Intermediate Scripting

Texas A&M High Performance Research Computing (HPRC)

Keith Jackson
Acknowledgements

• A few code examples taken from *Programming Perl* and *Beginning Perl for Bioinformatics*, as well as on-line references.
  (See hprc.tamu.edu)

• perlconsole was written by Alexis Sukrieh
  (See http://sukria.net/perlconsole.html)
Suggested Prerequisites

• HPRC account  
  (See http://hprc.tamu.edu/)
• Intro to Unix shortcourse (http://hprc.tamu.edu/shortcourses/)
• Intro to Perl shortcourse
Agenda

• Regular expressions
• Bash scripting
• Subroutines
• System calls
• Objects/modules
Binary Files

• Always use “sysopen” with “sysread” and “syswrite”
• Don’t mix “sys*” with buffered I/O functions (“< >”, “print”, etc.)
• Use “pack” and “unpack” to convert byte data to scalar variables
Read the “pack” Tutorial

- [http://perldoc.perl.org/perlpacktut.html](http://perldoc.perl.org/perlpacktut.html)
- Or: “man packtut”
- Example: “Packing and Unpacking C Structures”
- Need to copy the “\#define Pt()” macro to C stub file for determining Perl format string
typedef struct {
    char fc1; // pos 0 (& 1 byte pad)
    short fs; // pos 2
    char fc2; // pos 4 (& 3 byte pad)
    long fl; // pos 8
    float ff; // pos 16
} gappy_t;

gappy_t info; // ...
write(fh, (char *) &info, sizeof(info));
```c
#define Pt ...
Pt(gappy_t, fc1, c  );
Pt(gappy_t, fs, s!  );
Pt(gappy_t, fc2, c  );
Pt(gappy_t, fl, l!  );
Pt(gappy_t, ff, f   );
printf("total = %d\n", sizeof(gappy_t));
```

@0c @2s! @4c @8l! @16f

total = 24
my $packf = '@0c @2s! @4c @8l! @16f';
my $sz = 24; # C struct is 24 bytes
sysopen my $fh, $fname, O_RDONLY or die $!
my ($data, $count, @fields);
while ($count = sysread($fh, $data, 24)) {
    @fields = unpack($packf, $data);
    ...
}
Regular Expressions

• Regular expressions are patterns designed to concisely match a set of strings which follow the rules of the given pattern
• Regular expressions have a long history in Unix (ed, grep, vi, awk)
• Perl extends the traditional regular expressions, adding new rules
Quick Examples

$name =~ /Mich/; # Michael, Michelle, ...

$shell =~ /^[abck]sh/; # csh, ksh (not bash)

$fname !~ /.*\.[ch] $/; # not a source

$command =~ s?^?/usr/bin?; # prepend dir

$dosfile =~ tr/A-Z/a-z/; # case
## Main Regexp Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Use (return)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>qr/pattern/</code></td>
<td>• precompile pattern (regexp)</td>
</tr>
<tr>
<td><code>/pattern/</code></td>
<td>• match a pattern (success status)</td>
</tr>
<tr>
<td><code>m{pattern}</code></td>
<td>• substitute (count of replacements)</td>
</tr>
<tr>
<td><code>s/pattern/replacement/</code></td>
<td>• transliterate (count of replaced</td>
</tr>
<tr>
<td><code>s{pat}{repl}</code></td>
<td>characters)</td>
</tr>
<tr>
<td><code>tr/set1/set2/</code></td>
<td></td>
</tr>
<tr>
<td>`y</td>
<td>set1</td>
</tr>
</tbody>
</table>

Mar 24, 2017
split and grep

• **split** function divides a string using regexp to indicate separator pattern

```
split(/[:,:]/, 'a:fg:x:::2,2:3 KB');
```

• **grep** can use a regexp to test against a list

```
grep /^A.*s$/, qw(Adams Aaron Avons arts);
```
Metacharacters

\  Quote the next metacharacter

^  Match start of line

.  Match any one character

$  Match end of line

|  Alternation

( )  Grouping

[ ]  Character class

http://perldoc.perl.org/perlre.html
## Quantifiers

<table>
<thead>
<tr>
<th>Quantifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 or more times</td>
</tr>
<tr>
<td>+</td>
<td>1 or more times</td>
</tr>
<tr>
<td>?</td>
<td>1 or 0 times</td>
</tr>
<tr>
<td>{n}</td>
<td>Exactly n times</td>
</tr>
<tr>
<td>{n,}</td>
<td>At least n times</td>
</tr>
<tr>
<td>{n,m}</td>
<td>At least n but not more than m times</td>
</tr>
</tbody>
</table>

- Add a “?" after quantifier to make it not “greedy”, a “+” to force “greediness”

[http://perldoc.perl.org/perlre.html](http://perldoc.perl.org/perlre.html)
## Escape Sequences

<table>
<thead>
<tr>
<th>Escape Sequence</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\t</td>
<td>C-style control characters</td>
</tr>
<tr>
<td>\n</td>
<td></td>
</tr>
<tr>
<td>\033</td>
<td></td>
</tr>
<tr>
<td>\l</td>
<td>lowercase next character</td>
</tr>
<tr>
<td>\u</td>
<td>uppercase next character</td>
</tr>
<tr>
<td>\u</td>
<td>lowercase until \E</td>
</tr>
<tr>
<td>\U</td>
<td>uppercase until \E</td>
</tr>
<tr>
<td>\E</td>
<td>end case modification</td>
</tr>
<tr>
<td>\Q</td>
<td>quote (disable) metacharacters</td>
</tr>
<tr>
<td></td>
<td>until \E</td>
</tr>
</tbody>
</table>

[http://perldoc.perl.org/perlre.html](http://perldoc.perl.org/perlre.html)
## Character Classes

- **\w** “Word” character: `[a-zA-Z0-9_]`
- **\W** Non-“word” character: `[^a-zA-Z0-9_]`
- **\s** Whitespace character
- **\S** Non-whitespace character
- **\d** Digit character: `[0-9]`
- **\D** Non-digit character: `[^0-9]`
- **\1 \2 \3** Back references to groupings with “( )”

[http://perldoc.perl.org/perlre.html](http://perldoc.perl.org/perlre.html)
Capture Buffers

• ( ) grouping is saved in buffers \1, \2, …, or $1, $2, …

```perl
$line =~ /(^\w+) (\d+) \s*(\w+)?$/;
$name = $1; $count = $2; $optlabel = $3;

 fullname =~ s/^(\w+), (\w+)$/2 $1/?;

($fn, $ln) = ($N =~ /^\w+) (?:\w+) \(\w+\)$/;
```
Bash Scripting

• Instead of writing script in Perl, Python, etc., you have the option to write it in Bash, which also serves as your login shell

• Bash has a number of built-in features:
  – Scalars, arrays, associative arrays (hashes)
  – Control flow (if-elif-else, while, for, case, functions)
  – Regular expressions
  – Arithmetic operators
#!/bin/bash

# comment

var="Some value"
ary=(list of words here)

for (( i = 0 ; i < ${#ary[@]} ; i++ ))
  do
    printf "%3d: %s\n" $i ${ary[$i]}
  done
Assignments in Bash

- No space before or after equal sign
- If space in value, put it in quotes

```
varA=value
varB=this generates an error
varC="this does not"
varD="this interpolates $varC"
varE='this literally quotes $varD'
varF=one varG=two varH=three
```
Array Variables

- Surround values with parens
- Use curly braces to access

```bash
list=(one two three four)
list[4]=five
echo ${list[0]}
echo ${list[@]}
echo $#list[@]
for (( i = 0 ; i < $#list[@] ; i++ )); do echo $i ${list[$i]} ; done
```
Associative Arrays

• Declare with -A

declare -A office
office=([Fred]=102 [Janet]=115)
office[boss]=101
echo ${office[Janet]}
echo ${office[@]}
echo !${office[@]}
for key in ${!office[@]} ; do echo $key ${office[$key]} ; done
Arithmetic

- Use double parens
- Computation can be returned with `$( expr )`

```
i=10
(( a = i**2 ))
(( i += 83 ))
echo $(( i % 10 ))
```
Regular Expressions

• Use double square brackets
• Use =~ (as in Perl)

name="Jason Bourne"
if [[ $name =~ ^J[ae].*e$ ]] ; then echo matches ; fi
pattern='^([{[:alpha:]][[:alnum:]]*[{[:space:]}]+[{[:alpha:]}]}'
[[ $name =~ $pattern ]] && echo match || echo not
Perl References

Mar 24, 2017
References

• Perl references are scalar values which contain a pointer to:
  • another scalar
  • an array
  • a hash table
  • a subroutine
  • typeglobs
Examples of an Array Reference

@a = (9, 8, 3, 6);
$aref = @{$a};

@r = reverse @{$aref};
@s = sort @{$aref};

$third = @{$aref}[2];
$num3 = @{$aref}->[2];

same as:

@r = reverse @a;
@s = sort @a;

$third = $a[2];
$num3 = $a[2];
Making a Reference

• Perl references are created by:
  1. a backslash ("\") before a variable or subroutine, or
  2. an assignment to an “anonymous” list, hash, or code block.
References to Variables

$sc_ref = \$number;   # scalar
$ar_ref = \@namelist; # array
$hs_ref = \%lookup;   # hash
$sb_ref = \&mysub;    # subroutine
References to Anonymous

$ar_ref = [ 4, 3, 3, 7 ];       # array

$hs_ref = { m => 6, n => 9 };   # hash

$sb_ref = sub { return(shift(@_) + 1) }; # subroutine
Using a Reference

• Dereference by:
  1. using type symbol (“$”, “@”, “&”, “%”) then the reference variable in curly braces (“{ }”), or
  2. access an element by inserting “->” between reference variable and the element specifier, i.e., “[ ]” for arrays and “{ }” for hashes. For subroutines, the argument list in parentheses follows the “->”.
Bracing References

```perl
$sc_ref = \$number;    # scalar
printf("%d\n", $sc_ref);
printf("%d\n", $number);

$ar_ref = \@namelist; # array
push(@{$ar_ref}, "Harvey");
push(@namelist, "Harvey");    # Same thing
```

Same thing
Bracing References

$hs_ref = \%lookup;  # hash

@logins = keys %{$hs_ref};
@logins = keys %lookup;

Same thing

$sb_ref = \&mysub;  # subroutine

$rc = &{$sb_ref}($arg1, $arg2);
$rc = mysub($arg1, $arg2);

Same thing
Leaving Off the Braces

• You don’t always have to surround the reference variable with braces, as long as doing so doesn’t create ambiguity.

@{$ar_ref} @$ar_ref

%{$hs_ref} %$hs_ref
Bracing for Subelements

$ar_ref = \@namelist; # array

$fourth = @{$ar_ref}[3];
$fourth = $namelist[3];

Same thing

foreach $i (0..#{$ar_ref}) ... 
foreach $i (0..#$namelist) ... 

Same thing
Bracing for Subelements

\$hs\_ref = \%lookup; # hash

\$myid = \{:\$hs\_ref\}{\$login};
\$myid = \$lookup{\$login};

Same thing
Arrow Shorthand

```perl
$ar_ref = \@namelist; # array

$fourth = @{$ar_ref}[3];
$fourth = $ar_ref->[3];  # Same thing
$fourth = $namelist[3];  # Same thing

$oops   = $ar_ref[3];    # DIFFERENT!
```
Arrow Shorthand

```perl
$hs_ref = \%lookup; # hash

$myid = \${$hs_ref}\{$login\};
$myid = $hs_ref->{$login};
$myid = $lookup{$login};

$wrong = $hs_ref{$login};
```

*Same thing*

*Same thing*

*DIFFERENT!*
Arrow Shorthand

\$sb\_ref = \&my\_sub; \# subroutine

\$rc = \${\$sb\_ref}(\$arg1, \$arg2);
\$rc = \$sb\_ref\rightarrow(\$arg1, \$arg2);
\$rc = my\_sub(\$arg1, \$arg2);  \hspace{2cm} \text{Same thing}
\$wrong = \$sb\_ref(\$arg1, \$arg2);  \hspace{2cm} \text{SYNTAX ERROR}
\$wrong = sb\_ref(\$arg1, \$arg2);  \hspace{2cm} \text{different}
Breaking the 1-Dimension Barrier

• For a 2-dimensional array, create a list of references to individual lists, one per row:

```perl
@table = (
  [ 2, -1, 3 ],
  [ 0, 10, -9 ],
  [ 18, 3, 4 ],
);
```

```perl
$x = @{$table[1]}[2];
$x = $table[1]->[2];
$x = $table[1][2];
```

All are -9
Complex Data Structures

- You can nest arrays and hashes to accomplish multiple dimensions:

```perl
@info = (
    [3, "red", { Bldg => 'CSA', Floor => 1 } ],
    [7, "blue", { Bldg => 'Bright', Hrs => [8, 5] } ],
);

$start = $info[1][2]{Hrs}[0];
$flr = $info[0][2]{Floor};

$intermed = $info[1][2];  # use extra vars to simplify
$start = $intermed->{Hrs}[0];
```

Mar 24, 2017
When the Arrow is Required

• Leaving out the arrow only works between indices/keys:

```perl
@info = (
    [ 3, "red", { Bldg => 'CSA', Floor => 1 } ],
    [ 7, "blue", { Bldg => 'Bright', Hrs => [ 8, 5 ] } ],
);
$inf_ptr = \@info;
$start = $info[1][2]{Hrs}[0];
$end   = $inf_ptr->[1][2]{Hrs}[1];  # arrow required
```
Perl Subroutines

• Perl subroutines declared with “sub”
• The subroutine name follows rules of variable names
• Can leave off name to make an “anonymous” routine
• Can prototype, calls are type checked
• Sub returns a scalar or a list
Subroutine Arguments

- Arguments are a scalar list
- Arguments are not named in declaration (formal parameters) or prototype, but are put in the “@_” variable
- Contents of @ are call by reference
- Use shift(@_) and my to make local copies
A Sample Subroutine

```perl
sub add2array {
    my $val = shift @_; 
    my @newary = @_; 
    foreach my $idx (0..$#newary) {
        $newary[$idx] += $val; 
    } 
    $val = -1; 
    return @newary; 
}

$x = 5;
@orig = (1, 2, 4, 7);
@incr = add2array($x, @orig);
print "x = $x\norig = (@orig)\nincr = (@incr)\n";
```

x = 5
@orig = (1 2 4 7)
@incr = (6 7 9 12)
Prototypes

• After `sub sname`, put list of type characters in parentheses
• Backslash before symbol turns parameter into reference
• Semicolon separates mandatory from optional parameters
• Except references, only one argument (last one) can be list or hash
Sample Prototypes

sub myindex($$;$)
sub myjoin($@)
sub mypop(@)
sub mysplice(@$$@)
sub mygrep(&@)

myindex $c, $str
myindex $c, $str, $pos
myjoin ':', @items
mypad @stack
mysplice @ary, 0, 3
mysplice @ary, 0, 3, (2, 4)
mygrep { /pat/ } @lines
Prototype References

• When backslash used to indicate reference parameter, actual parameter in the call is the original type, but argument in the “@_” list is reference to the original

• You can always pass an actual reference, designated as scalar (“$”) in the prototype
Bash Functions

• Bash functions declared with “function”
• Alternatively, just give the function name followed by ()
• Delete with `unset -f`, export with `export -f`

```bash
function join() {
    local IFS="\$1"
    shift
    printf -- "\%s" "$*
}
```
Perl Functions
Perl Functions

• Perl has dozens of built-in functions
• Read descriptions at perlfunc man page, or on-line at:

http://perldoc.perl.org/index-functions.html
Queues and Stacks

• Use `push` and `shift` to implement FIFO queue
• Use `push` and `pop` to implement LIFO stack

```perl
while ($item = get_request()) {
    push(@mylist, $item);
}
# get first item from front of queue
while ($req = shift(@list)) {
    answer_request($req);
}
# get most recently added item from top of stack
while ($req = pop(@list)) {
    answer_request($req);
}
```
Sorting

• Use `sort` to order a list
• Specify your own code block to customize

```perl
@stlist = sort @namelist;

@ilist = sort { $a <=> $b } @numberlist;

@loginbyuid = sort
  { $userlist{$_}{UnixId} <=> $userlist{$b}{UnixId} }
keys %userlist;
```
Sorting With a Subroutine

• Define a sub with $a$ and $b$

```perl
sub bydatesizename {
    $mtime{a} <=> $mtime{b}
    or $filesize{a} <=> $filesize{b}
    or $filename{a} cmp $filename{b}
}

@sortfiles = sort bydatesizename @flist;

@mysorted = sort bydatesizename
    grep { $owner{$_} eq $USER } @flist;
```
Splitting and Joining

• Use a regexp for **split**
• Use a string for **join**

```
$line = "one:two:three:four";
@parts = split /:/, $line;

foreach (@parts) { s/^(.)/\u\1/ }

$newline = join("", ",", @parts);

print "newline = \'$newline'\n";
```

```
newline = `One, Two, Three, Four`
```
@surround = map { '' . $_. '' } @words;

foreach $idx (0..$#words) {
    $surround[$idx] = '' . $words[$idx] . ''; 
}

map { send_email($_) } @recipients;
Chomping the Input

- `chomp` removes newlines from end of input line

```perl
while (chomp(my $line = <STDIN>)){
    dosomething($line);
}

chomp(@lines = <$fh>);
myprocess(@lines);
$cwd = chomp(`pwd`);
```
Check Files

- shell test flags can check info on file

```bash
dosomething($myfile) if (-f $myfile);

print "cannot exec" unless (-x $myfile);
```
Perl Objects
Modules

• Perl modules are external files containing packages and symbol tables (different namespace, e.g., variable scope)
• Modules effectively implement libraries and are often done in an object-oriented fashion
• To use a given module, read its man page first for instructions
use IO::File;

my $fh = new IO::File $fname, "<" or die $!;
$input = <$fh>;
$fh->close;

use Fcntl; # get O_ constants
my $ofh = IO::File-&gt;new($outname,
    O_CREAT|O_WRONLY|O_EXCL);
print $ofh @data;
$ofh-&gt;close;
use File::stat;
use Fcntl qw(:mode); # get S_I macros

$st = stat($myfile) or die $!
next if (S_ISLNK($st->mode)); # skip if symbolic link

print "can read\n" if ($st->cando(S_IRUSR, 1));
use Getopt::Std;
my %opts = ();
getopts('of:v', %opts) or die("invalid options");
$fname = $opts{f} or $fname = 'default';
print "verbosity!\n" if ($opts{v});

$ ./myprog.pl -v -f altfile