

Learning Perl Together

Ian Norton

Learning together?

Introductions!

- What's your name?
- Where have you come from?
- Which Perl Mongers group are you with?
- What's your Perl experience?
- What are you hoping to get today?



Ian Norton

Lancaster

shadow.cat

Introductions!

- What's your name?
- Where have you come from?
- Which Perl Mongers group are you with?
- What's your Perl experience?
- What are you hoping to get today?

N.B.

- Can't cover everything
- Keeping it simple
- No stupid questions
- White space is free, sprinkle it liberally!
- I am not an expert

What are we covering?

- Getting started
- Data types
- Making decisions
- Flow control
- Subroutines

Boilerplate

Start our script

```
use strict;  
use warnings;
```

Restrict unsafe
constructs

Enable warnings

Running our script

```
perl myscript.pl
```


Comments

- Comments start #

```
# This is a comment
```

```
my $thing = 12; # This is also a comment
```

My First Perl Script

```
use strict;  
use warnings;  
  
# Print out a message  
print("Hello workshop\n");
```

My First Perl Script

```
use strict;  
use warnings;  
  
# Print out a message  
print("Hello workshop\n");
```



New line

Data types

- Store data in memory
- scalars
- arrays
- hashes

Data types

- Declared with 'my'
- Type determined by first character

scalars

- Prefixed with \$
- Storage of a single value
- Numbers
- Strings
- References

scalars

- Prefixed with \$
- Storage of a single value
- Numbers
- Strings
- References – we'll come back to these

scalars

- Scalar declarations

```
my $text = "Hello workshop\n";  
my $number = 12;
```


My First Perl Script

```
use strict;  
use warnings;
```

```
# Print out a message  
print("Hello workshop\n");
```

My First Perl Script

```
use strict;  
use warnings;  
  
my $text = "Hello workshop\n";  
  
# Print out a message  
print("Hello workshop\n");
```

My First Perl Script

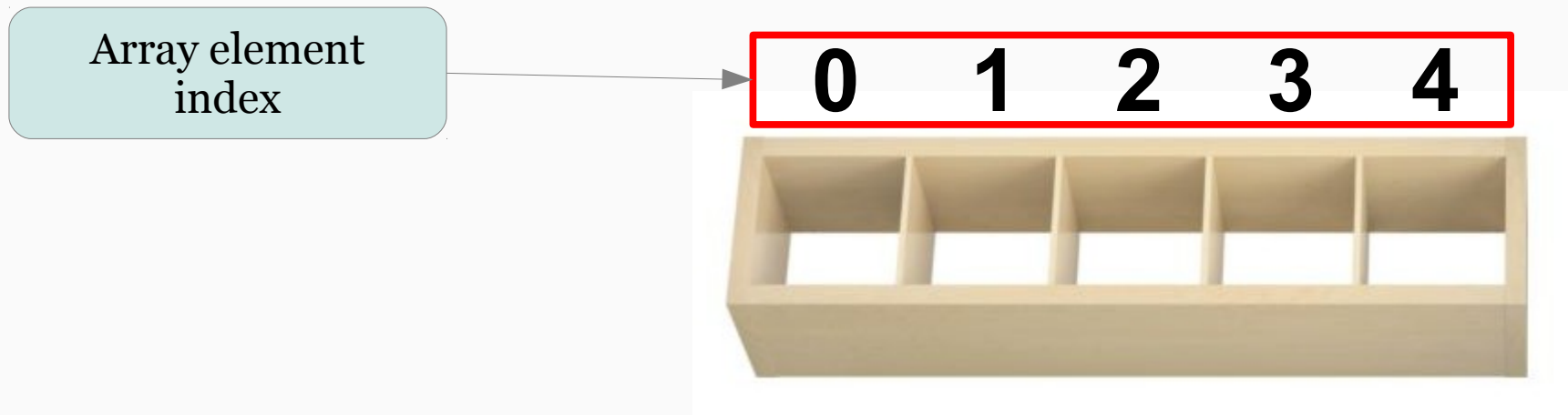
```
use strict;  
use warnings;  
  
my $text = "Hello workshop\n";  
  
# Print out a message  
print($text);
```

arrays

- Prefixed with @
- Storing multiple scalars
- Ordered

arrays

- Indexed by number



arrays

- Array declaration

```
my @shopping = ("Bread\n", "Butter\n", "Jam\n");
```

- Print can take an array

```
print(@shopping);
```

arrays

- Plural – the whole array

```
my @shopping = ("Bread\n", "Butter\n", "Jam\n");  
print(@shopping);
```

- Singular – an element of the array

```
$shopping[0] = "Brown bread";  
print($shopping[0]);
```

hashes

- Prefixed with %
- Indexed storage of scalars
- Unordered

hashes

- Indexed by a string
- Kind of like a card index



hashes

- Hash declaration

```
my %months = (  
    "January"    ,    31,  
    "March"      ,    31,  
    "April"      ,    30,  
);
```

hashes

- Hash declaration

```
my %months = (  
  "January" , 31,  
  "March"   , 31,  
  "April"   , 30,  
);
```

Keys

Values

hashes

- Hash declaration

```
my %months = (  
    "January"    ,    31,  
    "March"      ,    31,  
    "April"      ,    30,  
);
```

- This format is error prone

Odd number of elements in hash assignment

hashes

- Hash declaration

```
my %months = (  
    "January"    =>    31,  
    "March"      =>    31,  
    "April"      =>    30,  
);
```

hashes

- Hash declaration

```
my %months = (  
    "January" => 31,  
    "March"   => 31,  
    "April"   => 30,  
);
```



The fat comma

hashes

- Hash declaration

```
my %months = (  
    "January"    =>    31,  
    "March"      =>    31,  
    "April"      =>    30,  
);
```

hashes

- Hash declaration

```
my %months = (  
    January    =>    31,  
    March      =>    31,  
    April      =>    30,  
);
```


hashes

- Hash declaration

```
my %months = (  
    January    =>    31,  
    March      =>    31,  
    April      =>    30,  
);
```

- Quotation marks gone
- Assuming keys with no spaces
- Easier to read

hashes

- Plural – the whole hash

```
my %months = (  
    January    =>    31,  
    March      =>    31,  
    April      =>    30,  
);
```

- Singular – an element of the hash

```
$months{"May"} = 31;  
print($months{"May"});
```

Data types

Type	Prefix	Index	Element brackets	Contents ordered
Scalar	\$	-	-	-
Array	@	Numeric	[]	Yes
Hash	%	String	{ }	No

Excercise 1

- 10 minutes
- Write My First Perl Script and run it
- Switch to printing a scalar with own text
- Declare an array of fruit and print it
- Declare a hash with some of our introduction information in (name, country or town, PM group)
- Remember, this isn't a solo activity!

references

- Two shopping lists
- Supermarket
- Giftshop

references

- We can use arrays!

```
my @shopping1 = ("Bread", "Butter", "Jam");  
my @shopping2 = ("Paper", "Card", "Sellotape");
```

references

- We can use arrays!

```
my @shopping1 = ("Bread", "Butter", "Jam");  
my @shopping2 = ("Paper", "Card", "Sellotape");
```

- Scales really badly

references

We should put the arrays...

In a hash...!

references

- Hashes contain scalars
- Scalars contain:
 - Strings
 - Numbers
 - References

references

- Hashes contain scalars
- Scalars contain:
 - Strings
 - Numbers
 - References – The one we've come back to

references

We can only include an array in a hash
by reference

references

- Two ways to create a reference
- From an existing variable
- Anonymously

references

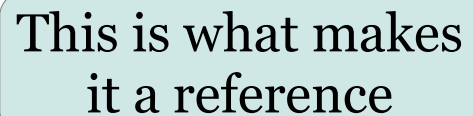
- Existing variable

```
my $array_ref = \@shopping;  
my $hash_ref  = \%months;
```

references

- Existing variable

```
my $array_ref = \@shopping;  
my $hash_ref  = \%months;
```



This is what makes
it a reference

references

- Anonymous array reference

```
my $array_ref = [ "Bread", "Butter", "Jam" ];
```

- Anonymous hash reference

```
my $hash_ref = {  
  January => 31,  
  March   => 31,  
  April   => 30,  
};
```

references

- Anonymous array reference

```
my $array_ref = [ "Bread", "Butter", "Jam" ];
```

- Anonymous hash reference

```
my $hash_ref = {  
    January => 31,  
    March   => 31,  
    April   => 30,  
};
```

- So what's different?

references


- Anonymous array reference

```
my $array_ref = [ "Bread", "Butter", "Jam" ];
```



- Anonymous hash reference

```
my $hash_ref = {  
  January => 31,  
  March   => 31,  
  April   => 30,  
};
```



- So what's different?

The brackets

references

Type	Anonymous reference creation	Element brackets
Array	[]	[]
Hash	{ }	{ }

references

- Making our shopping anonymous arrays

```
my $shopping1 = [ "Bread", "Butter", "Jam" ];  
my $shopping2 = [ "Paper", "Card", "Sellotape" ];
```

references

- Making our shopping anonymous arrays

```
my $shopping1 = [ "Bread", "Butter", "Jam" ];  
my $shopping2 = [ "Paper", "Card", "Sellotape" ];
```

- How we access an element now changes

```
my $element0 = $shopping1->[0];
```



This is new

references

- So long hand...

```
# Declare arrays shopping1 and shopping2
my @shopping1 = ("Bread", "Butter", "Jam");
my @shopping2 = ("Paper", "Card", "Tape");

# Declare hash shopping with keys supermarket & giftshop
# pointing to the arrays shopping1 and shopping2
my %shopping = (
    supermarket => \@shopping1,
    giftshop     => \@shopping2,
);
```

references

- Change to references for arrays

```
# Declare anonymous arrays shopping1 and shopping2
my $shopping1 = ["Bread", "Butter", "Jam"];
my $shopping2 = ["Paper", "Card", "Tape"];

# Declare hash shopping with keys supermarket & giftshop
# pointing to the arrays shopping1 and shopping2
my %shopping = (
    supermarket => $shopping1,
    giftshop    => $shopping2,
);
```

references

- Loose the named references

```
# Declare hash shopping with keys supermarket & giftshop
# pointing to the arrays shopping1 and shopping2
my %shopping = (
    supermarket => [ "Bread", "Butter", "Jam" ],
    giftshop     => [ "Paper", "Card", "Tape" ],
);
```

references

- We can get our array back

```
# Declare hash shopping with keys supermarket & giftshop
# pointing to the arrays shopping1 and shopping2
my %shopping = (
    supermarket => ["Bread", "Butter", "Jam"],
    giftshop     => ["Paper", "Card", "Tape"],
);

my @shopping1 = @{$shopping{"supermarket"}};
```


references

- Seem complex
- Really aren't anything magic
- Take things apart to help yourself

Exercise 2

- 10 minutes
- Discuss the MTA example

Making decisions

- Context
- Looping
- Flow control

context

- Perl cares about how we ask a question
- Not just what we ask
- void
- scalar
- list

context

- void

```
print("Hello\n");
```

- scalar

```
my $count = length("Hello");
```

- list

```
my @array = (1, 2, 3);  
my ($val1, $val2, $val3) = @array;  
my ($val1) = @array;
```

context

Functions can and will behave differently based on the context in which you call them

If in doubt, check the manual

Looping

- for
- foreach
- while
- until

for

- for(initialisation ; test ; operation)

```
for (my $i = 1 ; $i <= 10 ; $i++) {  
    print("$i\n");  
}
```

Short hand for
\$i = \$i + 1;

foreach

Range operator

- We can do the same slightly differently

```
foreach (1..10) {  
    print("$_\n");  
}
```

Perl default
scalar variable

foreach

- Providing a variable

```
foreach my $i (1..10) {  
    print("$i\n");  
}
```

- Using an array

```
foreach my $i (@shopping1) {  
    print("$i\n");  
}
```

while

- While \$i is less than or equal to 10.

```
my $i = 1;

while($i <= 10) {
    print("$i\n");
    $i++;
}
```

until

- Until \$i is equal to 10.

```
my $i = 1;

until($i == 10) {
    print("$i\n");
    $i++
}
```

Flow control

Comparisons

Numeric	String	Explanation
==	eq	Equals
!=	ne	Not equals
<	lt	Less than
>	gt	Greater than
<=	le	Less than or equal to
>=	ge	Greater than or equal to

if

- if

```
if($value == 0) {  
    print("Value is zero\n");  
}
```

if, elsif

- if, elsif

```
if($value == 0) {  
    print("Value is zero\n");  
}  
  
elsif($value == 1) {  
    print("Value is one\n");  
}
```

if, elsif, else

- if, elsif, else

```
if($value == 0) {  
    print("Value is zero\n");  
}  
  
elsif($value == 1) {  
    print("Value is one\n");  
}  
  
else {  
    print("Value was something else\n");  
}
```


unless

- Unless is a neat shortcut
- Same as writing:

```
if($value != 0) {  
    print("Value is not zero\n");  
}
```

- Syntactically nicer to read:

```
unless($value == 0) {  
    print("Value is not zero\n");  
}
```

Subroutines

- Modular & reusable code
- Writing a subroutine
- Arguments to a subroutine

Modular & reusable code

- Counting from 1 to N
- Do it more than once
- Copy and paste nastiness

Writing a subroutine

- Defined with the “sub” keyword

```
sub count {  
    for( my $i = 0 ; $i <= 10 ; $i++ ) {  
        print("$i\n");  
    }  
}
```

- Call our new subroutine

```
count( );
```

- But we can still only count to 10

Arguments to a subroutine

- Our call should be

```
count(20);
```

- Default array variable `@_`
- We unpack `@_` with `shift`

```
sub count {  
    my $max = shift;  
  
    for( my $i = 0 ; $i <= $max ; $i++ ) {  
        print("$i\n");  
    }  
}
```

Arguments to a subroutine

- If we have more arguments
- How do we remember which number?
- What if some are optional?
- Named arguments to the rescue.

Arguments to a subroutine

- Pass an anonymous hash reference

```
count({ max => 20 });
```

- So our subroutine now looks like this

```
sub count {  
    my $args = shift;  
    my $max = $args->{"max"};  
  
    for( my $i = 0 ; $i <= $max ; $i++ ) {  
        print("$i\n");  
    }  
}
```

- And this is why references are important

Further reading

- Manual pages
- Perl Best Practices
- The C Programming Language
- Beginning Perl
- Modern Perl
- Programming Perl

Questions

&

Answers

Thanks to you!

Don't forget to fill in the
conference survey!