 9
```

Pass by reference

```
my @vector = (1,4,3,8,9);

multiply_by_n(\@vector, 2);
print "@vector\n";

sub multiply_by_n {

    my $vector = $_[0];
    my $n = $_[1];

    foreach (@$vector) {
        $_ *= $n;
    }
}
```

```
2 8 6 16 18
```

Modules

Module

(file name: Statistics.pm)

```
package Statistics;

sub mean {

	my @vector = @_;

	my $sum = 0;
	foreach (@vector) {
		$sum += $_;
	}
	my $mean = $sum/@vector;

	return $mean;
}

1;
```

Program

```
use strict;
use warnings;

use Statistics;

my $m = Statistics::mean(1.2, 1.5, 1.7, 4.5, 6.7);
print "$m\n";
3.12
```

Acknowledgements

- Several slides were taken or re-worked from David Ardell and Yannick Pouliot.