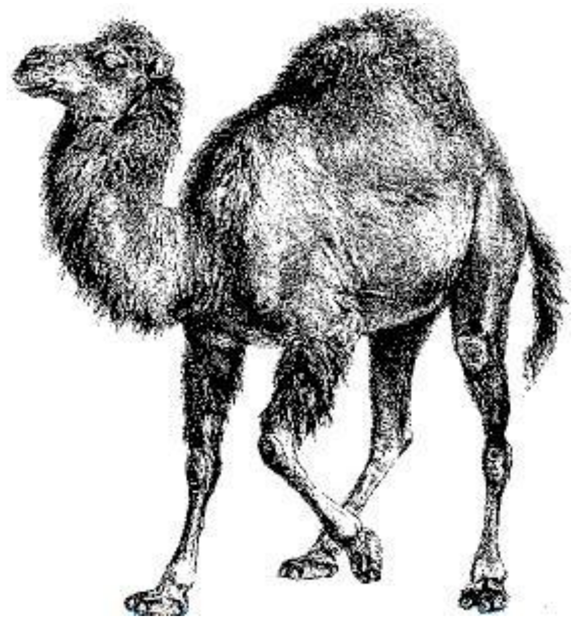
Perl Programming

Get Your Jobs Done!



Slides Contents

Introduction

Perl Poetry:

study, write, study,

do review (each word) if time.

close book. sleep? what's that?

From a graduate student (in finals week)

Introduction (1)

- PERL
  - **P**ractical **E**xtraction and **R**eport **L**anguage
  - PEARL was used by another language
  - Created by **Larry Wall** and first released in 1987
- Useful in
  - Text manipulation
  - Web development
  - Network programming
  - GUI development
  - System Prototyping
  - …anything to replace C, shell, or whatever you like

Introduction (2)

- Compiled and interpreted
  - Efficient
- Syntax Sugar
  - die unless $a == $b;
- Object oriented
- Modules
  - CPAN
- Perl6
  - http://dev.perl.org/perl6/
  - Pugs – http://www.pugscode.org/
    - /usr/ports/lang/pugs/
  - Parrot – http://www.parrotcode.org/

Introduction - Hello World (1)

- Hello World!

```
#!/usr/bin/perl –w
use strict;
# My First Perl Program
print “Hello”, “ world!\n”;
```

-
  ```
  #!/usr/bin/perl -w
  ```
  - Write down the location of perl interpreter

-
  ```
  use strict;
  ```
  - It is nice to be

-
  ```
  # My First Perl Program
  ```
  - Comment, to the end of line

-
  ```
  print(“Hello”, ” world!\n”);
  ```
  - Built-in function for output to STDOUT
- C-like “;” termination

Introduction - Hello World (2)

- hello.pl

```
#!/usr/bin/perl
print “What is your name? ”;
chomp($name = <STDIN>);
print(“Hello, $name!\n”);
```

$name = <STDIN>;
chomp $name;

chomp is not pass by value

Value interpolation into string

- scalar variable = <STDIN>
  - Read *ONE* line from standard input
- chomp
  - Remove trailing “\n” if exists
- Variables are global unless otherwise stated
- Run Perl Program

```
$ perl hello.pl                    (even no +x mode or
perl indicator)
$ ./hello.pl                  (Need +x mode and perl indicator)
```

Scalar Data

1 + 1 == 10

Scalar Data (1) - Types

- Use prefix ‘$’ in the variable name of a scalar data
  - $scalar_value
- Numerical literals
  - Perl manipulates numbers as double-decision float point values
  - Float / Integer constants, such as:
    - 1.25, -6.8, 6.23e23, 12, -8, 0377, 0xff, 0b00101100
- Strings
  - Sequence of characters
  - Single-Quoted Strings (No interpolation)
    - ‘$a\n is printed as is’, ‘don\’t’
  - Double-Quoted Strings (With interpolation)
    - “$a will be replaced by its value.\n”
    - Escape characters
      - \n, \t, \r, \f, \b, \a

Scalar Data (2) - Assignments

- Operators for assignment
  - Ordinary assignment
    - $a = 17
    - $b = “abc”
  - Short-cut assignment operators
    - Number: +=, -=, *=, /=, %=, **=
    - String: .=, x=
      - $str .= “.dat” ➔$str = $str . “.dat”
  - Auto-increment and auto-decrement
    - $a++, ++$a, $a--, --$a

Scalar Data (3) - Operators

- Operators for Numbers
  - Arithmetic
    - +, -, *, /, %, **
  - Logical comparison
    - <, <=, ==, >=, >, !=
- Operators for Strings
  - Concatenation “.”
    - “Hello” . “ ” . “world!” ➔ “Hello world!”
  - Repetition “x”
    - “abc” x 4 ➔ “abcabcabcabc”
  - Comparison
    - lt, le, eq, ge, gt, ne
- man perlop

Scalar Data (4) - Conversion

- Implicit conversion depending on the context
  - Number wanted? (3 + “15”)
    - Automatically convert to equivalent numeric value
    - Trailing nonnumeric are ignored
      - “123.45abc” ➔ 123.45
  - String wanted?
    - Automatically convert to equivalent string
    - “x” . (4 * 5) ➔”x20”

Scalar Data (5) - String Related Functions

- Find a sub-string
  - index(original-str, sub-str [,start position])

```
index(“a very long string”, “long”);  # 7
index(“a very long string”, “lame”);  # -1
index(“hello world”, “o”, 5);         # 7
index(“hello world”, “o”, 8);         # -1
```

- Sub-string
  - Substring(string, start, length)

```
substring(“a very long string”, 3, 2);       # “er”
substring(“a very long string”, -3, 3);      # “ing”
```

- Formatting data
  - sprintf (C-like sprintf)
- man perlfunc: Functions for SCALARs or strings

Branches - if / unless

- True and False
  - 0, “0”, “”, or undef are false, others are true
  - “00”, “0.00” are true, but 00, 0.00 are false
- if-elsif-else

```
if( $state == 0 ) {
       statement_1; statement_2; …; statement_n
} elsif( $state == 1 ) {
       statements;
} else {
       statements;
}
```

- unless: short cut for if (! ….)

```
unless( $weather eq “rain” ) {
       go-home;
}
```

```
if( ! $weather eq “rain” ) {
       go-home;
}
```

- print “Good-bye” if $gameOver;
- Keep_shopping() unless $money == 0;

Relational Operators

- if ($a == 1 && $b == 2) {…}
- if ($a == 1 || $b == 2) {…}
- if ($a == 1 && (! $b == 2)){…}
- if (not ($a == 1 and $b == 2) or ($c == 3)) {…}
  - not > and > or
- || has higher precedence than or, =
  - $a = $ARGV[0] || 40; # if $ARGV[0] is false, then $a = 40
  - $a = $ARGV[0] or 40; # $a = $ARGV[0]
- open XX, “file” or die “open file failure!”;
  - or can be used for statement short-cut.
- man perlop for precedence

List, Array and Hash

List

- Ordered scalars, similar to linked-list
- List literal
  - Comma-separated values
  - Ex:
    - (1, 2, 3, 4, 5,)
    - ($a, 8, 9, “hello”)
    - ($a, $b, $c) = (1, 2, 3)
    - ($a, $b) = ($b, $a) ➔ swap
- List constructor
  - Ex:
    - (1 .. 5) ➔(1,2,3,4,5)
    - (a .. z) ➔(a,b,c,d,e,…,z)
    - (1.3 .. 3.1) ➔(1,2,3)
    - ($a .. $b) ➔depend on values of $a and $b

Array (1)

- An indexed list, for random access
- Use prefix '@' in the variable name of an array
  - @ary = ("a", "b", "c")
  - @ary = qw(a b c)
  - @ary2 = @ary
  - @ary3 = (4.5, @ary2, 6.7) → (4.5, "a", "b", "c", 6.7)
  - $count = @ary3 → 5, scalar context returns the length of an array
  - $ary3[-1] → The last element of @ary3
  - $ary3[$#ary3] → $#ary3 is the last index
  - ($d, @ary4) = ($a, $b, $c) → $d = $a, @ary4 = ($b, $c)
  - ($e, @ary5) = @ary4 → $e = $b, @ary5 = ($c)

Array (2)

- Slice of array
  - Still an array, use prefix ‘@’
  - Ex:
    - @a[3] = (2)
    - @a[0,1] = (3, 5)
    - @a[1,2] = @a[0,1]
- Beyond the index
  - Access will get “undef”
    - @ary = (3, 4, 5)
    - $a = $ary[8]
  - Assignment will extend the array
    - @ary = (3, 4, 5)
    - $ary[5] = “hi” ➔ @ary = (3, 4, 5, undef, undef, “hi”)

Array (3)

- Interpolation by inserting whitespace
  - @ary = (“a”, “bb”, “ccc”, 1, 2, 3)
  - $all = “Now for @ary here!”
    - “Now for a bb ccc 1 2 3 here!”
  - $all = “Now for @ary[2,3] here!”
    - “Now for ccc 1 here!”
- Array context for file input
  - @ary = <STDIN>
    - Read multiple lines from STDIN, each element contains one line until the end of file.
  - print @ary ➔ Print the whole elements of @ary

Array (4)

Initially, @a = (1, 2);

- List or array operations
  - Push, pop and shift
    - Use array as a stack
      - push @a, 3, 4, 5 ➔ @a = (1, 2, 3, 4, 5)
      - $top = pop @a ➔ $top = 5, @a = (1, 2, 3, 4)
    - As a queue
      - $a = shift @a ➔ $a = 1, @a = (2, 3, 4)
  - Reverse list
    - Reverse the order of the elements
      - @a = reverse @a ➔ @a = (4, 3, 2)
  - Sort list
    - Sort elements as strings in ascending ASCII order
      - @a = (1, 2, 4, 8, 16, 32, 64)
      - @a = sort @a ➔ (1, 16, 2, 32, 4, 64, 8)
  - Join list
      - @a=(1,2,3); $b = join ":", @a ➔ $b = “1:2:3”

Hash (1)

- Collation of scalar data
  - An array whose elements are in <key, value> orders
  - Key is a string index, value is any scalar data
  - Use prefix “%” in the variable name of a hash
  - Ex:
    - %age = (john => 20, mary => 30,);
      ➔ same as (“john”, 20, “mary”, 30)
    - $age{john} = 21;   ➔ “john” => 21
    - %age = qw(john 20 mary 30)
    - print “$age{john} \n”

Hash (2)

%age = (john => 20, mary => 30,);

- Hash operations
  - keys
    - Yield a list of all current keys in hash
      - keys %age → ("john", "mary")
  - values
    - Yield a list of all current values in hash
      - values %age → (20, 30)
  - each
    - Return key-value pair until all elements have been accessed
      - each(%age) → ("john", 20)
      - each(%age) → ("mary", 30)
  - delete
    - Remove hash element
      - delete $age{"john"} → %age = (mary => 30)

Flow Control - while / until

- while and do-while

```
$a = 10; while ( $a  ) {  print “$a\n”; --$a }
```

```
$a = 10; print “$a\n” and --$a while $a ;
```

```
do {
        statements-of-true-part;
} while (condition);
```

- until and do-until
  - until (…) == while (! …)

```
$a = 10; until ($a == 0) { print “$a\n”; --$a }

do {
        statements-of-false-part;
} until (expression);
```

Flow Control - for / foreach

@a = (1, 2, 3, 4, 5)

- for

```
for (my $i = 0; $i <= $#a; ++$i) {
        print “$a[$i]\n”;
}
```

- foreach
  - For example:

%age = (john => 20, mary => 30,);

```
foreach $name (keys %age) {
        print “$name is $age{$name} years old.\n”;
}
```

```
for (keys %age) {
        print “$_ is $age{$_} years old.\n”;
}
```

Flow Control - last, next, redo

- Loop-control
  - last
    - Like C's "break"
  - next
    - Like C's "continue"
  - redo
    - Jump to the beginning of the current loop block <u>without revaluating the control expression</u>
    - Ex:

```
for($i=0;$i<10;$i++) { # infinite loop
        if($i == 1) {
                redo;
        }
}
```

Flow Control - Labeled Block

- Give name to block to archive “goto” purpose
- Use last, next, redo to goto any labeled block
- Example:

```
LAB1: for($i=1;$i<=3;$i++) {
 LAB2: for($j=1;$j<=3;$j++) {
  LAB3: for($k=1;$k<=3;$k++) {
   print “$i $j $k\n”;
   if(($i==1)&&($j==2)&&($k==3)) {last LAB2;}
   if(($i==2)&&($j==3)&&($k==1)) {next LAB1;}
   if(($i==3)&&($j==2)&&($k==1)) {next LAB2;}
  }
 }
}
```

```
1 1 1
1 1 2
1 1 3
1 2 1
1 2 2
1 2 3
-----
2 1 1
2 1 2
2 1 3
2 2 1
2 2 2
2 2 3
2 3 1
-----
3 1 1
3 1 2
-----
3 2 1
...
```

More on Variables

More on Variables (1) - undef

- Scalar data can be set to undef
  - $a = undef
  - $ary[2] = undef
  - $h{“aaa”} = undef
  - undef is convert to 0 in numeric, or empty string “” in string
- You can do undef on variables
  - undef $a ➔ $a = undef
  - undef @ary ➔ @ary = empty list ()
  - undef %h ➔ %h has no <key, value> pairs
- Test whether a variable is defined
  - if (defined $var) {…}

More on Variables (2) - use strict

- use strict contains
  - use strict vars
    - Need variable declaration, prevent from typo

```
use strict;
my ($x);                      # Use ‘my’ to declaration
use vars qw($y)               # Use  ‘use vars’ to declaration
```

  - use strict subs
    - Also prevent from typo, skip the details.
  - use strict refs
    - Reference type (skip)
- “no strict” to close the function
- Use –w option to enable warnings
  - Variables without initialized occur warnings

Predefined variables

- Predefined variables
  - $_ ➔ default input and pattern-searching space
  - $, ➔ output field separator for print
  - $/ ➔ input record separator (default: newline)
  - $$ ➔ pid
  - $< ➔ uid
  - $> ➔ euid
  - $0 ➔ program name (like $0 in shell-script)
  - $! ➔ errno, or the error string corresponding to the errno
  - %ENV ➔ Current environment variables (Appendix)
  - %SIG ➔ signal handlers for signals (Appendix)
  - @ARGV ➔ command line arguments (1st argument in $ARGV[0])
  - $ARGV ➔ current filename when reading from <> (Basic I/O)
  - @_ ➔ parameter list (subroutines)
  - STDIN, STDOUT, STDERR ➔ file handler names

Basic I/O

Basic I/O (1) - Input

- Using <STDIN>
  - In scalar context, return the next line or undef
  - In list context, return all remaining lines as a list, end by EOF
    - Including array and hash

```
while( $line = <STDIN>) {
        # process $line
}
while(<STDIN>) {
        # process $_
}
```

- Using diamond operator <>
  - Get data from files specified on the command line
    - $ARGV records the current filename
    - @ARGV shifts left to remove the current filename
  - Otherwise read from STDIN

Basic I/O (2) - Output

- print LIST
  - Take a list of strings and send each string to STDOUT in turn
  - A list of strings are separated by whitespace
    - Ex:
      - print(“hello”, $abc, “world\n”);
      - print “hello”, $abc, “world\n”;
      - print “hello $abc world\n”;
- Using printf
  - C-like printf
    - Ex:
      - printf “%15s, %5d, %20.2f”, $name, $int, $float;

Regular Expression

String pattern matching & substitution

Regular Expression

- String pattern
  - What is the common characteristic of the following set of strings?
  - {good boy, good girl, bad boy, goodbad girl, goodbadbad boy, …}
  - Basic regex: R_1 = “good”, R_2 = “bad” , R_3 = “boy” , R_4 = “girl”
- If R_x and R_y are regular expressions, so are the following
  - (R_x or R_y)
    - R_5= (R_1 or R_2) gives {good, bad}
    - R_6= (R_3 or R_4) gives {boy, girl}
  - (R_x . R_y)➔ R_7 =(R_5 . R_6) gives {good boy, good girl, bad boy, bad girl}
  - (R_x*) : repeat R_x as many times as you want, including 0 times
    - R_8 =R_5* gives {good, bad, goodgood, goodbad, badgood, badbad, …}
- Our final pattern is: (“good” or “bad”)* . (“boy” or “girl”)
- Regular expressions can be recognized very efficiently

Regular Expression in Perl (1)

- if ($string =~ /(good|bad)*(boy|girl)/) {…}
  - Return true if any substring of $string matches
  - /^hello$/ will match the entire string
  - if (/xxxxx/) {…} matches $_
- Match single character
  - /a/, /./, /[abc]/, /[0-9]/, /[a-zA-Z0-9]/, /[^0-9]/, /[abc\]]/
  - Predefined character class abbreviations
    - digit
      - \d ➔ [0-9]   \D ➔ [^0-9]
    - word
      - \w ➔ [a-zA-Z0-9_]   \W ➔ [^a-zA-Z0-9_]
    - whitespace
      - \s ➔ [\r\t\n\f]   \S ➔ [^ \r\t\n\f]

Regular Expression in Perl (2)

- Match more than one character
  - Multipliers
    - {m,n} ➔ m ~ n times, inclusive
    - * ➔ {0,}
    - ? ➔ {0,1}
    - + ➔ {1,}
    - {m,} ➔ >=m times.
    - {m} ➔ =m times.

```
/fo+ba?r/      # f, one or more o, b, optional a, r
/a.{5}b/       # a, any five non-newline char, b
```

Regular Expression in Perl (3)

- Grouping sub-regex by (...)
  - Besides matching, also remember the matched string for future reference
  - \1 refer to 1st grouping, \2 for 2nd, …
  - Ex:
    - /a(.*)b\1c/   # match aXYbXYc or abc, but not aXbc
- $1, $2, $3, …
  - The same value as \1, \2, \3, but can be used outside /xxx/
  - Ex:

```
$_ = “this is a test”;
/(\w+)\W+(\w+)/;        # match first two words,
                        # $1 = “this”, $2 = “is”
print “$1, $2\n”;
```

Regular Expression in Perl (4)

- $\`, $&, $’
  - Store before-matched, matched, after-matched strings
  - Ex:

```
$_ = “this is a sample string”;
/sa.*le/;                      # $` = “this is a ”
                               # $& = “sample”
                               # $’ = “ string”
```

Regular Expression in Perl (5) - Substitution

- Sed-like
  - s/pattern/replacement/
- Ex:

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

$sc = “this is a test”;
$sc =~ s/(\w+)/<$1>/g;
               # $sc = “<this> <is> <a> <test>”

$war3 = “WAR War war”;
$war3 =~ s/war/peace/gi;
               # $war3 = “peace peace peace”
```

Regular Expression in Perl (6) - Translation

- tr/search-list/replacement-list/
  - A little bit like tr command
- Ex:

```
$t = “This is a secret”;
$t =~ tr/A-Za-z/N-ZA-Mn-za-m/;
                      # rotate right 13 encrypt

$r = “bookkeeper”;
$r =~ tr/a-zA-Z//s;   # squash duplicate [a-zA-Z]
$a = “TTestt thiiis ccasse”;
$a =~ tr/Ttic/0123/s;  # $e = “0es1 1h2s 3asse”

$n = “0123456789”;
$n =~ tr/0-9/987654/d;
       # delete found but not given a
replacement
       # $n = “987654”
```

Regular Expression in Perl (7)

- Related functions
  - split(separator, string)
    - You can specify the delimit as regular expression
    - Unmatched string will form a list
    - Ex:

```
$s = “sshd:*:22:22:ssh:/var/empty:/sbin/nologin”
@fields = split(“:”, $s);
```

Subroutine

Subroutine (1)

- Definition

```
sub max {
        my ($a, $b) = @_;
        return $a if $a > $b;
        $b;
}
print &max (20, 8);
```

The value of the last statement will be returned

- Return value
  - Either single scalar value or a list value
- Arguments
  - @_ contains the subroutine's invocation arguments, and is private to the subroutine
  - $_[0], $_[1], …, $[$#_] to access individual arguments
  - Pass by value

Subroutine (2)

- Variables in subroutine
  - Private variables
    - Use “my” operator to create a list of private variables
  - Semi-private variables
    - Private, but visible within any subroutines calls in the same block
    - Use “local” to create a list of semi-private variables

```
sub add;
sub rev2 {
        local($n1, $n2) = @_;
        my ($n3) = add;
        return ($n2, $n1, $n3);
}
sub add {
        return ($n1 + $n2);
}
```

File

File (1) - open and close

- Automatically opened file handlers
  - STDIN, STDOUT, STDERR
- Open

```
open FILEHD, “filename”;      # open for read
open FILEHD, “>filename”;     # open for write
open FILEHD, “>>filename”;    # open for append
```

- Open with status checked

```
open FILEHD, “filename” || die “error-message: $!”;
```

- Use <FILEHD> to read from file handlers, just like <STDIN>
- Output ex:

```
open FH, “>>file”;
print FH ”abc”;                # output “abc” to file 
handler FH
close FH;                  # close file handler
```

File (2)

- Open with redirection
  - Open with redirection for read

```
open FD, “who |”;
```

  - Open with redirection for write
    - After the file handler closed, start the redirection.

```
open FD, “| mail –s \“Test Mail\” lwhsu@cs.nctu.edu.tw”;
close FD;
```

- Directory
  - chdir function

```
chdir “/home” || die “cannot cd to /home ($!)”;
```

  - Globbing

```
@a = </etc/host*>;
@b = glob(“/etc/host*”);                    # @a = @b
```

File (3) - File and Directory Manipulation

- unlink(filename-list) ➔ remove files

```
unlink(“data1.dat”, “hello.pl”);
unlink(“*.o”);
```

- rename(old-filename, new-filename) ➔ rename a file
- Create a link
  - link(origin, link-file) ➔ create a hard link
  - symlink(origin, link-file) ➔ create a symbolic link
- mkdir(dirname, mode) ➔ create a directory

```
mkdir(“test”, 0777);
```

- rmdir(dirname) ➔ remove a directory
- chmod(mode, filename) ➔ change file modes
- chown(UID, GID, filename) ➔ change ownership

Sort

Sort

- Without any modification, sort is based on ASCII code
- Sort by number, you can do the following

```
@list = (1, 2, 4, 8, 16, 32);
@sorted = sort {$a <=> $b} @list;
```

- You can sort by specifying your own method, defined as subroutine, use $a, $b, and return negative, 0, and positive

```
sub by_number {
    if($a < $b) {
        return 1;               # means $b, $a
    } elsif($a == $b) {
        return 0;               # means the same
    } else {
        return -1;              # means $a, $b
    }
}
```

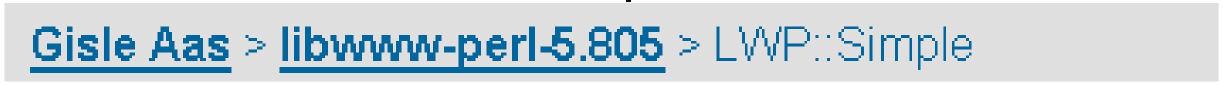
CPAN

CPAN (1)

- Comprehensive Perl Archive Network
  - http://www.cpan.org
  - http://search.cpan.org/
- 常用的五十個CPAN模組
  - http://perl.hcchien.org/app_b.html
- /usr/ports
  - p5-*
    - s/::/-/
  - Use “make search key” to find them out
- Contributing to CPAN
  - http://www.newzilla.org/programming/2005/03/16/CPAN/

CPAN (2)

- Install CPAN
  - Search the name of perl modules in CPAN



Gisle Aas > libwww-perl-5.805 > LWP::Simple

  - The LWP::Simple is in the libwww module
  - Use make search name=“p5-<name>” to find the perl module in freebsd ports tree
  - Install it
- Use CPAN
  - manual pages installed, you can use such as perldoc LWP::Simple
  - When you search the module name, the results are the same as the manual page

CPAN (3)

- A simple HTTP Proxy (with evil power!)

```
#!/usr/bin/perl

use HTTP::Proxy;
use HTTP::Recorder;

my $proxy = HTTP::Proxy->new();

# create a new HTTP::Recorder object
my $agent = new HTTP::Recorder;

# set the log file (optional)
$agent->file(“/tmp/myfile”);

# set HTTP::Recorder as the agent for the proxy
$proxy->agent($agent);

# start proxy
$proxy->start();
```

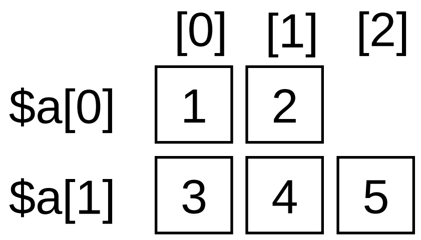
Complex Data Structure

Reference

- Create reference: store address of a variable
  - $scalarref = \$foo;
  - $arrayref = \@ARGV;
  - $hashref   = \%ENV;
  - $coderef   = \&subroutine;
- Use reference
  - $bar = $$scalarref;
  - push(@$arrayref, $filename);
  - $$arrayref[0] = "January";
  - $$hashref{"KEY"} = "VALUE";
  - &$coderef(1,2,3);

Multi-dimensional Array

- Anonymous array
  - $arrayref = [1, 2, 3];
  - $foo = $$arrayref[1];
- 2D array
  - @a = ([1, 2], [3, 4, 5]);
  - $a[0][1] == 2



- $arrayref = [1, 2, ['a', 'b', 'c']];
  - $$arrayref[2][1] == ‘b’
  - Another way to use reference by -> operator
  - $arrayref -> [2] ->[1] == ‘b’
  - $arrayref -> [2][1] == ‘b’ ➔ the 1st -> cannot be ignored

Anonymous hash

- $hashref = { john => 20, mary => 22};
  - $$hashref{john} == 20
- %student = (

```
        age => {john => 20, mary => 22},
        ident => {john => 0, mary => 1},
        NA_score => [ 99, 98 ],
);
```

- $student{age}{john} == 20
- $student{ident}{mary} == 1
- $student{NA_score}[1] == 98

Anonymous subroutine

- $coderef = sub { print "Boink $_[0]!\n" };
- &$coderef (“XD”);

Package – A different name space

```
package main;          #  the default name space
$life = 3;
package Mouse;         # switch to our package
$life = 1;
package main;          # switch back
print “$life\n”;       # shows 3
```

Perl Object Usage

- We have two files in the same directory
  - main.pl ➔ The main script, will be run as ./main.pl
  - Mouse.pm ➔ Definition of Mouse object
- In main.pl,

```
#!/usr/bin/perl –w
use Mouse;
```

Tell perl to load the object definition in Mouse.pm

```
$mouse = new Mouse( “Mickey” );
```

1. Create new object instance and store the reference to this object in $mouse
2. Pass “Mickey” to the constructor “new”

```
$mouse -> speak;
```

Call method and pass $mouse as the 1st argument

```
print “Age: ”, $mouse->{age}, "\n";
```

Perl Object Definition: Mouse.pm

```
package Mouse;

sub new {
        my $class = shift;
        my $self = { name => $_[0], age => 10, };
        bless $self, $class;
}


sub speak {
        my $self = shift;

        print “My name is ”, $self->{name}, “\n”;
}

1;
```

Class name used in creating object

Constructor

Data structure for this object

1. Associate the reference $self to the class Mouse, so we can call methods of Mouse on this reference, eg. $self->speak
2. Return the blessed reference $self

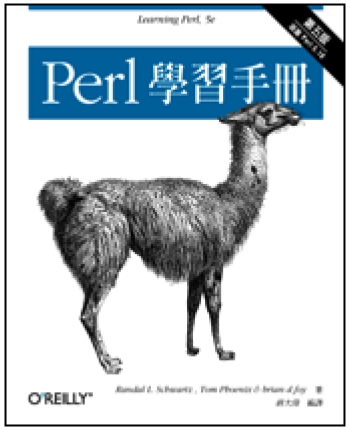
Retrieve its object data

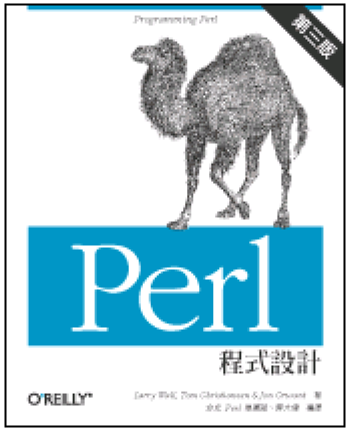
Perl module must return true at the end of script

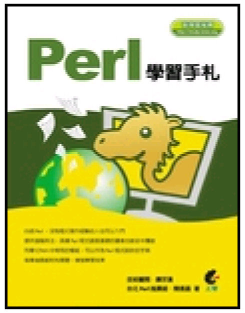
Reference Reading

Reference (1) - Document

- Book
  - Learning Perl
  - Programming Perl
  - Perl 學習手札
- Manual pages
- perldoc
  - perldoc –f PerlFunc
  - perldoc –q FAQKeywords
  - perldoc IO::Select
- Webs
  - http://perl.hcchien.org/TOC.html
  - http://linux.tnc.edu.tw/techdoc/perl_intro/
  - http://www.unix.org.ua/orelly/perl/sysadmin/







Reference (2) - manual pages

- Man Page

man perl	
man perlintro	➔ brief introduction and overview
man perlrun	➔ how to execute perl
man perldate	➔ data structure
man perlop	➔ operators and precedence
man perlsub	➔ subroutines
man perlfunc	➔ built-in functions
man perlvar	➔ predefined variables
man perlsyn	➔ syntax
man perlre	➔ regular expression
man perlopentut	➔ File I/O
man perlform	➔ Format

Reference (3) - perldoc

- Autrijus 大師的話，說 perldoc 要照下列順序讀
  - intro, toc, reftut, dsc, lol, requick, retut, boot, toot, tooc, boot, style, trap, debtut, faq[1-9]?, syn, data, op, sub, func, opentut, packtut, pod, podspec, run, diag, lexwarn, debug, var, re, ref, form, obj, tie, dbmfilter, ipc, fork, number, thrtut, othrtut, port, locale, uniintro, unicode, ebcdic, sec, mod, modlib, modstyle, modinstall, newmod, util, compile, filter, embed, debguts, xstut, xs, clib, guts, call, api, intern, iol, apio, hack.
  - 這樣讀完, 瞭解的會比 Programming Perl 要來得深入的多

Appendix

Appendix (1) - Process

- system() function
  - system() will fork a /bin/sh shell to execute the command specified in the argument
  - STDIN, STDOUT, and STDERR are inherited from the perl process

```
system(“date”);
system(“date ; who > $savehere”);
```

- Backquote
  - Execute the command and replace itself with execution result

```
foreach $_ (`who`) {
     ($who, $where, $when) = /^(\S+)\s+(\S+)\s+(.*)$/;
     print “$who on $where at $when”;
}
```

- fork() function
  - Just as fork(2)

Appendix (2) - Signals

- Catch the signal in your program
  - Using %SIG predefined hash
  - Using signal name in signal(3) without prefix “SIG” as the key
  - Ex: $SIG{‘INT’}, $SIG{‘TERM’}
  - Set the value to “DEFAULT”, “IGNORE”, or your subroutine name

```
$SIG{‘TERM’} = ‘my_TERM_catcher’;
sub my_TERM_catcher {
        print “I catch you!\n”;
}
```

- Sending the signal
  - kill(signal, pid-list)

```
kill(1, 234, 235);    # or kill(‘HUP’, 234, 235);
```

Appendix (3) - Built-in functions

- Scalars
  - chomp, chop, index, length, sprintf, substr, …
- Numeric
  - abs, exp, log, hex, int, oct, rand, sin, cos, sqrt, …
- For @ and %
  - push, pop, shift, sort, keys, values, delete
- I/O
  - open, close, read, write, print, printf, …
- Time-related
  - gmtime, localtime, time, times
- Network
  - bind, socket, accept, connect, listen, getsockopt, setsockopt, …
- User and group information
  - getpwent, setpwent, getpwuid, getpwnam, getgrent, setgrent, …

Appendix (4) - Switch

- perldoc perlsyn
  - “Basic BLOCKs and Switch Statements”

```
SWITCH: {
        /^abc/ && do { $abc = 1; last SWITCH; };
        /^def/ && do { $def = 1; last SWITCH; };
        /^xyz/ && do { $xyz = 1; last SWITCH; };
        $nothing = 1;
}
```

```
print do {
     ($flags & O_WRONLY) ? “write-only” :
     ($flags & O_RDWR)   ? “read-write” :
     "read-only";
};
```

```
SWITCH: for ($where) {
        /In Card Names/     && do { push @flags,  ‘-e’ ; last; };
        /Anywhere/          && do { push @flags,  ‘-h’ ; last; };
        /In Rulings/        && do {                     last; };
        die  “unknown value for form variable where: `$where ‘” ;
}
```

- use Switch;
  - “… after which one has switch and case. It is not as fast as it could be because it’s not really part of the language …”