Introduction to Perl

October 2013

Document reference: 3169-2013

```
#!/usr/local/bin/perl
#
# Find all the images that have been downloaded, and how often.
# Store in a nested hash: filename -> size -> count
use strict;
use Data::Dumper;

my %downloads;                  # Predeclare the hash
while (<>) {
  next unless /Download/;       # ignore everything thats not a download
  next if /imageBar/;           # ignore downloads for the imageBar
  m!/hulton/(\w+?)/(.+\.jpg)!; # get the directory (size) and filename

  # If the filename doesn't exist, we need to create it.
  $downloads{$2} = {} unless defined $downloads{$2}

  $downloads{$2}->{$1}++;       # increment filename -> size by one
}

$Data::Dumper::Indent = 3;      # Define the padding for the output
$Data::Dumper::Sortkeys = "true"; # Sort keys (default ASCII sort)

print Dumper( \%downloads );   #  display the gathered data
```

Practical Extraction and Report Language

- Language for easy manipulation of
  - text
  - files
  - processes
- Has many similarieties to:
  - C
  - Java
  - unix shell
- Converters available for:
  - awk
  - sed
- Version 1.000 was released in 1987, version 5 came out in 1994.
  - Unicode came in 2000, with Perl 5.6
  - Perl now release a major stable version upgrade annually (5.18.0 in 2013)
- Perl 6 (Perl 5 with lots of bells and whistles) started development in Summer 2005.
  - Don't hold your breath.

Overview

- Avoids inbuilt limits - uses full virtual memory
- Many defaults - does what you want
- Easy to do from the command prompt

```
perl -e 'print "hello world\n"'
```

- Easy to do simple scripts

```
#!/usr/local/bin/perl
# a hello world script
print “hello world\n”;
```

  - use of #! for specifying processor
  - file needs to be executable

```
chmod +x hello
```

- Only has three basic types of data

Versions available for

- acorn RISCOS
- AS/400
- BeOS
- MacOS (system 7.5 onwards)
- MS Dos
- Novell Netware
- IBM OS/2
- QNX
- Windows (i386, Alpha & Win64)
- Aix / HP-UX / Irix / Digital Unix / UnixWare / etc
- Solaris / SunOS
- Linux (all platforms)
- MacOS X
- Xenix
- VMS
- BSD (Free, Net & Open)
- PDAs
- Mobile Phones (Symbian, [Jailbroken] iPhone, Android, Windows Mobile)

Scalar Variables

- Simplest kind of Perl variable
- Basis for everything else
  - scalar variable names are identified by a leading $ character
    ```
    $name
    ```
  - variable name starts with a-zA-Z then [a-zA-Z0-9_]
    ```
    $A_very_long_variable_name_ending_in_1
    ```
  - hold numeric or string data, no separate types.
    ```
    $thing = 123;
    $thing2 = “fred”;
    ```
- does not need to be predeclared.
  - Lexical scope:

```
while (expression) {
  my $variable;     # $variable is only accessible within this loop
  local $var2;      # $var2 already exists, and can get a new value within
                    # this loop, but reverts when back outside this loop
  # more stuff;
}
```

Scalar Variables - numeric

- All numeric variables stored as real internally
  - floating point literals as in C

```
-1.25
7.25e45
$num = -12e24;
```

  - integer literals

```
12
-2004
```

  - octal (base 8) constants start with leading 0

```
0377
$y = 0177;  # 0177 octal is 127 decimal
```

  - hex (base 16) constants start with leading 0x (or 0X)

```
0x7f
$y = 0x7f;  # 0x7f hex is 127 decimal
```

Scalar Variables - String

- Literal strings - single-quoted or double-quoted

```
$fred = 'hello';
$fred = "hello";
```

- No limit on string length other than memory
- Allow full 8-bit 0-255 character set
  - no terminating NULL
  - can manipulate raw binary data (eg image files)

Double quoted strings

- Delimited by double quotes
- Scanned for backslash escapes (full list in ref guide)

```
\n       : newline
\cC      : control-C
\x8      : hex 08 = backspace

$string = “hello world\n”;
```

- Scanned for variable interpolation

```
$a = “fred”;
$b = “hello $a”; # Gives “hello fred”
```

- Unassigned variable interpolates as null string

Single quoted strings

- Delimited by single quotes
- Any character except \ and ‘ is legal , including newline
  - to get ‘ or \ use a backslash to escape: \‘ and \\

```
‘don\’t’; #5 characters

‘hello
 there’; #11 chars incl newline
```

- Not scanned for variable interpolation
- Used less often than double-quoted strings
  - Good for fixed text:

```
$footer = '<div id="credit_graphics" ><a href="http://lucas.ucs.ed.ac.uk/" title="Edited
by Kiz"><!--
--><img src="/images/kiz_logo.jpg" alt="Edited by Kiz"
class="noborder"/></a><!--
--><a href="http://web.archive.org/web/*/http://lucas.ucs.ed.ac.uk"
title="Archived by the WayBack Machine since July 2000" target="_blank"><img
src="/images/wayback_logo_sm.gif" alt="Archived by the WayBack Machine since July 2000"
class="noborder"/></a><!--
--><a href="http://www.apache.org/"><img src="/icons/apache_pb2.png" alt="Powered by
Apache" title="Powered by Apache 2.2.x web server. (c) Apache Foundation" class="left"
border="0"/></a> --></div>';
```

Operators

- Numeric operators generally the same as C/Java
  `+ - / * % ** & | >> <<` (and more)

- precedence as in C
  - full list in reference guide

Operators (contd)

- Numeric or string operands
  - context is taken from operator
  - string <-> numeric conversion is automatic
  - trailing non-number is stripped from strings in conversion

```
“123” + 1 gives 124
“123zzz” + 1 gives 124
“zzz123” + 1 gives 1
```

- auto increment is smart, and can work in strings

```
$a = “zzz”;
$a++; # $a is “aaaa”
```

  - auto-increment can be too clever:

```
$foo = "1zzz"; $foo++; print $foo; # gives "2"
$foo = "zz1"; $foo++; print $foo; # gives "zz2"
```

- use WARNINGS (mentioned at the end of the workbook)

Comparison Operators

- Different sets of operators for numeric and string comparison

Comparison	Numeric	String
Equal	==	eq
Not equal	!=	ne
Less than	<	lt
Greater than	>	gt
Less than or equal to	<=	le
Greater than or equal to	>=	ge

Thus...

```
(12 lt 5) # true since string comparison
( 5 < 12) # true in numeric context
```

Smartmatch operator (since 5.10 – but see **perldoc perlsyn**)

```
"Foo" ~~ "Foo"   # true      "Foo"  ~~ "Bar"                    # false
42    ~~ 42      # true      42     ~~ "42x"                    # false
42    ~~ 42.0    # true      "Moose" ~~ [“Foo”, “Bar”, “Baz”]   # false
42    ~~ "42.0"  # true      "Moose" ~~ [qw(Foo Bar Moose Baz)]# true
```

String Operators

- Concatenation operator

```
$string = “hello “.”world” ; # gives “hello world”
                             # Note '.' not '+' !!
```

- String repetition operator

```
$string = “fred” x 3;
#gives “fredfredfred”

(3 + 2) x 4 ;
# gives “5555”. Type conversion again.
# Brackets needed as ‘x’ has precedence over ‘+’

3 + 2 x 4;
# gives “2225”. Type conversion again.
```

- Reverse

```
$string = “fred”;
$newstring = reverse($string); # Gives “derf”
```

More String Operators

- chop()
  - removes the last character from a string variable

```
$x = “hello world”;
chop($x); #x is now “hello worl”
```

- chomp()
  - Perl 5 replacement for the common use of chop in Perl 4
  - removes a newline from the end of a string variable

```
$x = “hello world”;
chomp($x); # $x is still “hello world”

$x = “hello world\n”;
chomp($x); # $x is “hello world”
```

  - default behaviour. Special variable $/ for general case

```
$x = “hello world”;
$old_newline = $/;
$/ = “d”; # We know it’s a bad idea but...
chomp ($x); # $x is “hello worl”
```

Simple Input/Output

- <STDIN> represents standard input

```
$line = <STDIN>;
# gets one line from input stream into $line
```

  - newlines exist on the end of the line, can be removed with chomp()

```
$line = <STDIN>;
chomp($line);
```

- print()
  - takes a list of arguments
  - defaults to standard output

```
print “Hello there\n”;
```

Arrays

- Second variable type
- Simply an ordered list of scalar values
- **List** is a generic term for *list literal* and *array*.
- **List literal** is comma separated sequence of scalars, in parentheses – not a variable

```
(1,2,3)
(1,"one")
() #Empty list
```

- **Array** is list of scalars – a variable number of scalars
- Array variable names are identified by a leading @ character

```
@poets = ("jonson", "donne", "herbert", "marvell");
```

- $#name contains the index of the last element of array @name

```
$lastIndex = $#poets; # $lastIndex contains 3
```

- @fred and $fred can both exist independently, separate "name space" (memory allocation)

More about arrays

- Individual scalar values addressed as $name[*n*], etc, indexed from 0

```
@fred = (7,8,9);
print $fred[0];   # prints 7
$fred[0] = 5;     # Now @fred contains (5,8,9)
```

- Arrays can be included in arrays
  - Inserted array elements are subsumed into the literal. The list is not an element.

```
@barney = @fred;
@barney = (2, @fred, “zzz”);  # @barney contains (2,5,8,9, “zzz”)
```

- Literal lists can be assigned to
  - excess values on the right hand side are discarded
  - excess values on the left hand side are set to undef.

```
($a, $b, $c) = (1,2,3);  # $a = 1, $b = 2, $c = 3
($a, $b, $c) = (1,2);    # $a = 1, $b = 2, $c = undef
($a, $b) = (1,2,3);      # $a = 1, $b = 2, 3 is ignored
```

Array slices

- A list of elements from the same array is called a slice
  - indexes are a list enclosed in square brackets
  - @ prefix since a slice is a list

```
@fred[0,1] ;           # Same as ($fred[0], $fred[1])
@fred[1,2] = (9,10);   # Same as $fred[1]= 9; $fred[2] = 10;
```

- slices also work on literal lists

```
@who = (“fred”, “barney”, “betty”, “wilma”)[1,3];
# @who contains (“barney”, “wilma” )

@who = (“fred”, “barney”, “betty”, “wilma”)[1..3];
# @who contains (“barney”, “betty”, “wilma”)
```

- Array slices count as literal lists, not arrays
- @who[0] is a one element literal list, not an array!
  - It will probably do what you want, but not for the reason you think.
  - see "Scalar and Array Context" (slide 23)

Array operators

- push() and pop()
  - add and remove items from the right hand side of an array/list (the end)

```
push(@mylist, $newvalue);    # like @mylist = (@mylist, $newvalue)
$oldvalue = pop(@mylist);    # Puts last value of @mylist in $oldvalue
```

  - pop() returns undef if given an empty list
  - push() also accepts a list of values to be pushed

```
@mylist = (1,2,3);
push(@mylist,4,5,6); # @mylist = (1,2,3,4,5,6)
```

More array operators

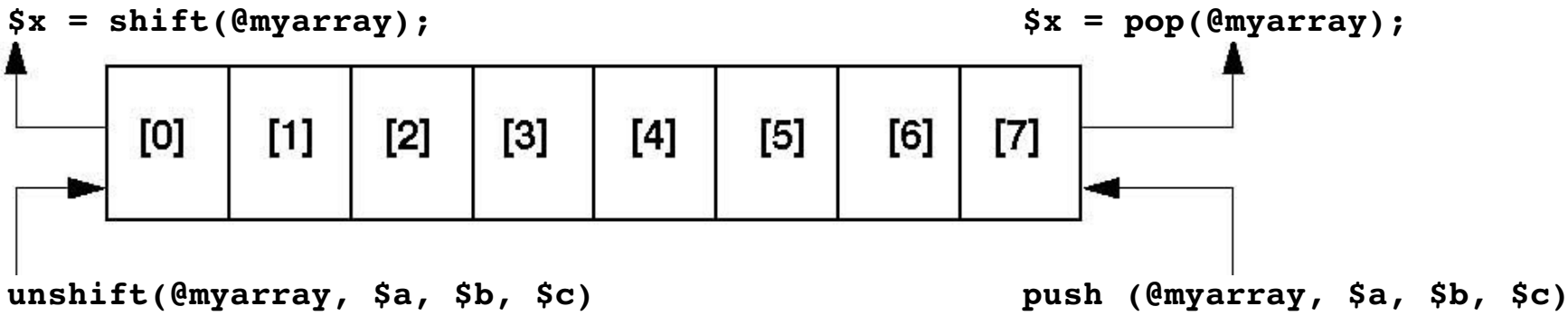
- shift() and unshift()
  - add and remove items from the left hand side of an array/list (the start)

```
unshift(@fred,$newValue); # like @fred = ($newValue, @fred)
$oldValue = shift(@fred); # like ($oldValue, @fred) = @fred
```

  - shift() returns undef if given an empty array
  - unshift() also accepts a list of values to be added

```
@mylist = (1,2,3);
unshift(@mylist,4,5,6);  # @mylist = (4,5,6,1,2,3)
```

push/pop/shift/unshift Diagram



More array operators

- reverse()
  - Reverses the order of the arguments

```
@a = (7,8,9);
@b = reverse(@a); # @b (9,8,7)
@b = reverse(1,@a); # @b (9,8,7,1)
```

- sort()
  - sorts arguments as strings in ascending ASCII order
  - default case of general sort facility

```
@x = sort("red", "green", "blue");   # gives ("blue", "green", "red")
@y = sort(1,5,10,15);                # gives (1, 10, 15, 5)
```

  - Sort can be told how to do the sorting:

```
@z = sort {$a <=> $b} @y;      # Numerical sort (ascending) - (1,5,10,15)
@z = sort {$b <=> $a} @y;      # Descending numerical sort, gives (15,10,5,1)
@x = sort {$b cmp $a}("b", "r", "g");   # gives ("r", "g", "b")
@z = sort my_sort_function @y  # Not telling you how to write functions...
```

- chomp()
  - Applies chomp to each element in an array

```
chomp (@words_in_LOTR);
```

Scalar and Array Context

- Lots of Perl is context driven
  - operands are evaluated in a scalar or array context

```
@fred = @barney;      # ordinary array assignment
$fred = @barney;
# Returns number of elements in array @barney
# $fred contains 5 (using @barney from slide 17)
```

  - Arrays and Literal Lists do different things

```
$number = @array;
# $number contains the number of elements in the array
$number = ('aa', 'bb', 'cc', 'dd', 'ee', 'ff');
$number = qw(aa bb cc dd ee ff);
# $number does not contain the number of elements in the literal list, but
# the value of the last element of the list: $number contains 'ff'
$number = @array[$n..$m];
# an array slice is considered a literal list, therefore $number contains
# the value of the last element of the slice
```

  - @all = <STDIN> is used to read all of standard input into an array

```
# Each line will be terminated with a newline
@all = <STDIN>;  # Read until End Of File
                 # from keyboard: use ctrl+d (unix) or ctrl+z(windows)
chomp(@all);
```

Control structures

- Statement block enclosed in curly brackets like C/Java
- selection (flow control)
  - if/unless statement
  - optional elsif and else blocks
    - no elsunless
- loops (iterative control)
  - while/until
  - for
  - foreach
- switch (addressed in part 2)

Control structures (contd)

- curly braces are always required (unlike C/Java).

```
if (expression) {
  statement1;
  statement2;
   ....
} elsif (expression2) {
  statement1;
  statement2;
   ....
} else if (expression3) {
    statement1;
    statement2;
     ....
  } else {
    statement1;
    statement2;
     ....
  }
}

print "Testing for quantum instabilities in sub-etheric singularities\n" if
$trekkie;
```

What is true?

- "Truth is self-evident"
- Control expressions are evaluated as a string (*scalar context*)

```
if ($thing) {
  statement;
}
```

- "" (null string) and 0 (numeric zero) are false
  - undef in a numeric context converts to 0
  - undef in a string context converts to ""

```
“0”       # converts to 0 so false
1-1       # sums to 0 so false
"1"       # not "" or 0 so true
"0e0"     # not "" or 0 so true! (0e0 is a string)
"0e0" + 0 # sums to 0, so false
undef     # converts to "" or 0 (depending on context) so false
scalar keys %my_new_hash  # there are no elements in the hash, so 0... false
```

  - From slide 12 – operators return true/false:

```
($count < $max)       # true for $count=5       & $max=10
($sring1 ne $string2) # true for $string1=”Ian” & $string2=”Alex”
($thing1 ~~ @thing2)  # true for $thing1=”a”    & @thing2=(“a”, “b”, “c”)
```

More control structures

- Iterative control structures
  - while iterates while (expression) is true

```
while ($temperature > 32) {
  statement1;
  statement2;
}
```

  - until iterates until (expression) is true

```
until ($temperature lt '3') {
  statement1;
  statement2;
}
```

  - next stops the current flow and executes the next iteration
  - last stops the current flow and exits the loop

```
my $i = 1;
while ($i > 0) {
  next if ($i%2);  # % is "modulus": returns the excess after the division
                   # 2%2 = 0; 3%2 = 1; 4%2 = 0; 8%3 = 2; 27%14 = 13; etc
  last if ($i > 101);
  print $i;
  $i++;
} # This code is flawed.... can you spot why?
```

for statement

- for statement much like C (and java)
  - for (initial_exp; test_exp; increment_exp)

```
for ($i = 1; $i <= 10; $i++) {
  print $i;
}
```

- an infinite loop

```
for (;;) {
  statement1;
  statement2;
  ...
  }
# suggest using last to exit
```

foreach statement

- Iterates through lists in order assigning to scalar variable
  - scalar variable is local to the loop

```
foreach my $i (@list) {
  statement1;
  $i += $random_number;  # Note: this modifies the actual element in @list
   ....
}
```

  - to print an array in reverse

```
foreach my $b (reverse @a) {
  print $b;
}
```

  - if scalar variable omitted defaults to $_

```
foreach (reverse @a) { # assigns to $_
  print; # prints $_
}
```

Summary

- Scalar Variables
  - numeric
  - strings - single v's double quoted
- Operators
  - numeric v's string
  - comparison and implications context
- Simple I/O
  - input from STDIN
  - print()
- Arrays
  - naming and comparing
  - slices and sizes
  - push(), pop(), shift(), unshift(), reverse() & sort()
- Control Structures
  - what is true?
  - if(), unless(), while(), until(), for() & foreach()
  - next & last

Practical Exercises

- All questions are from the Learning Perl book
  - Chapters 2, 3 & 4
  - There is no “Correct Answer” - if it works, it’s right[1]
  - Perl motto: "TIMTOWTDI" (*tim-tow-tiddy*: "There Is More Than One Way To Do It")
- Suggested way to write code:
  - write code and save to a file
  - type **`perl filename`** to run the program

For Windows users:

- click on the apps-menu, and run “cmd”
- In the window that opens, enter

  ```
  cd m:/users/trxx-yyy/Local Documents/Desktop
  ```
- reminder: **`ctrl-z`** rather than **`ctrl-d`** for “end-of-file”

---

1 Some answers are more aesthetically pleasing than others

Part 2

Associative arrays - Hashes

- Third and final variable-type.
  - “list” of indexes and associated values
  - like a two-column table

- hash variable name is identified by a leading % character

```
%hash
```

  - indexes are scalars - called keys
  - values are scalars - called values
  - individual scalar values addressed as $name{key}

```
$superheroes{“superman”} = “DC Comics”;
# creates a key “superman” in %superheroes, with value “DC Comics”

$superheroes{“Fantastic Four”} = “Marvel Comics”;
# creates a key “Fantastic Four” in %superheroes, with value “Marvel Comics”
```

- %superheroes, @superheroes and $superheroes can all exist independently

Initialising and copying hashes

- Initialising hashes
  - using a list

```
%identities = (“Superman”, “Clark Kent”, “Spiderman”, “Peter Parker”);
```

- using key and value pairs (since perl 5.001)

```
%identities = (“Superman” => “Clark Kent”, “Spiderman” => “Peter Parker”);
```

- Copying hashes
  - as hashes

```
%identities = %superheroes;   # normal way to copy whole hash
```

  - as lists

```
@ident_list = %identities;    # results in @ident_list as
              # (“Superman”, “Clark Kent”, “Spiderman”, “Peter Parker”)
              # or (“Spiderman”, “Peter Parker”, “Superman”, “Clark Kent”,)

%ident_2 = @ident_list;       # creates %ident_2 as a copy of %identities
```

- order of internal storage undefined

Hash Operators

- keys() operator
  - produces a list of keys
    - order is arbitrary

```
@list = keys(%identities);    # @list gets (“superman”, “Spiderman”)
                              #         or (“Spiderman”, “superman”)
```

    - parentheses optional

```
# once for each key of %identities
foreach $hero (keys %identities) {
  print “The secret identity of $hero is $identities{$hero}\n”;
}
```

    - in scalar context, keys() returns number of elements

```
$num_elems = keys(%identities) # $num_elems contains 2

if (keys(%superheroes)) {
  # if keys() is not zero, the hash is not empty
  statement1;
   ....
}
```

Hash Operators (contd)

- values() operator
  - produces a list of values
    - same order as keys()

```
@list = values(%identities);    # @list gets (“Peter Parker”, “Clark kent”)
                                          or (“Clark Kent”, “Peter Parker”)
```

    - parentheses optional

```
@list = values %identities;
```

Hash operators (contd)

- each() operator
  - returns each key-value pair as two-element list
  - after last element, returns empty list

```
while (($first,$last) = each(%lastnames)) {
  print “The last name of $first is $last\n”;
}
```

- delete() operator
  - removes a hash key-value pair
  - returns value of deleted element

```
%names = (“Clark” => “Kent”, “Peter” => “Parker”); # %names has two elements
$lastname = delete $names{“Clark”);                # %names has one element
```

A hash example

- Counting occurrence of words in a given array

```
# Assume @words_in_LOTR has been defined and contains all the words,
# one per cell, from the trilogy "Lord of The Rings" (there are over 12,000
# pages, so that's a lot of words!)

# predeclare the hash
my %count;

# calculate the word-count
foreach $word (@words_in_LOTR) {
  $count{ lc($word) }++;  # lc() returns the word in lower case
}

# now print out the results
foreach $word (keys %count) {
  print “$word was seen $count{$word} times\n”;
}

print "There are ".scalar (keys %count)." unique words, from a total of "
       . scalar @words_in_LOTR." words in the trilogy\n";
```

- 7 lines of code to count the words

Basic I/O : STDIN

- Input from STDIN

```
# read next input line
$line = <STDIN>;

# read rest of input until EOF (CTRL+D)
@lines = <STDIN>;    # returns undef on EOF
```

- Use of $_

```
$_ = <STDIN>;        # read next input line
```

- $_ is the default variable for input

```
<STDIN>;             # equivalent to $_ = <STDIN>;
```

- used in loops

```
while ($_ = <STDIN>) { ... };

while ( <STDIN> ) { ... };     # uses $_
```

Output to STDOUT

- print() is a list operator

```
print “hello world\n”;
print(“hello world\n”);
```

  - returns 0 or 1

```
$a = print (“hello ”, “world”, “\n”);
```

  - may need to add brackets (parentheses, or round braces)

```
print (2+3), “ hello\n”;        # wrong! prints 5,ignores “hello\n”
print ((2+3), “ hello\n”);      # right, prints 5 hello
print 2+3, “ hello\n”;          # right, prints 5 hello
```

- printf() for formatted output
  - same as C function
  - first argument - format control
  - subsequent arguments - data to be printed

```
printf “%15s %5d %10.2f\n”, $s, $n, $r;
```

  - parentheses optional

General Input/Output

- Use of file handles

```
# sample script to copy a file

open (SOURCE, “data.in”)          || die “Cannot open data.in for reading\n”;
open (OUTPUT, “>/tmp/data.out”)   || die “Cannot open output file
/tmp/data.out\n”;

while ($line = <SOURCE>) {
  # process in some way
  print OUTPUT $line;
}

close(SOURCE);
close(OUTPUT);
```

- File handles are uppercase by convention, not required

- You may also see:

```
open (my $foo, “data.in”) … ;
```

More I/O

- Options for open()

```
# equivalent to “file” for reading
open(FH1, “<file”);

# can use variable interpolation overwrites the existing file
open(FH2, “>$filename”);

# append mode output
open(FH3, “>>file”);

# Open for reading and writing pointer at the start of the file
open(LOCAL_LOG, “+<$filename”);

# open for reading and writing clear the file prior to use
open(LOCAL_LOG2, “+>file”);

# input from a pipe
open(PIPE, “ps|”);

# output to a pipe the sort does not occur until OUT is closed
open(SORTED, “|sort >>/tmp/file”);

# ensure the file is encoded correctly
open(UTF8_DATA, "<:encoding(utf8)", $filename);
```

Diamond operator

- reads lines from files specified on command line

```
$line = <>;
```

- uses @ARGV (Note: uses `shift`, so @ARGV empties in use)

```
while ($line = <>) { ... };      # an understandable form
while (    <>    ) { ... };      # a more common form (uses $_)
```

- reads from STDIN if @ARGV empty

```
while ($line = <>) { print $line; }; # “cat” command
while (    <>    ) { print;       }; # ditto
```

- can assign to @ARGV

```
@ARGV = ("The_Hobbits", "The_Fellowship_of_the_Ring",
         "The_Two_Towers", "The\ Return\ of\ the\ King");

while (<>) {                     # process the files
  push @lines_in_LOTR, $_;
}
```

  - (will say "`Can't open The_Hobbits: No such file or directory at ...`")
    - 4 lines to read three files on the disk

Regular expressions

- Pattern to match against a string
- Used in grep, sed, awk, ed, vi, emacs
- Perl uses a superset

- See the reference guide for a more complete list of options

Match operator

```
$string =~ m/pattern/
```

- matches the first occurrence of pattern
  - returns true or false

- to find "Land Rover" in various files:

```
# do “grep 'Land Rover' file1 file2 file3”
while ($line = <>) {
  if ($line =~ m/Land Rover/) {
    print “$line”;
  }
}
```

Match operator (contd)

- can use defaults
  - use $_
    ```
    m/Land Rover/;
    ```
  - omit m
    ```
    /Land Rover/;
    ```
  - thus:
    ```
    while (<>) { # using defaults
      print if /Land Rover/;
    }
    ```

Substitute operator

```
$string =~ s/pattern/string2/
```

- to replace "ipod" with "mp3 player"

```
while ($line = <>) {
  $line =~ s/ipod/mp3 player/;
  print “$line”;
}
```

- substitutes only the first occurrence of pattern
  - returns true or false
  - replaces sub strings in a given string
- Can use defaults
  - use $_

```
while (<>) { s/ipod/mp3 player/; print; }
```

- Be careful: "tripod" becomes "trmp3 player"

Patterns

- Single-character patterns
  - the character itself

```
/a/
```

  - a defined string of characters

```
/land rover/
```

  - any character (except newline)

```
/./
```

- Alternations (as opposed to alterations)

```
/Fred|Barney|Nate|Weirdly/  # Fred or Barney or Nate or Weirdly
                            # Nate Slate is Fred's boss at the quarry
                            # Weirdly Gruesome is Fred's neighbour
```

Patterns (contd)

- character classes

```
/[abcde]/;       # match a single letter
/[aeiouAEIOU]/; # match a single vowel
/[0-9]/;         # use of range
/[0-9\-]/;       # match digit or minus
/[a-zA-Z0-9_]/; # match letter, digit or
                 # underscore
```

- inverse character classes

```
/[^aeiouAEIOU]/; # match a single non-vowel
/[^0-9]/;        # match a single non-digit
/[^^]/;          # match any single character
                 # except caret
```

- Pre-defined character classes (abbreviations for)

```
\s               # whitespace  [\t\r\n\f ]
\d               # digits      [0-9]
\w               # words       [a-zA-Z0-9_]

\W               # non-words   [^a-zA-Z0-9_]
\D               # non-digits  [^0-9]
\S               # non-space   [^\r\t\n\f ]
```

Anchoring patterns

- ^ for beginning of string

```
/^fred/;   # matches fred at start of string
/^fr^ed/;  # matches fr^ed at start of string
```

- $ for end of string

```
/fred$/;   # matches fred at end of string
/$fred$/;  # matches the contents of the
           # variable $fred at end of string

/fr\$ed$/; # matches fr$ed at end of string
```

Anchoring patterns (contd)

- \b and \B for word boundaries

```
/ipod\b/;     # matches “tripod” and “ipod” but
              # not “ipodate”
/\bipod/;     # matches “ipod” and “ipodate” but
              # not “tripod”
/\bipod\B/;   # matches “ipodate” but not “ipod”
              # or “tripod”
```

Note: A boundary is defined as the bit between a **\w** character and a non **\w** character.

As **\w** excludes almost all punctuation, there are boundaries in the middle of strings like "isn't", "fred@flintstones.com", "M.I.T", and "key/value"

Note II: ipodate - *a compound $C_{12}H_{13}I_3N_2O_2$ that is administered in the form of its sodium or calcium salt for use as a radiopaque medium in cholecystography and cholangiography*

Grouping patterns

- Multipliers
  - * zero or more of preceding character (class)

```
/^\s*\w/; # matches any number of spaces at the start of a string
          # (copes with tab-indents, blank lines, etc)
```

  - **+** one or more of preceding character (class)

```
/\b\w+\b/; # a word, of any length: from 'a' to 'zenzizenzizenzic'
```

  - **?** zero or one of preceding character (class)

```
/\boff?\b/;# matches the words 'of' or 'off'
```

- Example – the pirate's cry

```
/aa?r*g+h*!?/
# aarrgghhhhh!, arrggggg!, arrrgggg
# a, ah, arh
# Note that "blag" is valid, as there is no boundary definition
# at the start of regexp
```

Matching variables

- Read-only variables
  - $& - string that matches the regular expression
  - $‘ - part of string before match
  - $’ - part of string after match

```
$_ = “this is a sample string”;

/sa.*le/; # match sample

# $‘ is “this is a “
# $& is “sample”
# $’ is “ string”
```

Avoid using them!
Whilst these are fine for simple code & debugging - once they appear in code once, they are applied to every match, which slows the performance of the program.

If you want to get $&, you would be better “capturing” – it is more efficient within Perl.

Capturing & Clustering & Cloistering

What do we do to *capture* the matched information in a regexp?

```
$foo = “1: Ian Stuart - 8097”;
$foo =~ /\b((\w+)\s+(\w+))\b/;
# $1 = “Ian Stuart”; $2 = “Ian”; $3 = “Stuart”
```

Sometimes one needs to *cluster* a group together:

```
/^cat|cow|dog$/
```

needs to be

```
/^(?:cat|cow|dog)$/
```

so that the `cat` doesn't run away with the `^`

Sometimes one needs to apply modifiers to just a small bit of the regexp (making a *cloister*):

```
/Harry (?ix: \s* [A-Z] \.? \s+ )?Truman/
# matches “Harry Truman”, “Harry S. Truman”, “Harry S Truman”,
# “Harry s. Truman”, “Harry s Truman” and even “Harry    s      Truman”
```

(See slide 57 & 58 for 'i' & 'x')

Matching operator revisited

```
$string =~ m/pattern/
```

- Operates on any string

```
$a = “hello world”;
$a =~ /^he/; # true
```

- works on any scalar

```
if (<STDIN> =~ /^[yY]/) { ... };  # input starting with Y
```

- Using different delimiter - m/.../
  - use any non-alphanumeric character

```
m/^\/usr\/etc/; # using slash
m#^/usr/etc#;   # using # for a delimeter
```

Substitution Operator revisited

```
$string =~ s/pattern/string2/
```

- works on any scalar value

```
# Change $which to “this is a quiz”
$which = “this is a quiz”;
$which =~ s/test/quiz/;

# Change an array element
$someplace[$here] = 'turn left at the roundabout';
$someplace[$here] =~ s/left/right/;

# prepend “x “ to hash element
$d{'t'} = “marks the spot”;
$d{“t”} =~ s/^/x /;
```

- Using different delimiter

```
$_ =~ s#fred#barney#;
$filename =~ s#\#/#g;
```

- Global replacement - s/pattern/string/g

```
$_ = “foot fool buffoon”;
s/foo/bar/g; # $_ becomes “bart barl bufbarn”
```

General modifiers

- Ignoring case

```
$which =~ s/TEST/quiz/i;
```

- How to react to newlines (\n)

```
# s modifier means "." can match the newline
/Perl.Programming/s   # 'I use Perl
                      # Programming is fun'

# m modifier means "^" & "$" matches the start
# and end of lines, not whole strings
s/^$editor/$author/m  # 'The editors name is
                      # Ian, and...'
```

- Faster matching

```
# Compile the pattern match once.
# Could be a problem as $speaker will not be
# changed
$person =~ /$speaker/o
```

General modifiers (continued)

- Allow whitespace and comments with /x
  - A very complex example - Find duplicate words:

```
my $input_record_separator = $/;
$/ = ""     # defines a "newline" as being a blank character, not "\n"
while (<>) {
  while (
          m!          # using '!' as the delimiter
            \b        # start at a word boundary
            (\w\S+) # capture a word-like chunk (2 or more characters)
            (
             \s+      # separated by some space
             \1       # from the thing in the braces above
            ) +       # repeatedly
            \b        # until another word boundary
           !xig       # 'x' allows spaces and comments in the regexp (as done here)
                      # 'i' to match "Is" with "is"
                      # 'g' to look for multiple matches in one paragraph
           ) {
    print "Duplicate word '$1' found in paragraph $.\n"  # the $1 matches the thing in
                                                         # the first set of braces above
  }
}
$/ = $input_record_separator;  # reset record separator
```

split() operator

```
split(/pattern/, string)
```

- splits string on each occurrence of a regular expression
- returns list of values

```
$line = “merlyn::118:10:Randal:/home/merlyn:/usr/bin/perl”;
@fields = split(/:/,$line);
# split $line on field delimeters
# now @fields becomes (“merlyn”, “”, “118”, “10”,
# “Randal” “/home/merlyn”, “/usr/bin/perl”)
```

- $_ is the default string

```
@words = split(/ /); # same as split(/ /, $_)
```

- white space is the default pattern

```
@words = split; # same as split(/\s+/, $_)
```

join() operator

```
join(string,list)
```

- glues together a list of values with a given string

```
@fields = (“merlyn”, “”, “118”, “10”, “Randal”, “/home/merlyn”,
“/usr/bin/perl”)
$newline = join(“:”,@fields);
# $newline now equals “merlyn::118:10:Randal:/home/merlyn:/usr/bin/perl”
```

- note: join uses string, not regular expression

```
$newline = join(“:”, @fields);
```

Switch statement

From Perl 5.10, the 'switch' feature was introduced

Old school:

```
for ($var) { # using $_
   /^abc/ && do { &some_stuff; last };
   /^def/ && do { &other_stuff; last };
   /^xyz/ && do { &alternative; last };
   { &nothing; };
}
```

Preferred form	`given` is “experimental”

`use v5.10.1;`
`for ($var) {`
`  when (/^abc/) { &some_stuff; }`
`  when (/^def/) { &other_stuff; }`
`  when (/^xyz/) { &alternative; }`
`  default { &nothing; }`
`}`
	`use v5.10.1;`
	`given ($var) {`
	`  when (/^abc/) { &some_stuff; }`
	`  when (/^def/) { &other_stuff; }`
	`  when (/^xyz/) { &alternative; }`
	`  default { &nothing; }`
	`}`

Perl 5.10.0 has a slightly broken `switch` implementation

Perl 5.10 to 5.14 have a “*hinky*” implementation of `given`

A final example

Analyze the words in the trilogy Lord of the Rings

```
@ARGV = ("The Fellowship of the Ring", "The Two Towers", "The Return of the King");
my %count;
my $sum;

while (<>) { # each line is in $_
  chomp;
  @words_in_line = split; # defaults to splitting on whitespace
  # Do the analysis
  foreach $word (@words_in_line) {
    $count{ lc($word) }++;  # lc() returns the word in lower case
    $sum++;
  }
}

# display the results
print "There are ".
      scalar (keys %count).
      " unique words, from a total of $sum words in the trilogy\n";

foreach $word (keys %count) {
  print "  $word was seen $count{$word} times\n";
}

# 14 lines to briefly analyze the words in the Lord of the Rings trilogy!
```

Summary

- Associative arrays - hashes
  - naming
  - initialising
  - copying
  - keys(), values(), each(), delete() operators

- Basic I/O
  - input from STDIN
  - diamond operator and @ARGV
  - output to STDOUT
  - use of filehandles
  - options to open()

Summary (contd)

- Match and subsitute operations
  - the read-only variables
  - Regular expressions
  - character classes
  - anchoring patterns
  - mutipliers
  - split() and join() operators
  - the switch statement

Diagnostics

- The -w switch gives useful diagnostics

```
perl -w filename
```

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

- warns of such things as:
  - use of variables without pre-declaration
  - re declaring localised variables
  - silent change of context
  - variables declared or set but never used

the debugger

```
perl -d filename

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

- Debugger commands

```
l       list
s       single step (goes into subroutines)
n       next statement (silently does the subroutine)
b nnn   set breakpoint at line nnn
c nnn   continue to line nnn
p ..    print scalar value
x ...   prints the contents of the thing (array, hash,etc...)
any perl statement
q quit
```

Information Resources

- Electronic
  - man pages
    ```
    man perlstyle
    ```
  - perldoc command
    ```
    perldoc Net::Ping
    perldoc perlfaq
    ```
  - Web sites
    ```
    http://www.perl.org               # The main Perl web site
    http://www.perl.com               # A "uses of Perl" blog
    http://www.perlfoundation.org/    # for the politically motivated

    http://perlmonks.org/             # help with perl
    http://www.pm.org/                # A loose association of international
                                      # Perl User Groups
    http://edinburgh.pm.org/          # Perl Mongers at Edinburgh

    http://search.cpan.org/           # CPAN (Comprehensive Perl Archive Network),
                                      # for modules
    ```

Information Resources (contd)

- Newsgroups
  - comp.lang.perl.* (announce, misc, modules, moderated & tk)

- Literature
  - O'Reilly books (http://www.oreilly.com/pub/topic/perl)
  - Manning Publications (http://www.manning.com/catalog/perl/)

Practical Exercises

- All questions are from the Learning Perl book
  - Chapters 5, 6 & 7
  - There is no “Correct Answer” - if it works, it’s right[2]
  - Perl motto: "TIMTOWTDI" (*tim-tow-tiddy*: "There Is More Than One Way To Do It")
  - Some solutions are faster or cleaner than others

For Windows users:

- click on the apps-menu, and run “cmd”
- In the window that opens, enter
  `cd m:/users/trxx-yyy/Local Documents/Desktop`
- Use **`ctrl-z`** rather than **`ctrl-d`** for “end-of-file”

For Unix users:
To email all files that start ***`my`*** to yourself, run:

```
for f in my*; do mail ad@dress < $f; done;
```

---

2 Some answers are more aesthetically pleasing than others