Perl Short Course - Third Session

Duncan C. White
dcw@doc.ic.ac.uk

- In this third session, we'll go over more of Perl in detail, with lots of examples. We'll look at:
  - **more complex file I/O**,
  - **special variables** and
  - **regular expressions**.
- But first, here's how to define a constant in Perl:

```
use constant PI => 3.1415926536;
```

Personally, I don't use constants as much as you might expect. For most purposes `$pi = 3.1415926536` is preferable.

- Now, let's start with a simple example (**eg1**):

  Exercise: Write a program that reads a list of numbers (one per line) from `STDIN`, adds all these numbers up, and prints the total.

- Hint: the idiom to use is:

```
while( $line = <STDIN> )
{
        chomp $line;
        # now process $line
}
```


A better way to run Perl Programs

- We have seen that when we want to run a Perl program called **eg1**, we say: `perl eg1`.
- Wouldn't it be better if we could just type `eg1` or `./eg1` to run our program? Then we could write our own programs in Perl, install them in a public place and let our friends run them - without them caring what language the programs are written in!
- Well, on Unix we can:
  - First, issue the Unix command:

    `chmod +x eg1`

    This makes the file executable.
  - Second, edit **eg1** and add the following line at the top:

    `#!/usr/bin/perl`

    This is a special line interpreted by Unix when it executes a non-machine code program. Unix then executes the named program (the Perl interpreter) with the script `eg1` as a command line argument. Perl starts up, reads eg1 and proceeds to run it - and then ignores the first line because it's a comment!
  - Now, you just run **eg1** by typing `./eg1`.

File Input/Output

- Perl can do much more than read from `STDIN` and write (`print`) to `STDOUT`.
- It can read the contents of named files (or Unix pipelines), overwrite a named file with new output, or append new output on the end.
- Suppose there is a file called `fred` which we wish to read. The first step is to create a filehandle like `STDIN` but connected to the file `"fred"`:

```
unless( open( FILEH, "fred" ) )
{
        # ... handle the failure...
}
```

If the file `"fred"` doesn't already exist, **open** will fail, returning `0` - a boolean false - so success can be checked by testing `if( open(...) )` and failure can be checked by testing `unless( open(...) )`.

- Often, handling the failure is done by printing an error message (on `STDERR`) and exiting - use `die`:

```
unless( open( FILEH, "fred" ) )
{
        die "can't open fred\n";
}
```

- Better still, use conditional modifier syntax:

```
die "can't open fred\n" unless open( FILEH, "fred" );
```


- Best of all, we can use the common *Do or die* Perl idiom. Remember short-circuit evaluation:

```
open( FILEH, "fred" ) || die "can't open fred!\n";
```

- Note: As well as `die` there's a function `warn` which prints a message to `STDERR` but doesn't exit.
- Now, `FILEH` is an open filehandle, just like `STDIN`, from which we can read data. Filehandles are always written in UPPERCASE in Perl.
- `$line = <FILEH>` reads the next line of input from `fred`. When there is no more input, `$line` will contain the undefined string (`undef`).
- A typical *read every line* program looks like **eg2**:

```
open( IN, "fred" ) || die "can't open fred\n";
while( $line = <IN> )
{
        chomp $line;
        print "read '$line'\n";
}
close( IN );
```

- We can also open a file for writing (replacing its previous contents with new data):

```
open( OUT, "> bob" ) || die "sob.. no bob\n";
```

- We may now write new data into `bob` by any number of:

```
print OUT "...some data...\n";
```

- Just as with reading, don't forget to `close(OUT)` when you finish writing data.
- We can append data to `bob`, instead of overwriting:

```
open( OUT, ">> bob" ) || die "sob.. no bob\n";
```

- `open` also lets us open a pipe to/from a Unix pipeline for reading/writing:
  - `open( INF, "ls | sort -r |" );`
    If the last character is a '|', then we can read data from the pipeline: Any output that `ls | sort -r` prints onto its `STDOUT` will be available for us to read via `<INF>`.
  - `open( OUTF, "| expand" );`
    If the first character is a '|', then we can write data to the pipe: `expand` will perceive a stream of data coming from its `STDIN`, but it'll really be whatever we write to `OUTF`.
- Exercise: Modify **eg1**, the sum-numbers-on-`STDIN` program, to read from a specific named file.
- Exercise: 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 such a program `mklower`.

Scalar/Array Context

- Some operators behave differently when placed in a scalar context or in an array context. An array context is one where an array is expected rather than a scalar.
- `<>` is one such operator:
  - In a scalar context it reads a single line.
  - In an array context, it reads the *entire input*, returning an array of lines.
  - So, `@line = <IN>` reads the whole of the remaining input stream into `@line`. Each line still has the newline in it.
- Fortunately, `chomp` can also be given an array variable, which chomps the newline from every array element.
- Similarly, while `@array` in an array context represents the entire array, in a scalar context it means *the number of elements in the array* - For example, `$count = @array`
- You can force a scalar context when you're not sure what Perl would do by prefixing an expression with the word `scalar` (or using it as a function).

Special Variables

Perl has many special variables (see `perldoc perlvar` for a complete list). Here are a few of the most useful:

The Argument Vector

- When you invoke one of your Perl programs, you can place *arguments* on the command line:

  eg. `myprog first second third`

- When you do this, Perl makes the strings `first`, `second` and `third` available in a special array called `@ARGV`. Specifically:

```
$ARGV[0] = "first";
$ARGV[1] = "second";
$ARGV[2] = "third";
```

- As usual, `@ARGV` evaluated in a scalar context gives the number of elements (in the example, 3).
- There is a useful function which can apply to any array, but which is very useful when applied to `@ARGV`:

```
$arg = shift @ARGV;
```

  This sets `$arg` to element 0 of the array, and removes that element from the array, shifting every other element down one space.

- Of course, it's up to the program to decide what the strings *mean*! In many programs, they may be names of files which are then opened and processed. In other cases, they may be genuine strings to use.
- The *open and process every line in every file* idiom is a very typical use of `@ARGV`:

```
foreach $arg (@ARGV)
{
        open( IN, $arg ) || next;
        while( $line = <IN> )
        {
                chomp $line;
                # now process $line
        }
        close( IN );
}
```

- In fact, the above (processing several files, not caring where one ends and the next begins) is so common that Perl has a special shorthand for it:

```
while( $line = <> )
{
        chomp $line;
        # now process $line
}
```

- Exercise: generalise one of the earlier `STDIN` or single-file processing programs to take one or more command line arguments using either of these idioms.

The Implicit Variable

- When you actually look at Perl programs you see that they're in a puzzling shorthand, seeming to omit some arguments, as here in `eg3`:

```
open( IN, "fred" ) || die "can't open fred\n";
while( <IN> )
{
        chomp;
        print "found '$_'\n" if /Dun[ck]/;
}
close( IN );
```

  - Where are we storing the line we read?
  - What are we chomping?
  - What are we matching `/Dun[ck]/` against?
  - What's that `$_` interpolated into the `print`?

- All of these are using Perl's most commonly used variable: `$_`.

- `$_` is a *default argument* to many functions:
  - The default scalar where `<>` stores its input line.
  - The default scalar that `chop` and `chomp` modify.
  - The default scalar to match a regex against.
  - The default value to print if none is given.
  - The default scalar to store each element of a `foreach`, as in `foreach (@ARGV)`.
  - The default scalar to `shift` the first element of an array into.

Environment Variables

- Unix (like most other OSes) has a concept known as *environment variables*. An environment variable is the Unix shell's equivalent of a Perl scalar - an arbitrary alphanumeric name and an arbitrary string value, created by typing special commands to a Unix shell.

  To see your current crop of environment variables, just type the command `env` at the shell command line. You will see a list of `NAME=value` pairs fly past.

- Once set, environment variables are passed around automatically to every Unix process in the current session. Perl makes these variables accessible via a single hash called `%ENV`.

- For example, an important environment variable is `HOME`. This contains the pathname of your home directory.

  To get the value of environment variable `HOME` into a Perl scalar, we could use:

```
$home = $ENV{"HOME"} || die "no home?\n";
```

- Note that other platforms – Windows, for example – also have environment variables, and that Perl on those platforms can access environment variables in the same way, but of course the semantics (what environment variables exist and what each of them actually means) are different.

Regular Expressions

- We saw in the first session that we could write:

```
if( $name =~ /^Dun[ck]/ )
```

- This is an example of matching a string against a *regular expression* (or *regex*), as found in the Unix utilities **sed, grep** and **awk**.

- A regular expression is a way of describing a class of similar strings in a very compact pattern notation. In the above example, the match will succeed if the string contained in `$name` contains:
  - A capital '`D`' as the very first character in the string,
  - The lower case letters '`u`' and '`n`' as the next two characters,
  - then *either* a lower case '`c`' or a '`k`'.
  - Any amount of junk beyond the first four characters.

- A whole regex is (usually) placed inside a pair of '`/`' signs. Within the slashes, characters are interpreted pretty much like in a double-quoted string. In particular, **variables are interpolated** before any actual pattern-matching occurs.

- A regex is made up of *single character patterns*, *grouping patterns*, *alternation patterns*, *anchoring patterns* and *bracketing patterns*.

Single character patterns

- '`.`' matches any single character.

- A single printable character matches itself (except special meta-characters like '`.`', which may be preceded by a backslash when you really want to match the character itself!).

- `[set]` matches any single char in the set.

  For example, `[aeiou]` matches any single lowercase vowel.

- Also, the set may contain items of the form `a-f`, which is a shorthand for `abcdef`.

  For example, `[a-z#%]` matches any single lowercase letter, a hash-mark, or a percent sign.

- If a set starts with a '`^`' character (as in `[^a-z#%]`), the set is negated so the pattern matches any single character NOT in the set.

- Several useful character classes are predefined:

Digit	`\d`	`[0-9]`
Non-digit	`\D`	`[^0-9]`
Word	`\w`	`[a-zA-Z0-9_]`
Non-word	`\W`	`[^a-zA-Z0-9_]`
Whitespace	`\s`	`space or tab`
Non-whitespace	`\S`	`not space or tab`

Grouping Patterns

- A sequence of adjacent single-character patterns will match a corresponding sequence of adjacent characters in the string.

  For example /[a-z]bc/ matches any lower case letter immediately followed by a 'b', immediately followed by a 'c', somewhere in the string.

- A '?' causes the preceding character pattern to be applied either zero times or once.

  For example, /he?llo/ matches 'hello' or 'hllo'.

- A '*' causes the preceding character pattern to be applied any number of times (from 0 to infinity!), in a maximally greedy fashion.

  /he*llo/ matches 'hllo', 'hello', 'heello' etc. It consumes the maximum number of 'e's possible.

- A '+' causes the preceding character pattern to be applied any number of times between 1 and infinity, again in a greedy fashion.

  For example, /he+llo/ matches 'hello', 'heello', 'heeello' etc but not 'hllo'.

- Where the greediness of '*' and '+' is a problem, you can use *? or +? to consume as few characters as possible.

- A regex can contain several of these operators:

  For example, /h[ue]*l+o/ matches 'hlo', 'hullo', 'hulllllo', 'huuulo', 'heueueueuueuellllllllo' etc.

Anchoring Patterns

- A '^' sign placed at the start of a regex matches the *start* of the string. Similarly, a $ sign at the end of a regex matches the *end* of the string.

- A '\b' pattern constrains the regex to match only at a word boundary.

Using Regular Expressions

There are two main ways of using regexes:

- To check whether a string matches a regex. We specify the string to match against using the =~ operator (or the *not match* operator !~):

  ```
  print "<$str> matches\n" if $str =~ /h[eu]*l+o/;
  ```

  If a regex match is followed by i, as in /h[eu]*l+o/i, the matching is done case insensitively.

- Secondly, a regex can be used to search and replace all occurrences of a regex within a string (again, we specify the string to modify using the =~ operator):

  ```
  $str =~ s/[aeiou]+/a/g;
  ```

  The trailing g makes Perl replace ALL vowel sequences in $str with 'a'. Without the g Perl only replaces the first match.

Note that a s/// operation returns a scalar - the number of times the string was modified.

Testing Regexes

- As a general way of testing regular expressions, I recommend a program like **eg4**:

```
#!/usr/bin/perl
#
# eg4: regex test harness..
#
print "Please enter a string: ";
$str = <STDIN>;
chomp $str;
print "\nat start  : <$str>\n";

# test search and replace:
$str =~ s/^\s+//;
print "\nafter s///: <$str>\n";

# test pattern match:
print "\n<$str> matches\n" if $str =~ /h[eu]*l+o/;
```

- This whole program exists in order to let you test search and replace and/or pattern matches using a string entered at the keyboard. By the way, s/^\s+// is a useful regex - worth committing to memory - that removes any leading whitespace. Similarly, the regex s/\s+$// removes trailing whitespace.

- I strongly recommend that you use this program to test lots of different regexes and their behaviour against various strings.

Alternation Patterns

- A regex of the form /h[eu]*llo|go+dbye|wo+tcha/ matches *either* /h[eu]*llo/, /go+dbye/ or /wo+tcha/. Note that a|b|c|g should be written as [abcg] instead for efficiency.

Bracketing Patterns

- Ordinary brackets may be placed around any complete sub-pattern, as a way of enforcing a desired precedence. For example, in /song|blackbird/ it is clear that bird is only part of the second alternative (blackbird).

- If you meant "/song|black/ followed by /bird/", then write that as /(song|black)bird/.

- In fact, if you want a repetition ('+', '*' or '?') of *anything* longer than a single character pattern, you must write something like /(hello)*/ – because /hello*/ means /hell/ followed by /o*/!

- However, brackets have another useful side effect: they tell Perl's regex engine to *remember* what text fragment matched the inner pattern for later reporting or reuse.

- For example, in the code:

```
print "found <$1>\n" if $str =~ /(song|black)bird/;
```

  If the regex matches, a special capture buffer variable called `$1` is created - *either* containing the string `song` or `black`, depending on which `$str` actually contained.

- You are allowed up to nine bracketed sub-patterns in a single pattern match - leading to the capture buffer variables `$1` to `$9` - available for use as soon as the pattern match has succeeded.

- Capture buffers may also be used in a search and replace operation like:

```
$str =~ s/^(\w+)\s+(\w+)/$2 $1/;
```

  which - if there are two or more words at the front of `$str` - swaps them.

- Another example: `/first(.*)second/` matches exactly the same strings as `/first.*second/`, but remembers the particular sequence of characters found between `first` and `second` as `$1`.

- If the string contains several occurrences of `first` and `second`, greediness will maximise the length of `/.*/` and hence match the *leftmost* occurrence of `first` and the *rightmost* occurrence of `second`:

```
.....first...first...second...first...second........
     ^^^^^!!!!!!!!!!!!!!!!!!!!!!!!!!!!^^^^^^
```


- We can also reuse a capture buffer (under a slightly different syntax `\1`) to enforce that the *same* literal text is found twice in a pattern match:

```
/first(.*)second\1/
```

- This will only match strings like:

```
..........firstXYZsecondXYZ..............
```

  but not strings like:

```
..................firstABCsecondXYZ..........
```

- Test **eg4** out with a variety of inputs and regexes and check you understand how they work.

- The match syntax `/regex/` is an abbreviation for `m/regex/`, which becomes useful if your pattern contains lots of '`/`' characters - while you can write each as '`\/`' it would be easier to change the regex quote character.

- This can be done as follows: the character immediately following the '`m`' (match) or '`s`' specifier doesn't have to be '`/`'. So for instance: `m#regex#` uses '`#`' as the quote, and `s!regex!string!` uses '`!`' as the quote.

- That's a basic overview of Perl regexes; there are loads more features (a few more bizarre ones seem to get added every year or so). See `perldoc perlre` for more details.

Another Useful operator: tr

`$str =~ tr/firstcharlist/secondcharlist/[cds]`

- `tr` is the character transliterator. It works very like the Unix filter `tr` - turning each occurrence of a character from the first character list into the corresponding character from the second character list.

- For instance, `tr/aB/Ab/` uppercases every '`a`' and lowercases every '`B`'.

- `tr` is rather like a series of regexes that only use character classes - the above example is equivalent to `s/a/A/g` followed by `s/B/b/g`. But `tr` is much more efficient.

- `tr//` is bound to a variable using the `=~` syntax (like regexes).

- Like `s///`, `tr//` also returns a scalar value - a count of how many characters were modified/deleted.

- Let's give some examples:

`$str =~ tr/A-Z/a-z/`	lowercase every character in `$str`.
`$str =~ tr/xyz/ZYX/`	turn every occurence of `x` into `Z`, `y` into `Y` and `z` into `X`.
`$str =~ tr/A-Z//d`	delete all upper case letters.
`$str =~ tr/A-Z//cd`	delete all characters *except* upper case letters.
`$str =~ tr/aeiou/V/`	replace any lower case vowel with a '`V`'.
`$str =~ tr/aeiou/V/s`	replace each *sequence* of vowels with a single '`V`'.
`$count = ($str =~ tr/a-z/a-z/)`	don't change `$str` at all, but set `$count` to the number of lower case letters found.

Extended Examples

Here are a couple of multiple-stage examples to try:

- First, implement a simplified Perl version of the Unix grep command - write a program which takes two or more command line arguments (and gives a nice usage message if it doesn't have enough!):
  - The first should be a literal string to search for (use `shift @ARGV` to extract and remove it).
  - All remaining arguments should be filenames to search.

  Use the process-all-lines-in-all-files idiom, and print out the current filename and line if the line contains the search string. Hint: recall that variable interpolation works inside a regex, i.e. `/$search/` will work unless the search string contains regex meta-characters.

- Add line number counting (reset it for each file) and print out the current line number on matching lines (as well as the filename and line itself).

- Now add in a sanity check to ensure that the search string does not contain any regex meta-characters. Die with a nice helpful message if it does (or escape them first!).

- 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 - using a hash, do `$freq{$w}++` for each word (line) `$w` you find. After processing all lines, print out a sorted list of frequencies of all the words found - using magic syntax:

  ```
  foreach $w (sort {$a <=> $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 particular word to search for, and all the rest of `@ARGV` contains filenames to look in.

- For each file, 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 (the file contents of all these must be in one-word-per-line format) and it searches for the word in all the filenames.

- It would be nice to generalise this to normal text files with multiple words per line, but you don't yet know the Perl function to use to split a string apart into pieces. Hold this over until next time!

- 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...