Perl programming

darkx

NCTU CSCC 2014

Hello, world

```
#!/usr/bin/perl

use 5.014;
say "Hello, world!";         # say hello!
```

- Sha-bang
- perl 5.14+
- ; terminator

Use the right perl

- `perl` is **different** from *Perl*
- `/usr/ports/lang/perl5.1x` in FreeBSD
- builtin in Modern Linux distros
- `perl --version`

Why Perl?

- Scripting language
    - Making Easy Things Easy & Hard Things Possible
- `perl` interpreter: compile -> interpret
- General purpose
    - Text processing
    - Web dev
    - Networking
    - **System Administration**
    - etc ...
- Complete, mature ecosystem for Perl developers: CPAN
- Lazy! Lazy! Lazy!
- More ...

Hello, J4PH

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

use 5.014;

print "Ur name? ";
my $name = <STDIN>;          # read one line and store that into $name
chomp($name);                # remove '\n'
say "Hello, $name!";         # variable interpolation
```

- `my $variable`
- `print`
- `say`
- `chomp`
- `chop`
- `<STDIN>`

- Remember to `chmod +x` your script.

Data types

- Perl has three built-in data types:
  - `$scalars`
  - `@arrays` of scalars
  - `%hashes` (associative arrays) of scalars

```
A scalar is a
    - single string (of any size, limited only by the available memory)
    - number
    - or a reference to something

Arrays are
    - ordered lists of scalars indexed by number
    - starting with 0

Hashes are
    - unordered collections of scalar values
    - indexed by their associated string key
```

http://perldoc.perl.org/perldata.html

Data types (cont.)

- All data in Perl is a scalar, an array of scalars, or a hash of scalars.
- Variables are case-sensitive.
- Values are usually referred to by name, or through a named reference.
- The first character of the name tells you to what sort of data structure it refers.
- The rest of the name tells you the particular value to which it refers.

```
$days                # the simple scalar value "days"
$days[28]            # the 29th element of array @days
$days{'Feb'}         # the 'Feb' value from hash %days
$#days               # the last index of array @days

@days                # ($days[0], $days[1],... $days[n])
@days[3,4,5]         # same as ($days[3], $days[4], $days[5])
@days{'a','c'}       # same as ($days{'a'}, $days{'c'})

%days                # (key1, val1, key2, val2 ...)
```

http://perldoc.perl.org/perlref.html

How to read that?

```
$: the
@: these / those
%: the hash
```

- Examples:

```
$cat        # the cat
@cats       # those cats
%pets       # the hash pets
```

Context

- One of the most important concepts in Perl.
- Two major contexts: list and scalar
    - and void (which means the value has been discarded).

```perl
my $scalar = 10;
chop $scalar;
say $scalar;         # Output: 1

my @array = (11, 12, 13);
chop @array;
say @array;          # Output: 111
```

Scalar

- A scalar may contain one single value in any of three different flavors: a number, a string(, or a reference).
- Conversion from one form to another is transparent.
- Perl is a contextually polymorphic language whose scalars can be strings, numbers, or references.
- The length of an array is a scalar value.

```
my $scalar = 55;                # 55
$scalar += 0.66;                # 55.66
$scalar .= " der di yii";       # 55.66 der di yii (string concatenation)

my $s1 = "QQ";
my $s2 = 123;
my $s3 = $s1 + $s2;             # $s3: 123, the string will turn to 0
```

Scalar (cont.)

- Quote
    - Single-quoted string: no interpolation
    - Double-quoted string: with interpolation
    - Escapes
    - Quote-like operators

```
$Price = '$100';                    # not interpolated
print "The price is $Price.\n";    # interpolated
```

- Numeric literals

```
12345
12345.67
3.14_15_92          # a very important number
4_294_967_296       # underscore for legibility
0xdead_beef         # hex
0377                # octal (only numbers, begins with 0)
0b011011            # binary
```

Array & List

- List values are denoted by separating individual values by commas (and enclosing the list in parentheses where precedence requires it).
- The null list is represented by ().
- Interpolating it in a list has no effect. Thus `((),(),())` is equivalent to `()`.

```
@foo = ('cc', '-E', $bar);            # @foo contains ('cc', '-E', $bar)
$foo = ('cc', '-E', $bar);            # $foo is $bar now! Careful!
$foo = @foo;                          # $foo = 3, the length of @foo
@foo = (1, (2, 3), 4);                # same as @foo = (1, 2, 3, 4)
($a, $b, $c) = (1, 2, 3);             # list assignment
(@foo, $bar) = (1, 2, 3, 4);          # swallow! @foo = (1, 2, 3, 4)
                                      # and $bar = undef
($b, $a) = ($a, $b);                  # swap
(1 .. 5);                             # list constructor (1, 2, 3, 4, 5)
(5 .. 1);                             # ()!, use 'reverse (1 .. 5)'
```

Hash

- A hash can be initialized using a literal list holding pairs of items to be interpreted as a key and a value:
- Use the => (fat comma) operator between key/value pairs (left-hand operand could be a bareword).
- Object-oriented?!

```
# same as map assignment above
my %map = ('red', 0x00f, 'blue', 0x0f0, 'green', 0xf00);

my %map = (
    red   => 0x00f,
    blue  => 0x0f0,
    green => 0xf00,
);
```

Slices

- A slice accesses several elements of a list, an array, or a hash simultaneously using a list of subscripts.
- You can also assign to an array or hash slice.
- A slice of an empty list is still an empty list.

```
($him, $her)   = @folks[0,-1];              # array slice
@them          = @folks[0 .. 3];            # array slice
($who, $home)  = @ENV{"USER", "HOME"};      # hash slice
($uid, $dir)   = (getpwnam("daemon"))[2,7]; # list slice

@days[3..5]    = qw/Wed Thu Fri/;
@colors{'red', 'blue', 'green'} = (0xff0000, 0x0000ff, 0x00ff00);
@folks[0, -1]  = @folks[-1, 0];
```

Operators

- Arithmetic
  - `+, -, *, /, %, ++, --, **`
  - `<, <=, ==, >=, >, !=`
- Strings
  - `.`: concatenation
  - `x`: repeat
  - `lt, le, eq, ge, gt, ne`: comparison
- Logic
  - `!, ||, &&`
  - `not, or, and`
- Bitwise
  - `~, |, &, <<, >>`
- perlop

More

- Array out of range
  - Element access will get `undef`
  - Assignment will extend the array
- `chomp, chop, chr, ord, oct, hex, index, rindex, substr, sprintf, lc, uc, length, s, tr`
- `push, pop, reverse, sort, join`
- `keys, values, each, delete`
- undefine on variables

```
undef $s;   # $a = undef
undef @a;   # @a = ()
undef %h;   # %h = ()

if (defined $blah) { ... }
```

http://perldoc.perl.org/index-functions-by-cat.html

Predefined variables

- Magic!

```
$_                      # read as 'it'!
@_                      # the parameters passed to the subroutine
$"                      # list separator
$$                      # PID (same as in the shell)
$0                      # program name
$<                      # UID
$>                      # EUID
$(                      # GID
$)                      # EGID
$a, $b                  # used in 'sort'
$1, $2, $3 ...          # used in regex matched patterns
$`                      # pre-match
$&                      # matched
$'                      # post-match
$ARGV                   # current file when reading from <>
@ARGV                   # arg-list
```

Predefined variables

```
$,                       # OFS (output field separator)
$/                       # RS (input record separator)
$|                       # autoflush
$.                       # input line number
$^E                      # extended os error
$^W                      # warngins
$!                       # errno
$?                       # child return state
$@                       # eval error
%ENV                     # env
%SIG                     # signal table
@INC                     # include path
$^O                      # OS name
$^V                      # perl version


... and much more
```

http://perldoc.perl.org/perlvar.html

Control flow

- A scalar value is interpreted as `FALSE` in the Boolean sense if it is undefined, the null string or the number 0 (or its string equivalent, "0"), and `TRUE` if it is anything else.
- `{ }` are needed

```
my $s = 3;

if ($s == 1) {
    ...
}
elsif ($s == 2) {              # note! elsif!
    ...
}
else {
    ...
}

unless ($s == 1) { ... }       # if (!($s == 1))

while ($s % 2) { ... }

until ($s % 2) { ... }

say "Hello" if ($s eq "perl");
```

http://perldoc.perl.org/perlsyn.html

for: two kinds of syntaxes

```
# C-style for
for (my $i = 1; $i < 10; $i++) {
    ...
}

# foreach (use $_ if ommited)
for my $elem (@elements) {
    $elem *= 2;
}
```

- loop control (or you can use that with LABELs)

```
last                # as break in C

next                # as continue in C

redo
```

subroutines

- functions in other languages

```
sub foo {
    my ($a, $b) = @_;           # grab two args
    $a + $b;                    # the last value will be returned
}

my $ret = foo(1, 2);            # $ret = 3
```

I/O

- In scalar context, return the next line or undef.
- In list context, return all remaining lines as a list, end by EOF.

```
while( $line = <STDIN>) {
    # ...
}
while(<STDIN>) {
    # play with $_
}

print while <>;    # This is a cat!

say LIST
print LIST
printf LIST
```

File I/O

```
open FD, "<", "filename";      # read a file
open FD, ">", "filename";      # write a file
open FD, ">>", "filename";     # append to a file
open FD, "-|", "command";      # read from shell commands
open FD, "|-", "command";      # write to shell commands

close FD;
```

```
<FD>                           # read from a FD
<>                             # read from STDIN
say FD "blah";                 # write to a FD
say "blah";                    # write to STDOUT
```

http://perldoc.perl.org/functions/open.html

Regular Expression

Pattern matching

```
catabolically
catachrestically
cataclysmically
catallactically
catalytically
catarrhally
catastrophically
catawampously
catawamptiously
catchfly
catchingly
```

Pattern matching

- cat.....ly

```
my @a = `cat /usr/share/dict/words`;
for (@a) {
    print if /^cat.*ly/;
}
```

Regex

- The most powerful part of Perl!
- Understanding, creating and using regular expressions ('regexes') in Perl.
- Capture / filter whatever you want!
- RE in Perl: define a pattern.
- The UNIX utility - `g/re/p`
- libpcre: Perl Compatible Regular Expressions
- perlrequick, perlretut !!

Regular operations

- Three operations: union, concatenation, star
- A, B: languages
    - Union: `A | B = A or B`
    - Concatenation: `AB = A and then B`
    - Kleene Star: `A* = zero or more A(s)`
- Perl extents the regular expression in math.

For example

```
A = good
B = bad
C = boy
D = girl

A              # good
C              # boy
AC             # goodboy
A|B            # good or bad
(A|B)C         # goodboy or badboy
A*C            # boy, goodboy, goodgoodboy, goodgoodgoodboy ...

(A|B)(C|D)     # goodboy, goodgirl, badboy, badgirl
```

- RE brings a good representation for pattern matching.

Using RE in Perl

- using `=~` the 'binding' operator
  - `!~` the complement of `=~`
- using `[]` to define a set of elements
  - `[^]` means no in the set

```
if ($sentence =~ /the/) {      # if $sensitive matches /the/
}

if (/the/) {                   # match with $_
}

say $blah if /pattern/;        # print it if matches /pattern/
```

```
[qjk]       # Either q or j or k
[^qjk]      # Neither q nor j nor k
[a-z]       # Anything from a to z inclusive
[^a-z]      # No lower case letters
[a-zA-Z]    # Any letter
[a-z]+      # Any non-zero sequence of lower case letters
```

- Metacharacters

```
\        Quote the next metacharacter
^        Match the beginning of the line
.        Match any character (except newline)
$        Match the end of the line
|        Alternation                  -> the Union operation
()       Grouping
[]       Bracketed Character class
```

- Quantifiers

```
*           Match 0 or more times      -> the Kleene star
+           Match 1 or more times
?           Match 1 or 0 times
{n}         Match exactly n times
{n,}        Match at least n times
{n,m}       Match at least n but not more than m times
```

Examples

```
t.e     # t followed by anthing followed by e
        # This will match the
        #                 tre
        #                 tle
        # but not te
        #         tale
^f      # f at the beginning of a line
^ftp    # ftp at the beginning of a line
e$      # e at the end of a line
tle$    # tle at the end of a line
und*    # un followed by zero or more d characters
        # This will match un
        #                 und
        #                 undd
        #                 unddd (etc)
.*      # Any string without a newline. This is because
        # the . matches anything except a newline and
        # the * means zero or more of these.
^$      # A line with nothing in it.
```

Examples

```
abc         # abc (that exact character sequence, but anywhere in the
            # string)
^abc        # abc at the beginning of the string
abc$        # abc at the end of the string
ab{2,4}c    # an a followed by two, three or four b’s followed by a
            # abbc, abbbc, abbbbc

ab{2,}c     # an a followed by at least two b’s followed by a c
            # abbc, abbbc, abbbbc, abbbbbc, ...

ab*c        # an a followed by any number (zero or more) of b’s followe
            # by a c
            # ac, abc, abbc, abbbc, abbbbc, ...

ab+c        # an a followed by one or more b’s followed by a
            # abc, abbc, abbbc, abbbbc, ...
```

- charset

```
\w          # Match a "word" character (alphanumeric plus "_", plus
            #                    other connector punctuation chars plus Unicode
            #                    marks)
\W          # Match a non-"word" character
\s          # Match a whitespace character
\S          # Match a non-whitespace character
\d          # Match a decimal digit character
\D          # Match a non-digit character
```

- grouping

```
/(a|b)b/;                    # matches 'ab' or 'bb'
/(ac|b)b/;                   # matches 'acb' or 'bb'
/(^a|b)c/;                   # matches 'ac' at start of string or 'bc' anywhere
/(a|[bc])d/;                 # matches 'ad', 'bd', or 'cd'
/house(cat|)/;               # matches either 'housecat' or 'house'
/house(cat(s|)|)/;           # matches either 'housecats' or 'housecat' or
                             # 'house'.  Note groups can be nested.

/(19|20|)\d\d/;              # match years 19xx, 20xx, or the Y2K problem, xx
"20" =~ /(19|20|)\d\d/;      # matches the null alternative '()\d\d',
                             # because '20\d\d' can't match
```

grouping

```
# extract hours, minutes, seconds
if ($time =~ /(\d\d):(\d\d):(\d\d)/) {    # match hh:mm:ss format
    $hours = $1;
    $minutes = $2;
    $seconds = $3;
}


/(ab(cd|ef)((gi)|j))/;
 1  2      34

$x = "the cat in the hat";
$x =~ /^(.*)(cat)(.*)$/; # matches,
                         # $1 = 'the '
                         # $2 = 'cat'
                         # $3 = ' in the hat'
```

Search and replace

- `s/regexp/replacement/`

```
$x = "Time to feed the cat!";
$x =~ s/cat/hacker/;         # $x contains "Time to feed the hacker!"
if ($x =~ s/^(Time.*hacker)!$/$1 now!/) {
    $more_insistent = 1;
}
$y = "'quoted words'";
$y =~ s/^'(.*)'$/$1/;        # strip single quotes,
                             # $y contains "quoted words"

$x = "I batted 4 for 4";
$x =~ s/4/four/;             # doesn't do it all:
                             # $x contains "I batted four for 4"
$x = "I batted 4 for 4";
$x =~ s/4/four/g;            # does it all:
                             # $x contains "I batted four for four"
```

split

- split a scalar (string) by re

```
split /PATTERN/,EXPR,LIMIT
split /PATTERN/,EXPR
split /PATTERN/
split
```

```
#!/usr/bin/perl

use 5.014;

open PW, "<", "/etc/passwd";

while (<PW>) {
    my @arr = split /:/;
    say @arr[0,2];
}

close PW;
```

```
$ perl -nE 'my @a=split /:/; say "$a[0] $a[2]"' /etc/passwd
```

- http://perldoc.perl.org/functions/split.html

Get output from commands

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


use 5.014;

my @ping = `ping -c 5 linux1.cs.nctu.edu.tw | tail -n +2 | head -n 5`;

my $max = 0;
my $min = 1e10;
my $sum = 0;

# 64 bytes from 140.113.235.151: icmp_seq=0 ttl=52 time=16.353 ms
for my $line (@ping) {
    if ($line =~ /time=(\d*\.\d*)/) {
        $max = $1 > $max ? $1 : $max;
        $min = $1 < $min ? $1 : $min;
        $sum += $1;
    }
}


say $sum/5;
say $max;
say $min;
```

More Perlish

```
#!/usr/bin/perl


use 5.014;
use List::Util qw/sum max min/;

my @ping = `ping -c 5 linux1.cs.nctu.edu.tw | tail -n +2 | head -n 5`;

my @times = ();

# 64 bytes from 140.113.235.151: icmp_seq=0 ttl=52 time=16.353 ms
for (@ping) {
   push @times, $1 if /time=(\d*\.\d*)/;
}


say "@times";
say (sum(@times)/5);
say max @times;
say min @times;
```

Taiwan ID card No.

```
#!/usr/bin/perl

use 5.014;

while (<>) {
    chomp;
    if (length != 10) {
        say (length);
        say "must be 10 digits!";
        next;
    }
    elsif (!/^[A-Z]\d{9}$/) {
        say "wrong format!";
        next;
    }
    else {
        check($_);
    }
}
```

```
sub check {

    my $id = shift;
    my @digits = split //, $id;

    if ($digits[0] =~ /[ABCDEFGH]/) {
        $digits[0] = (ord($digits[0]) - 55);
    }
    elsif ($digits[0] =~ /[JKLMN]/) {
        $digits[0] = (ord($digits[0]) - 56);
    }
    elsif ($digits[0] =~ /[PQRSTUV]/) {
        $digits[0] = (ord($digits[0]) - 57);
    }
    elsif ($digits[0] =~ /[XYWZIO]/) {
        $digits[0] =~ y/XYWZIO/0-5/;
        $digits[0] += 30;
    }
    else {
        say "bang!";
    }

    my $sum = int($digits[0] / 10) + ($digits[0] % 10) * 9;
    $sum += $digits[$_] * (9-$_) for (1 .. 8);
    $sum += $digits[9];

    say ($sum % 10 == 0 ? "valid" : "invalid");

}
```

cpan

- Comprehensive Perl Archive Network
    - 129,527 Perl modules
    - 11,253 authors
- doc on cpan.org
- cpanminus

```
$ sudo cpanm LWP::Simple
```

- perlreftut
- http://www.youtube.com/watch?v=3C7Ngq6bM4M

Some useful CPAN modules

```
DBI
Data::Dumper
Net::SCP
Mail::Sendmail
LWP::UserAgent
WWW::Mechanize
Net::FTP
GD::Graph
Net::Telnet
Parallel::ForkManager
NetPacket::*
AnyEvent
Mojoliciou
JSON
WWW::Shorten::TinyURL
List::MoreUtils
PSGI/Plack
```

My ip

```
#!/usr/bin/perl

use 5.014;

use LWP::Simple;
my $d = get("https://www.esolutions.se/whatsmyinfo");
$d =~ /<div class="col-md-8">(\d+\.\d+\.\d+\.\d+)<\/div>/;
my $ip = $1;
say $ip;



use LWP::UserAgent;
my $ua = LWP::UserAgent->new;
$ua->agent('Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_1) AppleWebKit
/537.36 (KHTML, like Gecko) Chrome/33.0.1750.117 Safari/537.36');

$d = $ua->get("https://www.esolutions.se/whatsmyinfo");
$d->decoded_content =~ /<div class="col-md-8">(\d+\.\d+\.\d+\.\d+)<\/div>/;
$ip = $1;
say $ip;
```

youtube.pl

```
#!/usr/bin/perl

use 5.014;
no warnings;
use WWW::Mechanize;
use Getopt::Std;

our $opt_n;
getopts('n:');

my $keywords = join("+", @ARGV);
my $limit = $opt_n // 6;

if ($keywords eq "") {
    say <<EOF;
    Usage:
        ./u2b.pl keywords
        ./u2b.pl -n 3 keywords

        default n = 6
EOF
    exit;
}
```

youtube.pl

```
# new Mechanize
my $mech = WWW::Mechanize->new();

# youtube query URL
my $url = "http://www.youtube.com/results?hl=en&search_query=$keywords";
say "try to search for $limit results ...\n" . $url . "\n\n";

$mech->get( $url );

# http://www.youtube.com/watch?v=XXXXXXXXXXX
my $ref = $mech->find_all_links( url_regex => qr/watch\?v=/i );


# for all valid video links
for (@$ref) {
    if ($_->url() =~ /watch\?v=.{11}/ and $_->text() !~ /Watch Later/) {
        say $_->url_abs();
        say $_->text();
        $limit--;
    }
    last if not $limit;
}
```

xferlog parser

```
#!/usr/bin/perl
#
#
#Dec 21 17:07:08 nat235 pure-ftpd: (?@192.168.0.15) [INFO] ioi32 is now logged in

use 5.014;

system('sudo cat /var/log/xferlog | grep "logged" | grep "Dec 22" > log1');

my %table = ();

# record src IP
open F, "<", "log1";
while (<F>) {
    my @line = split;
    $table{$line[7]} //= [];
    if (not $line[5] ~~ @{$table{$line[7]}}) {
        push $table{$line[7]}, $line[5];
    }
}
close F;

for (sort keys %table) {
    say "$_ @{$table{$_}}";
}
```

socket programming

- server

```
#!/usr/bin/perl

use 5.014;
use IO::Socket;

my $server = "127.0.0.1";
my $sock = new IO::Socket::INET ( LocalHost => $server, LocalPort => 6667,
                                  Proto => 'tcp', Listen => 5, Reuse => 1)
or die "ERROR in Socket Creation : $!\n";

# accept a connection from client
while (my $client = $sock->accept()) {
    $client->autoflush(1);
    say "accept a connetion!";
    while (<$client>) {
        print $client "--> $_";
        print "--> $_";
    }
    $client->close;
}

$sock->close;
```

Any questions?

Thanks

Reference

- perldoc
- Perl Maven

Reading

- Learning Perl
- Intermediate Perl
- Perl Best Practices
- Programming Perl
- Advanced Perl Programming
- Mastering Perl
- Perl Hacks
- Perl Cookbook