Introduction to Perl

An Introduction to Perl Programming

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

UKUUG
25th February 2009

What We Will Cover

- What is Perl?
- Creating and running a Perl program
- Input and Output
- Perl variables
- Operators and Functions
- Conditional Constructs

What We Will Cover

- Subroutines
- Regular Expressions
- References
- Smart Matching
- Finding and using Modules
- Further Information

Schedule

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

Resources

- Slides available on-line
  - http://mag-sol.com/train/public/2009-02/begin
- Also see Slideshare
  - http://www.slideshare.net/davorg/slideshows
- Mailing List
  - http://lists.mag-sol.com/mailman/listinfo/beg2009
- Get Satisfaction
  - http://getsatisfaction.com/magnum

What is Perl?

UKUUG
25th February 2009

Perl's Name

- Practical Extraction and Reporting Language
- Pathologically Eclectic Rubbish Lister
- "Perl" is the language
- "perl" is the compiler
- Never "PERL"

Typical uses of Perl

- Text processing
- System administration tasks
- CGI and web programming
- Database interaction
- Other Internet programming

Less typical uses of Perl

- Human Genome Project
- NASA

What is Perl Like?

- General purpose programming language
- Free (open source)
- Fast
- Flexible
- Secure
- Fun

The Perl Philosophy

- There's more than one way to do it
- Three virtues of a programmer
  - Laziness
  - Impatience
  - Hubris
- Share and enjoy!

Creating and Running a Perl Program

UKUUG
25th February 2009

Creating a Perl Program

- Our first Perl program
  ```
  print "Hello world\n";
  ```
- Put this in a file called hello.pl

Running a Perl Program

- Running a Perl program from the command line
- `$ perl hello.pl`

Running a Perl Program

- The "shebang" line (Unix, not Perl)
  ```
  #!/usr/bin/perl
  ```
- Make program executable
  ```
  $ chmod +x hello.pl
  ```
- Run from command line
  ```
  $ ./hello.pl
  ```


Perl Comments

- Add comments to your code
- Start with a hash (#)
- Continue to end of line
- ```
  # This is a hello world program
  print "Hello, world!\n"; # print
  ```



  ```

Command Line Options

- Many options to control execution of the program
- For example, `-w` turns on warnings
- Use on command line
  `perl -w hello.pl`
- Or on shebang line
  `#!/usr/bin/perl -w`

Perl variables

UKUUG
25th February 2009

What is a Variable?

- A place where we can store data
- A variable needs a name to
  - retrieve the data stored in it
  - put new data in it

Variable Names

- Contain alphanumeric characters and underscores
- User variable names may not start with numbers
- Variable names are preceded by a punctuation mark indicating the type of data

Types of Perl Variable

- Different types of variables start with a different symbol
  - Scalar variables start with $
  - Array variables start with @
  - Hash variables start with %
- More on these types soon

Declaring Variables

- You don't need to declare variables in Perl
- But it's a very good idea
  - typos
  - scoping
- Using the strict pragma

```
use strict;
my $var;
```


Scalar Variables

- Store a single item of data
- `my $name = "Arthur";`
- `my $whoami = 'Just Another Perl Hacker';`
- `my $meaning_of_life = 42;`
- `my $number_less_than_1 = 0.000001;`
- `my $very_large_number = 3.27e17;`
  `# 3.27 times 10 to the power of 17`

Type Conversions

- Perl converts between strings and numbers whenever necessary
- Add int to a floating point number

```
my $sum = $meaning_of_life +
          $number_less_than_1;
```

- Putting a number into a string

```
print "$name says, 'The meaning of
life is $sum.'\n";
```


Quoting Strings

- Single quotes don't expand variables or escape sequences

```
my $price = '$9.95';
```

- Double quotes do

```
my $invline =
        "24 widgets @ $price each\n";
```

- Use a backslash to escape special characters in double quoted strings

```
print "He said \"The price is \$300\"";
```


Better Quotes

- This can look ugly

```
print "He said \"The price is \$300\"";
```

- This is a tidier alternative

```
print qq(He said "The price is \$300");
```

- Also works for single quotes

```
print q(He said "That's too expensive");
```


Undefined Values

- A scalar variable that hasn't had data put into it will contain the special value “undef”
- Test for it with “defined()” function
- `if (defined($my_var)) { ... }`

Array Variables

- Arrays contain an ordered list of scalar values
- ```
  my @fruit = ('apples', 'oranges',
               'guavas', 'passionfruit',
               'grapes');
  ```
- ```
  my @magic_numbers = (23, 42, 69);
  ```
- ```
  my @random_scalars = ('mumble', 123.45,
                        'dave cross',
                        -300, $name);
  ```



  ```

Array Elements

- Accessing individual elements of an array
- ```
  print $fruits[0];
  # prints "apples"
  ```
- Note: Indexes start from zero
- ```
  print $random_scalars[2];
  # prints "dave cross"
  ```
- Note use of $ as individual element of an array is a scalar

Array Slices

- Returns a list of elements from an array
- ```
  print @fruits[0,2,4];
  # prints "apples", "guavas",
  # "grapes"
  ```
- ```
  print @fruits[1 .. 3];
  # prints "oranges", "guavas",
  # "passionfruit"
  ```
- Note use of @ as we are accessing more than one element of the array

Setting Array Values

- `$array[4] = 'something';`
  `$array[400] = 'something else';`
- Also with slices
- `@array[4, 7 .. 9] = ('four', 'seven',`
  `'eight', 'nine');`
- `@array[1, 2] = @array[2, 1];`
- Doesn't need to be an array!
  - `($x, $y) = ($y, $x);`

Array Size

- `$#array` is the index of the last element in `@array`
- Therefore `$#array + 1` is the number of elements
- `$count = @array;`
  does the same thing and is easier to understand

Hash Variables

- Hashes implement “look-up tables” or “dictionaries”
- Initialised with a list

```
%french = ('one', 'un', 'two', 'deux',
           'three', 'trois');
```

- "fat comma" (=>) is easier to understand

```
%german = (one   => 'ein',
           two   => 'zwei',
           three => 'drei');
```


Accessing Hash Values

- `$three = $french{three};`
- `print $german{two};`
- As with arrays, notice the use of $ to indicate that we're accessing a single value

Hash Slices

- Just like array slices
- Returns a list of elements from a hash

```
print @french{'one','two','three'};
# prints "un", "deux" & "trois"
```

- Again, note use of @ as we are accessing more than one value from the hash

Setting Hash Values

- `$hash{foo} = 'something';`
- `$hash{bar} = 'something else';`
- Also with slices
- `@hash{'foo', 'bar'} = ('something', 'else');`
- `@hash{'foo', 'bar'} = @hash{'bar', 'foo'};`

More About Hashes

- Hashes are not sorted
- There is no equivalent to `$#array`
- `print %hash` is unhelpful
- We'll see ways round these restrictions later

Special Perl Variables

- Perl has many special variables
- Many of them have punctuation marks as names
- Others have names in ALL_CAPS
- They are documented in perlvar

The Default Variable

- Many Perl operations either set `$_` or use its value if no other is given
- `print; # prints the value of $_`
- If a piece of Perl code seems to be missing a variable, then it's probably using `$_`
- Think of “it” or “that” in English

Using $_

- ```
  while (<FILE>) {
     if (/regex/) {
        print;
     }
  }
  ```
- Three uses of $_


  ```

A Special Array

- @ARGV
- Contains your programs command line arguments
- `perl printargs.pl foo bar baz`
- ```
  my $num = @ARGV;
  print "$num arguments: @ARGV\n";
  ```



  ```

A Special Hash

- `%ENV`
- Contains the *environment variables* that your script has access to.
- Keys are the variable names
  Values are the… well… values!
- `print $ENV{PATH};`

Input and Output

UKUUG
25th February 2009

Input and Output

- Programs become more useful with input and output
- We'll see more input and output methods later in the day
- But here are some simple methods

Output

- The easiest way to get data out of a Perl program is to use `print`
- `print “Hello world\n”;`

Input

- The easiest way to get data into a Perl program is to read from `STDIN`
- `$input = <STDIN>;`
- `< ... >` is the “read from filehandle” operator
- `STDIN` is the standard input filehandle

A Complete Example

- ```
  #!/usr/bin/perl

  print 'What is your name: ';
  $name = <STDIN>;
  print “Hello $name”;
  ```



  ```

Operators and Functions

UKUUG
25th February 2009

Operators and Functions

- What are operators and functions?
- "Things" that do "stuff"
- Routines built into Perl to manipulate data
- Other languages have a strong distinction between operators and functions
  - in Perl that distinction can be a bit blurred
- See perlop and perlfunc

Arithmetic Operators

- Standard arithmetic operations
  add (+), subtract (-), multiply (*), divide (/)
- Less standard operations
  modulus (%), exponentiation (**)
- ```
  $speed = $distance / $time;
  $vol = $length * $breadth * $height;
  $area = $pi * ($radius ** 2);
  $odd = $number % 2;
  ```



  ```

Shortcut Operators

- Often need to do things like
```
$total = $total + $amount;
```
- Can be abbreviated to
```
$total += $amount;
```
- Even shorter
```
$x++; # same as $x += 1 or $x = $x + 1
$y--; # same as $y -= 1 or $y = $y - 1
```
- Subtle difference between $x++ and ++$x

String Operators

- Concaternation (.)
```
$name = $firstname . ' ' . $surname;
```
- Repetition (x)
```
$line = '-' x 80;
$police = 'hello ' x 3;
```
- Shortcut versions available
```
$page .= $line; # $page = $page . $line
$thing x= $i;   # $thing = $thing x $i
```


File Test Operators

- Check various attributes of a file
- `-e $file` does the file exist
- `-r $file` is the file readable
- `-w $file` is the file writeable
- `-d $file` is the file a directory
- `-f $file` is the file a normal file
- `-T $file` is a text file
- `-B $file` is a binary file

Functions

- Have longer names than operators
- Can take more arguments than operators
- Arguments follow the function name
- See perlfunc for a complete list of Perl's built-in functions

Function Return Values

- Functions can return scalars or lists (or nothing)
- ```
  $age = 29.75;
  $years = int($age);
  ```
- ```
  @list = ('a', 'random',
           'collection', 'of',
           'words');
  @sorted = sort(@list);
  # a collection of random words
  ```


String Functions

- `length` returns the length of a string
```
$len = length $a_string;
```
- `uc` and `lc` return upper and lower case versions of a string
```
$string = 'MiXeD CaSe';
print "$string\n", uc $string, "\n",
      lc $string;
```
- See also `ucfirst` and `lcfirst`

More String Functions

- `chop` removes the last character from a string and returns it

```
$word = 'word';
$letter = chop $word;
```

- `chomp` removes the last character only if it is a newline and returns true or false appropriately

Substrings

- `substr` returns substrings from a string

```
$string = 'Hello world';
print substr($string, 0, 5);
                # prints 'Hello'
```

- Unlike many other languages you can *assign* to a substring

```
substr($string, 0, 5) = 'Greetings';
print $string;
# prints 'Greetings world'
```


Numeric Functions

- `abs` returns the absolute value
- `cos`, `sin`, `tan` standard trigonometric functions
- `exp` exponentiation using *e*
- `log` logarithm to base *e*
- `rand` returns a random number
- `sqrt` returns the square root

Array Manipulation

- `push` adds a new element to the end of an array

```
push @array, $value;
```

- `pop` removes and returns the last element in an array

```
$value = pop @array;
```

- `shift` and `unshift` do the same for the start of an array

Array Manipulation

- `sort` returns a sorted list (it *does not* sort the list in place)
```
@sorted = sort @array;
```
- `sort` does a lot more besides, see the docs (perldoc -f sort)
- `reverse` returns a reversed list
```
@reverse = reverse @array;
```


Arrays and Strings

- `join` takes an array and returns a string

```
@array = (1 .. 5);
$string = join ' ', @array;
# $string is '1 2 3 4 5'
```

- `split` takes a string and converts it into an array

```
$string = '1~2~3~4~5';
@array = split(/~/, $string);
# @array is (1, 2, 3, 4, 5)
```


Hash Functions

- `delete` removes a key/value pair from a hash
- `exists` tells you if an element exists in a hash
- `keys` returns a list of all the keys in a hash
- `values` returns a list of all the values in a hash

File Operations

- `open` opens a file and associates it with a filehandle
```
open(FILE, 'in.dat');
```
- You can then read the file with `<FILE>`
```
$line = <FILE>; # one line
@lines = <FILE>; # all lines
```
- Finally, close the file with close
```
close(FILE);
```


Other File Functions

- `read` to read a fixed number of bytes into a buffer
```
$bytes = read(FILE, $buffer, 1024);
```
- `seek` to move to a random position in a file
```
seek(FILE, 0, 0);
```
- `tell` to get current file position
```
$where = tell FILE;
```
- `truncate` to truncate file to given size
```
truncate FILE, $where;
```


Time Functions

- `time` returns the number of seconds since Jan 1st 1970
- `$now = time;`
- `localtime` converts that into more usable values
- `($sec, $min, $hour, $mday, $mon, $year, $wday, $yday, $isdst) = localtime($now);`

`localtime` Caveats

- `$mon` is 0 to 11
- `$year` is years since 1900
- `$wday` is 0 (Sun) to 6 (Sat)

`localtime` Shortcuts

- It's common to use `localtime` on the current time
- `@time_bits = localtime(time);`
- Call to time can be omitted
- `@time_bits = localtime;`
- Use array slices
- `($d, $m, $y) = (localtime)[3 .. 5];`

Date & Time Formats

- You can get formatted dates by fiddling the return values from `localtime`
- Easier to use `strftime` (from POSIX.pm)
- `print strftime('%Y-%m-%d', localtime);`
- Format followed by list of date/time values
- Format is POSIX standard
  - Like UNIX date command
- `print strftime ('%d %B %y', localtime);`

Conditional Constructs

UKUUG
25th February 2009

Conditional Constructs

- Conditional constructs allow us to choose different routes of execution through the program
- This makes for far more interesting programs
- The unit of program execution is a *block* of code
- Blocks are delimited with braces { … }

Conditional Constructs

- Conditional blocks are controlled by the evaluation of an expression to see if it is true or false
- But what is truth?

What is Truth?

- In Perl it's easier to answer the question "what is false?"
  - 0 (the number zero)
  - '' (the empty string)
  - undef (an undefined value)
  - () (an empty list)
- Everything else is true

Comparison Operators

- Compare two values in some way
  - are they equal
    ```
    $x == $y or $x eq $y
    $x != $y or $x ne $y
    ```
  - Is one greater than another
    ```
    $x > $y or $x gt $y
    $x >= $y or $x ge $y
    ```
  - Also < (`lt`) and <= (`le`)

Comparison Examples

- `62 > 42                    # true`
- `'0' == (3 * 2) - 6  # true`
- `'apple' gt 'banana' # false`
- `'apple' == 'banana' # true(!)`
- `1 + 2 == '3 bears'  # true`
- `1 + 3 == 'three'    # false`

Boolean Operators

- Combine two or more conditional expressions into one
- `EXPR_1 and EXPR_2`
  true if both EXPR_1 and EXPR_2 are true
- `EXPR_1 or _EXPR_2`
  true if either EXPR_1 or _EXPR_2 are true
- Alternative syntax && for `and` and `||` for `or`
- Different precedence though

Short-Circuit Operators

- `EXPR_1 or EXPR_2`
  Only need to evaluate EXPR_2 if EXPR_1 evaluates as false
- We can use this to make code easier to follow

```
open FILE, 'something.dat'
  or die "Can't open file: $!";
```

- `@ARGV == 2 or print $usage_msg;`

if

- if - our first conditional
- `if (EXPR) { BLOCK }`
- Only executes BLOCK if EXPR is true

```
if ($name eq 'Doctor') {
   regenerate();
}
```


if ... else ...

- if … else ... - an extended if
  `if (EXPR) { BLOCK1 } else { BLOCK2}`
- If EXPR is true, execute BLOCK1, otherwise execute BLOCK2
- ```
  if ($name eq 'Doctor') {
    regenerate();
  } else {
    die "Game over!\n";
  }
  ```



  ```

if ... elsif ... else ...

- if … elsif … else ... - even more control

```
if (EXPR1) { BLOCK1 }
elsif (EXPR2) { BLOCK2 }
else { BLOCK3 }
```

- If EXPR1 is true, execute BLOCK1
  else if EXPR2 is true, execute BLOCK2
  otherwise execute BLOCK3

if ... elsif ... else ...

- An example

```
if ($name eq 'Doctor') {
  regenerate();
} elsif ($tardis_location
            eq $here) {
  escape();
} else {
  die "Game over!\n";
}
```


while

- while - repeat the same code
  `while (EXPR) { BLOCK }`
- Repeat BLOCK while EXPR is true

```
while ($dalek_prisoners) {
   print "Ex-ter-min-ate\n";
   $dalek_prisoners--;
}
```


until

- until - the opposite of while
  `until (EXPR) { BLOCK }`
- Execute BLOCK until EXPR is true

```
until ($regenerations == 12) {
   print "Regenerating\n";
   regenerate();
   $regenerations++;
}
```


for

- for - more complex loops
  `for (INIT; EXPR; INCR) { BLOCK }`
- Like C
- Execute INIT
  If EXPR is false, exit loop, otherwise execute BLOCK, execute INCR and retest EXPR

for

- An example

```
for ($i = 1; $i <= 10; $i++) {
   print "$i squared is ",
         $i * $i, "\n";
}
```

- Used surprisingly rarely

foreach

- foreach - simpler looping over lists
  `foreach VAR (LIST) { BLOCK }`
- For each element of LIST, set VAR to equal the element and execute BLOCK
- ```
  foreach $i (1 .. 10) {
     print "$i squared is ",
           $i * $i, "\n";
  }
  ```



  ```

foreach

- Another example

```
my %months = (Jan => 31, Feb => 28,
              Mar => 31, Apr => 30,
              May => 31, Jun => 30,
              … );
foreach (keys %months) {
  print "$_ has $months{$_} days\n";
}
```


Using `while` Loops

- Taking input from STDIN
- ```
  while (<STDIN>) {
     print;
  }
  ```
- This is the same as
  ```
  while (defined($_ = <STDIN>)) {
     print $_;
  }
  ```



  ```

Breaking Out of Loops

- `next` - jump to next iteration of loop
- `last` - jump out of loop
- `redo` - jump to start of *same* iteration of loop

Subroutines

UKUUG
25th February 2009

Subroutines

- Self-contained "mini-programs" within your program
- Make it easy to repeat code
- Subroutines have a name and a block of code
- ```
  sub NAME {
     BLOCK
  }
  ```



  ```

Subroutine Example

- Simple subroutine example

```
sub exterminate {
   print "Ex-Ter-Min-Ate!!\n";
   $timelords--;
}
```


Calling a Subroutine

- `&exterminate;`
- `exterminate();`
- `exterminate;`
- last one only works if function has been predeclared

Subroutine Arguments

- Functions become far more useful if you can pass arguments to them
- `exterminate('The Doctor');`
- Arguments end up in the `@_` array within the function
- ```
  sub exterminate {
     my ($name) = @_;
     print "Ex-Ter-Min-Ate $name\n";
     $timelords--;
  }
  ```



  ```

Multiple Arguments

- As @_ is an array it can contain multiple arguments
- ```
  sub exterminate {
    foreach (@_) {
      print "Ex-Ter-Min-Ate $_\n";
      $timelords--;
    }
  }
  ```



  ```

Calling Subroutines

- A subtle difference between `&my_sub` and `my_sub()`
- `&my_sub` passes on the contents of `@_` to the called subroutine

```
sub first { &second };
sub second { print @_ };
first('some', 'random', 'data');
```

- You usually don't want to do that

By Value or Reference

- Passing by value passes the *value* of the variable into the subroutine. Changing the argument doesn't alter the external variable
- Passing by value passes the *actual* variable. Changing the argument alters the external value
- Perl allows you to choose

By Value or Reference

- Simulating pass by value
- `my ($arg1, $arg2) = @_;`
- Updating `$arg1` and `$arg2` doesn't effect anything outside the subroutine
- Simulating pass by reference
- `$_[0] = 'whatever';`
- Updating the contents of `@_` updates the external values

Returning Values

- Use `return` to return a value from a subroutine

```
sub exterminate {
  if (rand > .25) {
    print "Ex-Ter-Min-Ate $_[0]\n";
    $timelords--;
    return 1;
  } else {
    return;
  }
}
```


Returning a List

- Returning a list from a subroutine

```
sub exterminate {
  my @exterminated;
  foreach (@_) {
    if (rand > .25) {
      print "Ex-Ter-Min-Ate $_\n";
      $timelords--;
      push @exterminated, $_;
    }
  }
  return @exterminated;
}
```


Regular Expressions

UKUUG
25th February 2009

Regular Expressions

- Patterns that match strings
- A bit like wild-cards
- A "mini-language" within Perl
  - Alien DNA
- The key to Perl's text processing power
- Sometimes overused!
- Documented in perldoc perlre

Match Operator

- m/PATTERN/ - the match operator
- Works on $_ by default
- In scalar context returns true if the match succeeds
- In list context returns list of "captured" text
- m is optional if you use / characters
- With m you can use any delimiters

Match Examples

- ```
  while (<FILE>) {
     print if /foo/;
     print if /bar/i;
     print if m|http://|;
  }
  ```



  ```

Substitutions

- s/PATTERN/REPLACEMENT/ - the substitution operator
- Works on $_ by default
- In scalar context returns true if substitution succeeds
- In list context returns number of replacements
- Can choose any delimiter

Substitution Examples

- ```
  while (<FILE>) {
     s/teh/the/gi;
     s/freind/friend/gi;
     s/sholud/should/gi;
     print;
  }
  ```



  ```

Binding Operator

- If we want `m//` or `s///` to work on something other than `$_` then we need to use the binding operator
- `$name =~ s/Dave/David/;`

Metacharacters

- Matching something other than literal text
- ^ - matches start of string
- $ - matches end of string
- . - matches any character (except \n)
- \s - matches a whitespace character
- \S - matches a non-whitespace character

More Metacharacters

- `\d` - matches any digit
- `\D` - matches any non-digit
- `\w` - matches any "word" character
- `\W` - matches any "non-word" character
- `\b` - matches a word boundary
- `\B` - matches anywhere except a word boundary

Metacharacter Examples

- ```
  while (<FILE>) {
      print if m|^http|;
      print if /\bperl\b/;
      print if /\S/;
      print if /\$\d\.\d\d/;
  }
  ```



  ```

Quantifiers

- Specify the number of occurrences
- `?` - match zero or one
- `*` - match zero or more
- `+` - match one or more
- `{n}` - match exactly n
- `{n,}` - match n or more
- `{n,m}` - match between n and m

Quantifier Examples

- ```
  while (<FILE>) {
     print if /whiske?y/i;
     print if /so+n/;
     print if /\d*\.\d+/;
     print if /\bA\w{3}\b/;
  }
  ```



  ```

Character Classes

- Define a class of characters to match
- `/[aeiou]/ # match any vowel`
- Use `-` to define a contiguous set
- `/[A-Z]/ # match upper case letters`
- Use `^` to match inverse set
- `/[^A-Za-z] # match non-letters`

Alternation

- Use | to match one of a set of options
- `/rose|martha|donna/i;`
- Use parentheses for grouping
- `/^(rose|martha|donna)$/i;`

Capturing Matches

- Parentheses are also used to capture parts of the matched string
- The captured parts are in $1, $2, etc…

```
while (<FILE>) {
   if (/^(\w+)\s+(\w+)/) {
      print "The first word was $1\n";
      print "The second word was $2";
   }
}
```


Returning Captures

- Captured values are also returned if the match operator is used in list context
- ```
  my @nums = $text =~ /(\d+)/g;
  print "I found these integers:\n";
  print "@nums\n";
  ```



  ```

More Information

- perldoc perlre
- perldoc perlretut
- *Mastering Regular Expressions* – Jeffrey Freidl

References


Magnum
Solutions Limited
Open Source Consultancy, Development & Training

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
  ```


Why Use References?

- Parameter parsing
- 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?

- 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';
}
```


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)

Smart Matching

UKUUG
25th February 2009

Smart Matching

- Introduced in Perl 5.10
- Powerful matching operator
- DWIM
- Examines operands
- Decides which match to apply

Smart Match Operator

- ~~
- New operator
- Looks a bit like the binding operator (=~)
- Can be used in place of it
- `$some_text =~ /some regex/`
- Can be replaced with
- `$some_text ~~ /some regex/`

Smarter Matching

- If one of its operands is a regex
- `~~` does a regex match
- Cleverer than that though
- `%hash ~~ /regex/`
- Regex match on hash keys
- `@array ~~ /regex/`
- Regex match on array elements

More Smart Matches

- `@array1 ~~ @array2`
- Checks that arrays are the same
- `$scalar ~~ @array`
- Checks scalar exists in array
- `$scalar ~~ %hash`
- Checks scalar is a hash key

Smart Scalar Matches

- What kind of match does this do?
- `$scalar1 ~~ $scalar2`
- It depends
- If both look like numbers
- ~~ acts like ==
- Otherwise
- ~~ acts like `eq`

Finding and Using Modules

UKUUG
25th February 2009

Modules

- A module is a reusable 'chunk' of code
- Perl comes with over 100 modules (see “perldoc perlmodlib” for list)
- Perl has a repository of freely-available modules - the Comprehensive Perl Archive Network (CPAN)
  - http://www.cpan.org
  - http://search.cpan.org

Finding Modules

- http://search.cpan.org
- Search by:
  - module name
  - distribution name
  - author name
- Note: CPAN also contains newer versions of standard modules

Installing Modules (The Hard Way)

- Download distribution file
  - MyModule-X.XX.tar.gz
- Unzip
  - `$ gunzip MyModule-X.XX.tar.gz`
- Untar
  - `$ tar xvf MyModule-X.XX.tar`
- Change directory
  - `$ cd MyModule-X.XX`

Installing Modules (The Hard Way)

- Create Makefile
  - `$ perl Makefile.PL`
- Build Module
  - `$ make`
- Test Build
  - `$ make test`
- Install Module
  - `$ make install`

Installing Modules (The Hard Way)

- Note: May need root permissions for `make install`
- You can have your own personal module library
- `perl Makefile.PL PREFIX=~/perl`
  - need to adjust `@INC`

Installing Modules (The Easy Way)

- Note: May not work (or may need some configuration) through a firewall
- CPANPLUS.pm is included with newer Perls
- Automatically carries out installation process
- Can also handle required modules
- May still need to be root
  - Can use 'sudo'

Installing Modules (The Easy Way)

- ```
  cpanp
  [ ... some stuff ... ]
  CPAN Terminal> install Some::Module
  [ ... some more stuff ... ]
  CPAN Terminal> quit
  ```
- Or
- `cpanp -i Some::Module`


  ```

Using Modules

- Two types of module:
  - Functions vs Objects
- Functional modules export new subroutines and variables into your program
- Object modules usually don't
- Difference not clear cut (e.g. CGI.pm)

Using Functional Modules

- Import defaults:
```
use My::Module;
```
- Import optional components:
```
use My::Module qw(my_sub
                  @my_arr);
```
- Import defined sets of components:
```
use My:Module qw(:advanced);
```
- Use imported components:
```
$data = my_sub(@my_arr);
```


Using Object Modules

- Use the module:
  ```
  use My::Object;
  ```
- Create an object:
  ```
  $obj = My::Object->new;
  ```
  - Note: new is just a convention
- Interact using object's methods
  ```
  $obj->set_name($name);
  ```


Useful Standard Modules

- constant
- Time::Local
- Text::ParseWords
- Getopt::Std
- Cwd
- File::Basename
- File::Copy
- POSIX
- CGI
- Carp
- Benchmark
- Data::Dumper

Useful Non-Standard Modules

- Template
- DBI
- DBIx::Class
- DateTime
- HTML::Parser
- HTML::Tidy
- LWP
- WWW::Mechanize
- Email::Simple
- XML::LibXML
- XML::Feed
- Moose

More Information

Perl Websites

- Perl Home Page
  - http://www.perl.org
- CPAN
  - http://www.cpan.org
  - http://search.cpan.org
- Perl Mongers (Perl User Groups)
  - http://www.pm.org
  - http://london.pm.org

Perl Websites

- use.perl – Perl News & Journals
  - http://use.perl.org/
- Perl Monks – Perl Help and Advice
  - http://perlmonks.org/
- Perl Documentation Online
  - http://perldoc.perl.org/

Perl Conferences

- The Perl Conference
  (part of the Open Source Convention)
  - July 20-24 2009, San Jose, California
  - http://conferences.oreilly.com/oscon
- Yet Another Perl Conference
  - August 3-5 2009, Lisbon, Portugal
  - http://www.yapceurope.org

Perl Conferences

- Other YAPCs
  - Pittsburgh, Pennsylvania
  - Brazil
  - Tokyo
- OSDC
  - Israel
  - Australia

Perl Workshops

- One-day grassroots conferences
- Germany, Israel, Pittsburgh, Nordic, Netherlands, France, Belgium, Russia, Minnesota, Austria
- Perl Review Calendar
  - www.theperlreview.com/community_calendar

Perl Mailing Lists

- See http://lists.perl.org for full details
  - Perl Mongers (social discussion)
  - CGI
  - DBI
  - XML
  - Beginners
  - Advocacy
  - Fun with Perl

Perl Books

- Books for learning Perl
  - Learning Perl (5th ed - June 2008)
    Schwartz, Phoenix & foy (O'Reilly)
  - Intermediate Perl
    Schwartz, foy & Phoenix (O'Reilly)
  - Beginning Perl
    Cozens (Wrox)
    http://www.perl.org/books/beginning-perl/

Perl Books

- Books you should have access to
  - Programming Perl (3rd edition)
    Wall, Christiansen & Orwant (O'Reilly)
  - The Perl Cookbook (2nd edition)
    Christiansen & Torkington (O'Reilly)
  - Perl Best Practices
    Conway (O'Reilly)
  - Perl in a Nutshell
    Siever, Spainhour & Patwardhan (O'Reilly)

Perl Books

- Books you should probably look at
  - Mastering Regular Expressions
    Friedl (O'Reilly)
  - Data Munging with Perl
    Cross (Manning)
  - Advanced Perl Programming
    Cozens (O'Reilly)
  - Perl Medic
    Scott (Addison Wesley)

Perl Books

- Specialised Perl books
  - Object Oriented Perl
    Conway (Manning)
  - Programming the Perl DBI
    Descartes & Bunce (O'Reilly)
  - Writing CGI Applications with Perl
    Meltzer & Michelski (Addison Wesley)
  - Practical mod_perl
    Bekman & Cholet (O'Reilly)
  - Perl Template Toolkit
    Chamberlain, Cross & Wardley

Perl Magazines

- The Perl Review
  - http://www.theperlreview.com

That's All Folks

- Questions?