Perl Short Course: Some Exercises to try

Duncan C. White (d.white@imperial.ac.uk)

Dept of Computing,
Imperial College London

January 2014

- Try writing a little Perl program that reads in a number from stdin, and computes the square root of that number using the following while loop, printing out a message like `"The square root of $x is $w"`.

```
my $w = $x; my $h = 1;
while ( abs($w-$h) > 0.001 )
{
      $w = ($w+$h)/2;
      $h = $x/$w;
}
```

- Hint: you might want to use `printf()` (see `perldoc -f printf`) to control the number of decimal places of the square root.
- Now wrap your "find and display a square root" logic in a for loop, displaying square roots of some range of numbers.
- Once you've read the 4th lecture (defining your own functions), turn the "find the square root of x" while loop into a separate function and call it.

- Merge **eg2** and **eg3** to sum up the leading numbers in a specific named file.
- Write a program that reads every line from stdin, lower-cases it using `lc()` and writes the lower-cased lines into a file called `lower`. You might call the program `mklower`.
- Do the same for `uc()` and `ucfirst(lc())`.
- Exercise: Write a simple one word per line translator program that reads every line from `STDIN`, chomps it, assumes it's a single word, looks it up in a hash, replaces it (if present) with the corresponding value, and prints it out. i.e. given:

```
%trans = ( "hello" => "bonjour", "my" => "mon", friend" => "ami" );
```

  translates the input (given one word per line)
  `hello duncan my friend` to `bonjour duncan mon ami`.
- Exercise: modify `mklower` slightly: remove the word `STDIN` from the `<STDIN>` getline call, leaving the mysterious syntax `<>`. The third lecture explains what this syntax means - but try to work it out for yourself experimentally with `mklower`'s behaviour.

- Build a Perl program storing `(username,roomno)` pairs in a hash, which then reads usernames - from stdin, or a text file, as you like - until the end-of-input is reached, and prints the name and corresponding room number for each.
- Then replace the set of names and their associated room numbers with an external input file, reading them in and then building the in-memory hash. (At this early stage in your Perl knowledge, you might need to store usernames and room numbers on adjacent lines in the input file).
- Next, replace the hash with a dbm file (with an initialization program that reads the names and room numbers from a text file, and stores them in the dbm-tied hash).
- Consider how to use regexes to add approximate matching for a name. One option: build a regex matching the same pattern of consonants as the target name, but replacing every vowel with `[aeiou]+`, then grep through the keys of the roomno hash for regex matches.

- Generalise several of your earlier `STDIN` or single-file processing programs to act as proper filters, receiving one or more command line arguments and opening and processing each of them in turn, defaulting to stdin if no arguments are given.
- Implement a simplified Perl version of the Unix grep command - taking two or more command line arguments (die with a nice usage message if not enough arguments!):
  - 1st argument: a literal string to search for (use `shift @ARGV` to extract and remove it).
  - All remaining arguments: filenames to search for the above string.
- Use the process-all-lines-in-all-files idiom, and print out the current filename and line if the line contains the search string.
- Count line numbers (reset each file) and print out the filename and current line number on matching lines.
- Now check that your tool works when Perl regex meta-characters are embedded (as long as your invocation from the command line single quotes the search string). How might you disable regex meta-characters?

- Prepare an input file containing a list of words, in no particular order, one per line. Write a program to open such a file - take the filename on the command line - read each line, delete leading and trailing whitespace from each line, delete leading or trailing punctuation too, and then print each line (word) out.
- Now make this word-splitter program count word frequencies - do `$freq{$word}++` for each word `$word` you find. After processing all lines, print out a sorted list of frequencies of all the words found - using magic *sort numerically by key* syntax:

```
foreach my $word (sort {$freq{$a} <=> $freq{$b}} (keys(%freq)))
```

- Now, wrap the complete make and print frequency table logic in a loop that processes each of the files named in `@ARGV` - emptying the frequency hash for each file.
- Now, modify the program so that `$ARGV[0]` contains a word to search for, and all the rest of `@ARGV` contains filenames to look in.

- For each file, first use your frequency building code to build the frequency table for that file.
- Then print out the filename if the particular word was present in the table (i.e. if the frequency of that word is more than 0!).
- This is now a primitive indexer - you give it a word and a list of filenames (all in one-word-per-line format) and it searches for the word in all the filenames.
- Now store the frequency arrays to disk, so that next time we could just use the frequency table not have to recalculate it!
- To do this, you'd need to use a Unix DBM file for each file's frequency array. It would be sensible to store the DBM files in a separate directory, to avoid cluttering up the normal directory.
- You'd also need two programs - one to index (or reindex) a list of files, and another to perform a search for a word...

- Several examples from the 2nd and 3rd session assume each line of a file contained a single word - because we didn't know how to split multiple words in a line apart - but now use `split()`):
- The general "foreach word in each line of a file" idiom is:

```
while( my $line = <STDIN> )          # each $line == sequence of words
{
   chomp $line;
   $line =~ s/^\s+//; # remove leading whitespace
   $line =~ s/\s+$//;                # remove trailing whitespace
   foreach my $word (split( /\s+/, $line))
   {
       # process $word..
   }
```

- Similarly, some examples read records - where we spread each record out across several lines, one field per line. Much nicer to have all fields on a single line, perhaps comma separated: `username,roomno`
- The general "foreach (x,y) pair in each line of a file" idiom is:

```
while( my $line = <STDIN> )          # each $line == comma-separated pair x,y
{
   chomp $line;
   my( $x, $y ) = split( /\s*,\s*/, $line);
   # process ($x,$y) pair
}
```

- Revisit such one-field-per-line programs, most obviously the `username,roomno` exercise (from lecture 3 exercises above), and use the "foreach (x,y) pair in each line of a file" idiom - or variants of it dealing with triples etc - to make them take input where usernames and room numbers are on the same line.
- Returning to the word indexer example from lecture 3:
- Use the "foreach word in each line of a file" idiom to allow the indexer to split each line into multiple words and index each word.
- Convert the indexer into separate functions, nicely laid out. Use `strict`, `warnings` and `Function::Parameters`.
- Record (perhaps in a dbm file?) when each data file was last indexed. Write a `reindex` program to check the modification time of each indexed document file and reindex modified documents.

- Hint: use Perl's `stat()` function to find a file's modification timestamp (see `perldoc -f stat` for details).
- Take any of your earlier exercises, such as the various filters you've written, and divide them up into appropriately named and commented functions.
- Familiarise yourself with complex references - use the `Data::Dumper` module to print them out.
- Modify the 4th lecture's array-of-hash example **eg14**, replacing the entire inner `foreach` loop that builds the `@x` array with a `map` invocation that begins `my @x = map ...`. If you're feeling brave, you can make the body of the outer `foreach` a single statement beginning `print join( ", ", map...`.

- In your personal Postgres database, create some new table with a few fields, perhaps storing information about People (eg name, sex, age, phone no), or information about your CD collection (if you still have one). Then write a CGI script that connects to your Postgres database, and experiment with retrieving information for the table and displaying it (perhaps as an HTML table), inserting a record into the table, querying, deleting and updating records etc. How general can you make it? could it be reused to edit a different table?
- Read HTML::Parse's documentation, work out how to extract the plain text (i.e. stuff between html tags) from a given URL fetched by LWP::Simple. First: display the plain text, minimally formatted. Second: build a frequency hash of the frequencies of each word in the plain text, and produce an ascii histogram of non-unique words sorted in descending order of frequency.
- Use Getopt::Long to merge your lower-casing and upper-casing filters into one, via an `--uppercase` flag. Could it do anything else?

- Convert (any version of) the List module's function comments into POD and add an overview section. `perldoc List` now displays the POD documentation - make List's documentation as good as other modules.
- Build a few more simple classes, either with bless or Moose (read the extra notes document), perhaps linking with Person (via "class A contains one-or-more instances of class B"). eg representing information about people, the organisations they work for, their immediate bosses etc.
- Recode some medium size collection of Java classes into Perl. Treat interfaces as stub parent classes. Familiarize yourself with Java vs Perl OO practice. Compare and contrast.
- Build another common ADT as a Perl module/class, eg. a sorted binary tree (modelled on a Haskell style recursive data declaration: *tree = leaf(string) or node(left-tree,right-tree)*. Write an in-order traversal function which makes a callback to a user-supplied function for each leaf string. Write a tree rebalance function. Implement tree sorting.

- Try implementing a functional fold/reduce operator in Perl. Reimplement sumarray using it. Using lecture 8 benchmarking info, benchmark the original sumarray (on a reasonably large list) vs the reduced version.
- Think of other situations where you could usefully embed coderefs in data structures, eg. data-driven programming, and implement one of them in Perl.
- If you can't think of such a situation, a *Delta Queue* of event functions to call at future times is extremely useful for discrete event simulations. A suitable model is a list of *(deltatime, eventfunction, eventdata)* triples, where events are enqueued in "from now delta" time order.
- Consider some design patterns from Rob Chatley's Software Engineering Design course - can they be translated into Perl?
- Think where else you could use iterators and function factories?
- Think where else you could use lazy lists (streams)?

- Given only the code snippet:

```
push @{$list{$x}}, $y;
```

  what can you deduce about the structure that `%list` represents?
- What about `$info` in this snippet:

```
$info->{A}{B}{C}[5]
```

- Use `Data::Dumper` to investigate Perl's auto-vivification via various examples. One such example might be:

```
my $ref;
push @{$ref->{A}{B}{C}}, 10
```

- Look back at programs you've written in other languages, and see if any, recoded into Perl, offer scope for Agile Data Structures techniques. Try them out.
- Benchmark some Perl snippets, eg. compare the speeds of various Perl builtins, such as string comparison (eq) versus pattern matching an anchored regex against the same string, or `x+x` vs `x*2` vs `x<<1`. Any surprises?
- Profile the biggest piece of Perl code you've written yet - perhaps the word indexer from lecture 3 (including the extensions described here in this document under lecture 4 exercises), or the lazy lists of programmers from lecture 6. Any surprises?