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, we'll look at:
  - **arrays and lists**.
  - **hashes**,
  - **special variables** (`@ARGV`, `$_`, `%ENV`) 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 `$pi = 3.1415926536` is preferable - because you can interpolate `$pi` into a string, but you can't interpolate a constant.

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` - or simply `eg1` if `.` is on your path.

Arrays and Lists

- An array is an ordered collection of scalars (strings or numbers). Just as all scalar variable names start with `$`, all array variable names start with `@`.

- A typical array such as `@fred` is *not the same as* `$fred`. Perl keeps the namespaces of arrays and scalars separate.

- An array may be built up piece by piece, using the following syntax:

  ```
  $fred[0] = "hello";
  $fred[1] = 7.1+$a;
  $fred[2] = 17.3;
  $fred[3] = $c;
  ```

  Note that:

  – Array indices start at 0, like C, C++ and Java, but unlike many other languages.

  – Each element of the array is a scalar, which is why an individual element of `@fred` is accessed using `$fred[expr]` not `@fred[expr]`. This is admittedly confusing!

- Assigning to an element beyond the current end of the array causes the array to be silently extended. All intervening elements are made the undefined value (looks just like 0).

- A single item may be extracted from an array by using `$name[expr]` on the RHS of an expression:

  ```
  $sum += $fred[$i];
  ```

  The index expression will be truncated to an integer before the array is accessed.

- Building an array piece by piece is a bit of a pain. You can write a comma-separated list of scalar expressions inside a pair of brackets:

  ```
  ( "hello", 7.1+$a, 17.3, $c )
  ```

- Such a literal can be assigned straight into an array variable (notice the `@` this time):

  ```
  @fred = ( "hello", 7.1+$a, 17.3, $c );
  ```

- Inside a list, the `..` operator can be used as in `( 1..20 )` or `( 'a'..'z' )`.

- If you have a list of single words, Perl provides an special convenient syntax. Instead of:

  ```
  @fred = ( "hello", "there", "how", "are", "you" );
  ```

  you can simply type:

  ```
  @fred = qw(hello there how are you);
  ```

- You can also break up an array into a list:

  ```
  ( $a, $b, $c ) = @fred;
  ```

  This copies the element `$fred[0]` to `$a`, `$fred[1]` to `$b` and so on. Any remaining elements (beyond the first 3 in this example) are quietly ignored. If the array contains too few elements, then the remaining variables get set to the undefined value.

- An array can be used to soak up the remainder:

  ```
  ( $a, $b, @c ) = @fred;
  ```

- This gives you a very easy swap operation. In most languages, a temporary variable `t` is needed, as in the C code:

  ```
  t = x;
  x = y;
  y = t;
  ```

- In Perl, you can write:

  ```
  ( $x, $y ) = ( $y, $x );
  ```

  which takes `y` and `x`, forms them into a two-element list, and assigns the first two elements of that list back into `x` and `y`.

- In this respect, Perl arrays also act as Prolog-style tuples, where similar things are possible.

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 @fred in an array context represents the entire array, in a scalar context it means *the number of elements in the array* - For example, $count = @fred

- 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).

Hashes - Associative Arrays

- Hash variable names start with `%`. Again, Perl is capable of keeping a hash called `%fred` entirely distinct from `$fred` and `@fred` - they all occupy separate namespaces.

- A hash *associates* one arbitrary scalar (the *value*) with another scalar (the *key*).

- A hash literal can be written as a list of pairs with the special => operator:

```
%roomno = (
     "dcw"  => "225",
     "sza"  => "226",
     "iwm"  => "220"
);
```

- The first element in each pair is the key, and the second is the value.

- A hash is really like a two-column database table, indexed on the key field:

Key	Value
dcw	225
sza	226
iwm	220
....	....

- Hashes have a highly efficient internal indexing system (a hash table, hence the name) built in, so that you can look up a particular pair by it's key very quickly.

- To add a single *(key, value)* pair into a hash, do:

```
$roomno{"susan"} = "569";
```

- As a convenience, Perl allows you to omit the quotes on a literal key string, and write:

```
$roomno{susan} = "569";
```

- To check whether a key exists in the hash, use:

```
if( exists $roomno{elvis} ) ...
```

- To delete a single (key, value) pair from a hash:

```
delete $roomno{dcw};
```

- Alternatively, the entire hash may be cleared by:

```
%roomno = ();
```

- Our original hash literal example could be written as:

```
%roomno = ();
$roomno{dcw}  = "225";
$roomno{sza}  = "226";
$roomno{iwm}  = "220";
```

- To retrieve a particular value from a hash, use `$room = $roomno{$person}`

  If no value has been stored against the key `$person`, the undefined value is returned.

- To process an entire hash, you can use the **keys()** function:

  ```
  foreach $k (keys(%roomno))
  {
      $v = $roomno{$k};
      print "$k in room $v\n";
  }
  ```

- Note that `keys(%roomno)` actually builds an ordinary array containing the keys of `%roomno`. This is potentially huge!

- Note also that the keys do not come out in any predefined order - certainly not alphabetical order, or insertion order! They come out in an efficient internal order! Because of this, you often see:

  ```
  foreach $k (sort keys(%roomno))
  ...
  ```

- Very occasionally, you only need the **values**:

  ```
  foreach $v (values(%roomno))
  {
      print "room value $v\n";
  }
  ```

- If you need both keys and values, the preferable (most efficient) loop uses the **each()** function and a **while** loop which iterates over all the (key, value) pairs:

```
while( ($k,$v) = each(%roomno) )
{
    print "$k in room $v\n";
}
```

- Once again, let us stress that **keys(), values()** and **each()** produces results in whatever *order* Perl likes. You cannot rely on any particular order - in particular, it will not be in sorted order, or in the order items were inserted!

- Exercise: Try building yourself a Perl program that has a set of (username, roomno) pairs in a hash, and then repeatedly reads usernames - from stdin, or a text file, as you like - until the end-of-input is reached, looking up the corresponding room number for each name and printing it out.

- Then replace the set of names and their associated room numbers with an external data 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 file).

- Next, replace the hash altogether with a dbm file (with a separate initialization program that reads the names and room nos from a text file, and stores them in the dbm-tied hash).

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 `eg1`:

```
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: `$_`, the *implicit* 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 array element 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 current value of `$name` starts with:
  - A capital '`D`' [must be the very first character],
  - The lower case letters '`u`' and '`n`' as the next two characters,
  - then *either* a lower case '`c`' or a '`k`'.

- 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 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 lower-case 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 lower-case 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`', '`heeelo`', '`heueueueuueuelllllllo`' 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.
- Without any anchoring, the regex can match anywhere in the string.

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.

Testing Regexes

- As a general way of testing regular expressions, I recommend a program like **eg2**:

```
#!/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

- Brackets may be placed around any complete subpattern, 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 need brackets, as in `/(hello)*/` – because `/hello*/` means `/hell/` followed by `/o*/`!

- 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/^\s*(\w+)\s+(\w+)/$2 $1/;
  ```

  which - if there are two or more space separated 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 **eg2** 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...