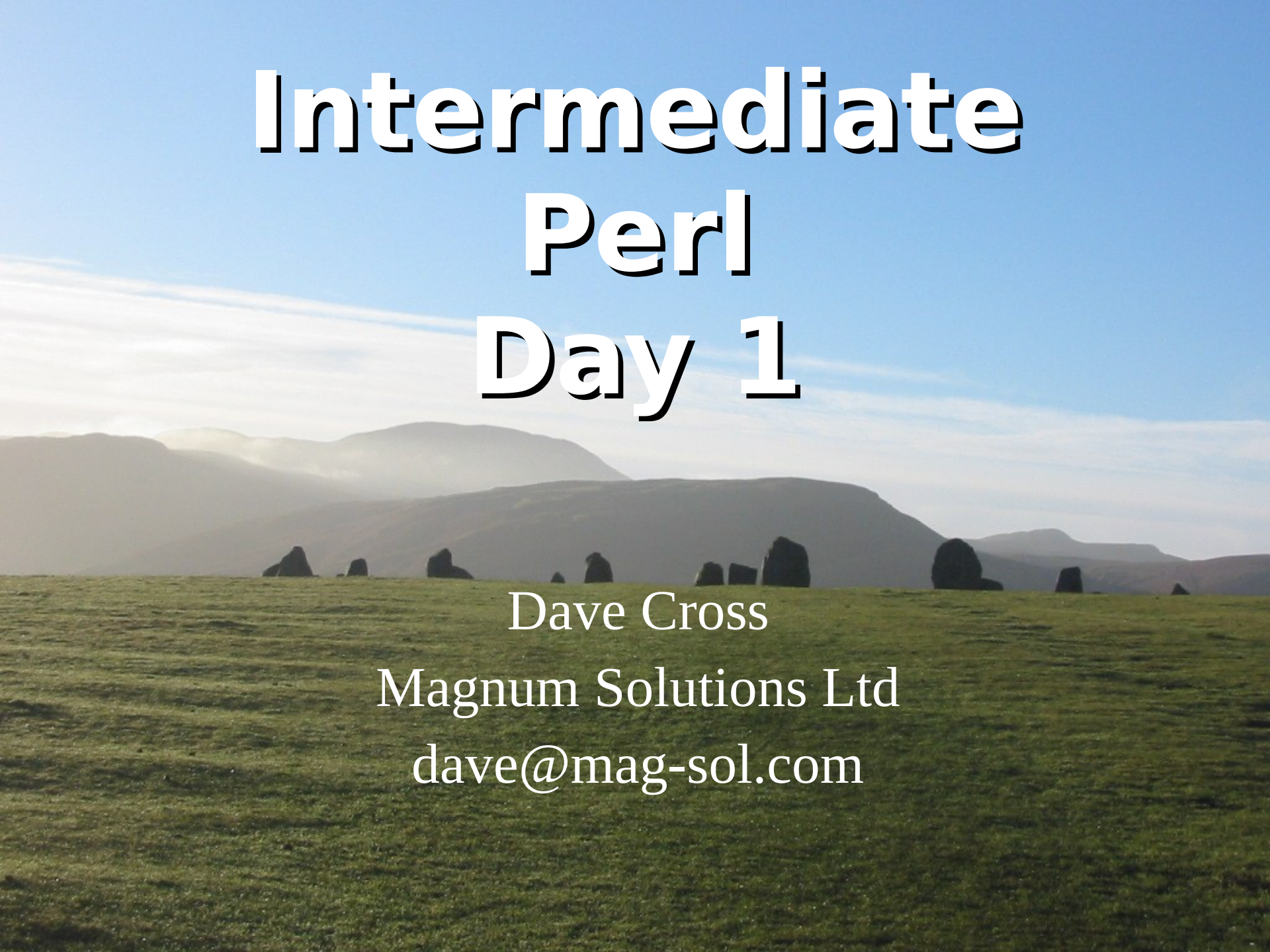


Intermediate Perl
Day 1

Dave Cross
Magnum Solutions Ltd
dave@mag-sol.com

What We Will Cover

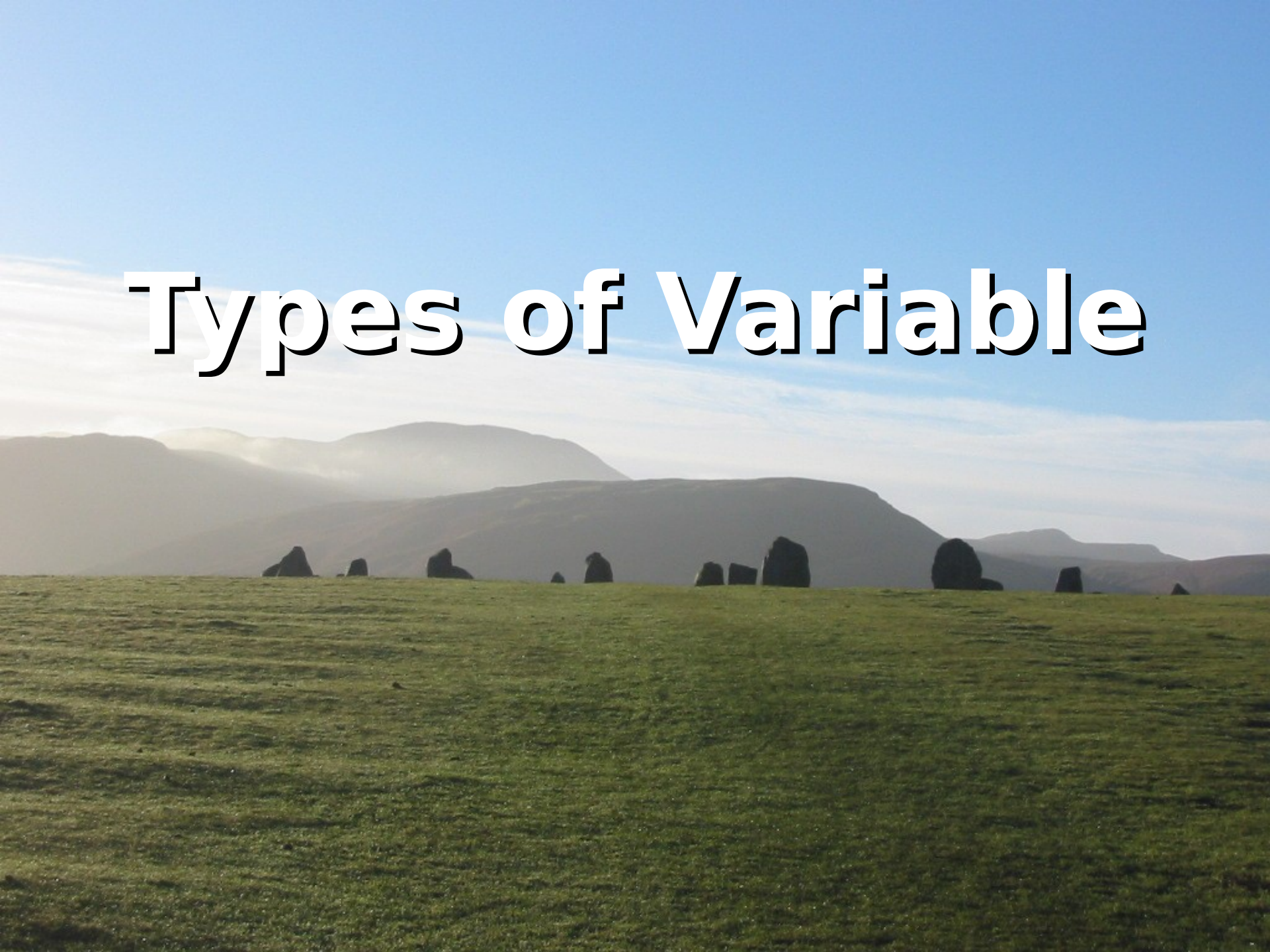
- Types of variable
- Strict and warnings
- References
- Sorting
- Reusable Code

Schedule

- 09:45 – Begin
- 11:15 – Coffee break (15 mins)
- 13:00 – Lunch (60 mins)
- 14:00 – Begin
- 15:30 – Coffee break (15 mins)
- 17:00 – End

Resources

- Slides available on-line
  - http://mag-sol.com/train/public/2011-02/ukuug
- Also see Slideshare
  - http://www.slideshare.net/davorg/slideshows
- Get Satisfaction
  - http://getsatisfaction.com/magnum


Types of Variable

Types of Variable

- Perl variables are of two types
- Important to know the difference
- Lexical variables are created with `my`
- Package variables are created by `our`
- Lexical variables are associated with a code block
- Package variables are associated with a package

Lexical Variables

- Created with `my`
- `my ($doctor, @timelords, %home_planets);`
- Live in a pad (associated with a block of code)
  - Piece of code delimited by braces
  - Source file

Lexical Variables

- Only visible within enclosing block
- ```
  while (<$fh>) {
     my $data = munge($_);
  }
  # can't see $data here
  ```
- "Lexical" because the scope is defined purely by the text


  ```

Packages

- All Perl code is associated with a package
- A new package is created with package
  - `package MyPackage;`
- Think of it as a namespace
- Used to avoid name clashes with libraries
- Default package is called main

Package Variables

- Live in a package's symbol table
- Can be referred to using a fully qualified name
  - `$main::doctor`
  - `@Gallifrey::timelords`

Package Variables

- Package name not required within own package
- ```
  package Gallifrey;
  @timelords = ('Doctor', 'Master',
                'Rani');
  ```
- Can be seen from anywhere in the package (or anywhere at all when fully qualified)


  ```

Declaring Package Vars

- Can be predeclared with `our`
- `our ($doctor, @timelords, %home_planet);`
- Or (in older Perls) with `use vars`
- `use vars qw($doctor @timelords %home_planet);`

Lexical or Package

- When to use lexical variables or package variables?
- Simple answer
  - Always use lexical variables
- More complete answer
  - Always use lexical variables
  - Except for a tiny number of cases
- http://perl.plover.com/local.html

local

- You might see code that uses local
- `local $variable;`
- This doesn't do what you think it does
- Badly named function
- Doesn't create local variables
- Creates a local copy of a package variable
- Can be useful
  - In a small number of cases

local Example

- `$/` is a package variable
- It defines the input record separator
- You might want to change it
- Always localise changes
- ```
  {
    local $/ = “\n\n”;
    while (<FILE> ) {
      ...
    }
  }
  ```



  ```

Variable Exercises

- Write a simple script that contains two packages
  - One can be main
- Declare a package variable in one package
- Use it in the other package
- Declare lexical variables in both packages
- Where are they visible?
- Write a program that alters $/ using local


Strict and
Warnings

Coding Safety Net

- Perl can be a very loose programming language
- Two features can minimise the dangers
- `use strict` / `use warnings`
- A good habit to get into
- No serious Perl programmer codes without them

use strict

- Controls three things
- `use strict 'refs'` – no symbolic references
- `use strict 'subs'` – no barewords
- `use strict 'vars'` – no undeclared variables
- `use strict` – turn on all three at once
- turn them off (carefully) with `no strict`

use strict 'refs'

- Prevents symbolic references
- Using a variable as another variable's name
- ```
  $what = 'dalek';
  $$what = 'Karn';
  # sets $dalek to 'Karn'
  ```
- What if 'dalek' came from user input?
- People often think this is a cool feature
- It isn't


  ```

use strict 'refs' (cont)

- Better to use a hash
- ```
  $what = 'dalek';
  $alien{$what} = 'Karn';
  ```
- Self contained namespace
- Less chance of clashes
- More information (e.g. all keys)


  ```

use strict 'subs'

- No barewords
- Bareword is a word with no other interpretation
- e.g. word without $, @, %, &
- Treated as a function call or a quoted string
- `$dalek = Karn;`
- May clash with future reserved words

use strict 'vars'

- Forces predeclaration of variable names
- Prevents typos
- Less like BASIC - more like Ada
- Thinking about scope is good

use warnings

- Warns against dubious programming habits
- Some typical warnings
  - Variables used only once
  - Using undefined variables
  - Writing to read-only file handles
  - And many more...

Allowing Warnings

- Sometimes it's too much work to make code warnings clean
- Turn off use warnings locally
- Turn off specific warnings
- ```
  {
    no warnings 'deprecated';
    # dodgy code ...
  }
  ```
- See perldoc perllexwarn


  ```

Strict/Warning Examples

- Write a program that breaks each of strict's rules
- Turn strict off so that the programs compile
- Write a program that generates warnings
- How would you code around those warnings?

References

Introducing References

- A reference is a bit like pointer in languages like C and Pascal (but better)
- A reference is a unique way to refer to a variable.
- A reference can always fit into a scalar variable
- A reference looks like `SCALAR(0x20026730)`

Creating References

- Put \ in front of a variable name
  - `$scalar_ref = \$scalar;`
  - `$array_ref = \@array;`
  - `$hash_ref = \%hash;`
- Can now treat it just like any other scalar
  - `$var = $scalar_ref;`
  - `$refs[0] = $array_ref;`
  - `$another_ref = $refs[0];`

Creating References

- `[ LIST ]` creates anonymous array and returns a reference
- ```
  $aref = [ 'this', 'is', 'a', 'list'];
  $aref2 = [ @array ];
  ```
- `{ LIST }` creates anonymous hash and returns a reference
- ```
  $href = { 1 => 'one', 2 => 'two' };
  $href = { %hash };
  ```


Creating References

- ```
  @arr = (1, 2, 3, 4);
  $aref1 = \@arr;
  $aref2 = [ @arr ];
  print "$aref1\n$aref2\n";
  ```
- Output
  ```
  ARRAY(0x20026800)
  ARRAY(0x2002bc00)
  ```
- Second method creates a **copy** of the array


  ```

Using Array References

- Use `{$aref}` to get back an array that you have a reference to
- Whole array
- `@array = @{$aref};`
- `@rev = reverse @{$aref};`
- Single elements
- `$elem = ${$aref}[0];`
- `${$aref}[0] = 'foo';`

Using Hash References

- Use `{$href}` to get back a hash that you have a reference to
- Whole hash
- `%hash = %{$href};`
- `@keys = keys %{$href};`
- Single elements
- `$elem = ${$href}{key};`
- `${$href}{key} = 'foo';`

Using References

- Use arrow (`->`) to access elements of arrays or hashes
- Instead of `${$aref}[0]` you can use `$aref->[0]`
- Instead of `${$href}{key}` you can use `$href->{key}`

Using References

- You can find out what a reference is referring to using ref
- ```
  $aref = [ 1, 2, 3 ];
  print ref $aref; # prints ARRAY
  ```
- ```
  $href = { 1 => 'one',
            2 => 'two' };
  print ref $href; # prints HASH
  ```


Reference Examples

- Write a program that creates a reference to an array
- Update values in the array
- Print element from the array
- Print the whole array
- Repeat the exercise using a hash

Why Use References?

- Parameter passing
- Complex data structures

Parameter Passing

- What does this do?
- ```
  @arr1 = (1, 2, 3);
  @arr2 = (4, 5, 6);
  check_size(@arr1, @arr2);

  sub check_size {
    my (@a1, @a2) = @_;
    print @a1 == @a2 ?
              'Yes' : 'No';
  }
  ```



  ```

Why Doesn't It Work?

- `my (@a1, @a2) = @_;`
- Arrays are combined in `@_`
- All elements end up in `@a1`
- How do we fix it?
- Pass references to the arrays

Another Attempt

```
@arr1 = (1, 2, 3);
@arr2 = (4, 5, 6);
check_size(\@arr1, \@arr2);

sub check_size {
   my ($a1, $a2) = @_;
   print @$a1 == @$a2 ?
            'Yes' : 'No';
}
```


Parameter Exercise

- Write a subroutine that takes an array and a hash as arguments
- The function should return a list of the values from the hash that match keys given in the array

Complex Data Structures

- Another good use for references
- Try to create a 2-D array
- ```
  @arr_2d = ((1, 2, 3),
             (4, 5, 6),
             (7, 8, 9));
  ```
- @arr_2d contains
  `(1, 2, 3, 4, 5, 6, 7, 8, 9)`
- This is known a *array flattening*


  ```

Complex Data Structures

- 2D Array using references
- `@arr_2d = ([1, 2, 3],`
  `[4, 5, 6],`
  `[7, 8, 9]);`
- But how do you access individual elements?
- `$arr_2d[1]` is ref to array (4, 5, 6)
- `$arr_2d[1]->[1]` is element 5

Complex Data Structures

- Another 2D Array
- `$arr_2d = [[1, 2, 3],`
  `[4, 5, 6],`
  `[7, 8, 9]];`
- `$arr_2d->[1]` is ref to array (4, 5, 6)
- `$arr_2d->[1]->[1]` is element 5
- Can omit intermediate arrows
- `$arr_2d->[1][1]`

More Data Structures

- Imagine the following data file
- ```
  Jones,Martha,UNIT
  Harkness,Jack,Torchwood
  Smith,Sarah Jane,Journalist
  ```
- What would be a good data structure?
- Hash for each record
- Array of records
- Array of hashes


  ```

More Data Structures

- Building an array of hashes
- ```
  my @records;
  my @cols =
   ('s_name', 'f_name', 'job');

  while (<FILE>) {
    chomp;
    my %rec;
    @rec{@cols} = split /,/;
    push @records, \%rec;
  }
  ```



  ```

Using an Array of Hashes

```
foreach (@records) {
   print "$_->{f_name} ",
         "$_->{s_name} ".
         "is a $_->{job}\n";
}
```


Complex Data Structures

- Many more possibilities
  - Hash of hashes
  - Hash of lists
  - Multiple levels (list of hash of hash, etc.)
- Lots of examples in “perldoc perldsc” (the data structures cookbook)

Data Structure Examples

- Dalek,Skaro,The Daleks
  Cyberman,Mondas,The Tenth Planet
  Sontaran,Sontar,The Time Warrior
  Ood,Oodsphere,The Impossible Planet
- Read the data into a hash of hashes
- Prompt for a species
- Print the data for that species


Sorting

Sorting

- Perl has a sort function that takes a list and sorts it
- `@sorted = sort @array;`
- Note that it does not sort the list in place
- `@array = sort @array;`

Sort Order

- The default sort order is ASCII
- ```
  @chars = sort 'e', 'b', 'a', 'd', 'c';
  # @chars has ('a', 'b', 'c', 'd', 'e')
  ```
- This can sometimes give strange results
- ```
  @chars = sort 'E', 'b', 'a', 'D', 'c';
  # @chars has ('D', 'E', 'a', 'b', 'c')
  ```
- ```
  @nums = sort 1 .. 10;
  # @nums has (1, 10, 2, 3, 4,
  #            5,  6, 7, 8, 9)
  ```



  ```

Sorting Blocks

- Can add a "sorting block" to customise sort order
- `@nums =`
  ```
  sort { $a <=> $b } 1 .. 10;
  ```
- Perl puts two of the values from the list into `$a` and `$b`
- Block compares values and returns -1, 0 or 1
- `<=>` does this for numbers (`cmp` for strings)

Sort Examples

- Other simple sort examples
- `sort { $b cmp $a } @words`
- `sort { lc $a cmp lc $b } @words`
-
```
sort { substr($a, 4)
       cmp substr($b, 4) } @lines
```


Sorting Subroutines

- Can also use a subroutine name in place of a code block
- `@words = sort dictionary @words;`

```
sub dictionary {
   # Don't change $a and $b
   my ($A, $B) = ($a, $b);
   $A =~ s/\W+//g;
   $B =~ s/\W+//g;
   $A cmp $B;
}
```


Sorting Names

- ```
  my @names = ('Rose Tyler',
               'Martha Jones',
               'Donna Noble',
               'Amy Pond');

  @names = sort sort_names @names;
  ```
- Need to write sort_names so that it sorts on surname and then forename.


  ```

Sorting Names (cont)

```
sub sort_names {
    my @a = split /\s/, $a;
    my @b = split /\s/, $b;

    return $a[1] cmp $b[1]
      or $a[0] cmp $b[0];
}
```


Sort Examples

- A data file contains forename, surname and age
- Sort the data by these criteria
  - Surname (ascending)
  - Forename (ascending)
  - Age (descending)

More Complex Sorts

- ```
  sub sort_names {
     my @a = split /\s/, $a;
     my @b = split /\s/, $b;

     return $a[1] cmp $b[1]
       or $a[0] cmp $b[0];
  }
  ```
- Can be inefficient on large amounts of data
- Multiple splits on the same data


  ```

More Efficient Sorts

- Split each row only once
- `@split = map { [ split ] } @names;`
- Do the comparison
- `@sort = sort { $a->[1] cmp $b->[1]`
  `or $a->[0] cmp $b->[0] } @split;`
- Join the data together
- `@names = map { join ' ', @$_ }`
  `@sort;`

Put It All Together

- Can rewrite this as
- ```
  @names = map  { join ' ', @$_ }
     sort { $a->[1] cmp $b->[1]
       || $a->[0] cmp $b->[0] }
     map  { [ split ] } @names;
  ```
- All functions work on the output from the previous function in the chain


  ```

Schwartzian Transform

- ```
  @data_out =
     map  { $_->[1] }
     sort { $a->[0] cmp $a->[0] }
     map  { [func($_), $_] }
      @data_in;
  ```
- Old Lisp trick
- Named after Randal Schwartz


  ```

More Sort Examples

- Rewrite the previous sort example to use a Schwartzian Transform

Reusable Code

Why Write Modules?

- Code reuse
- Prevent reinventing the wheel
- Easier to share across projects
- Better design, more generic

Basic Module

- ```
  use strict;
  use warnings;

  package MyModule;

  use Exporter;
  our @ISA = ('Exporter');
  our @EXPORT = ('my_sub');

  sub my_sub {
    print "This is my_sub\n";
  }

  1;
  ```



  ```

Using MyModule.pm

- use MyModule;

```
# my_sub is now available
# for use within your
# program

my_sub();
# Prints "This is my_sub()"
```


Explaining MyModule.pm

- Much of MyModule.pm is concerned with exporting subroutine names
- Subroutine full name
  - `MyModule::my_sub()`
- Exporting abbreviates that
  - `my_sub()`

Packages Revisited

- Every subroutine lives in a package
- The default package is `main`
- New packages are introduced with the `package` keyword
- A subroutine's full name is `package::name`
- Package name can be omitted from within same package
- Like family names

Using Exporter

- The module Exporter.pm handles the export of subroutine (and variable) names
- Exporter.pm defines a subroutine called `import`
- `import` is automatically called whenever a module is used
- `import` puts references to our subroutines into our caller's symbol table

How Exporter Works

- How does MyModule use Exporter's `import` subroutine?
- We make use of inheritance
- Inheritance is defined using the `@ISA` array
- If we call a subroutine that doesn't exist in our module, then the modules in `@ISA` are also checked
- Therefore `Exporter::import` is called

Exporting Symbols

- How does import know which subroutines to export?
- Exports are defined in `@EXPORT` or `@EXPORT_OK`
- Automatic exports are defined in `@EXPORT`
- Optional exports are defined in `@EXPORT_OK`

Exporting Symbol Sets

- You can define sets of exports in `%EXPORT_TAGS`
- Key is set name
- Value is reference to an array of names
-
```
our %EXPORT_TAGS =
   (advanced => [ qw( my_sub
                      my_other_sub ) ];

use MyModule qw(:advanced);
my_sub();
my_other_sub();
```


Why Use @EXPORT_OK?

- Give your users the choice of which subroutines to import
- Less chances of name clashes
- Use `@EXPORT_OK` in preference to `@EXPORT`
- Document the exported names and sets

Exporting Variables

- You can also export variables
- ```
  @EXPORT_OK = qw($scalar,
                  @array,
                  %hash);
  ```
- Can be part of export sets
- Any variables you export must be package variables


  ```

Writing Modules The Easy Way

- A lot of module code is similar
- Don't write the boilerplate yourself
- Copy from an existing module
- Or look at Module::Starter

Module Examples

- Write a module which contains at least three functions
- Put some functions in @EXPORT and some in @EXPORT_OK
- Create some export tags
- Use your module in a program

That's All Folks

- Any Questions?