Introduction to Perl: First Lecture

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

Dept of Computing,
Imperial College London

January 2015

- This short practical course introduces you to Larry Wall's immensely flexible **Perl** programming language.
- It consists of a series of 8 one-hour lectures, with slides and practical examples.
- The course materials for the course can be found at:
  `http://www.doc.ic.ac.uk/~dcw/perl2014/`
  - This first lecture gives a general introduction to Perl, a fast and shallow scan across most of the important features.
  - The second, third and fourth lectures goes over Perl in more detail, introducing most interesting features, including **arrays**, **hashes**, **functions**, Perl's **standard library** and **references**.
  - The fifth and sixth lectures describe the **CPAN** module archive, using and writing **modules**, **objects and classes**.
  - The seventh and eighth lectures cover some more advanced Perl topics - **higher-order functions**, **program transformations**, **data structures**, **testing** and **benchmarking**.
- There are two good books describing Perl: Randal Schwartz's excellent introduction **Learning Perl** and Larry Wall and Randal Schwartz's rather more advanced **Programming Perl.**

- Perl was designed by a busy sysadmin - someone who needs a powerful language in which programs can be written, tested and deployed quickly to solve urgent problems.
- Such programs range from "Perl one liners" typed direct on the command line as part of a pipeline, to well-designed and engineered large modular programs.
- Perl, like Python and Ruby, is a "scripting language" which gives us great **leverage** in writing programs when compared to languages like C, C++ or Java. Perl lets you program at a very high level, with powerful tools and data structures.
- Perl borrows features from many other languages - forming a coherent whole, more powerful than the sum of its parts.
- Perl is known as the **Swiss Army Chainsaw** of programming; it makes *the easy tasks easy, the hard tasks possible*.

- **Control structures**: from C and the shell.
- **Expression syntax**: mainly from C, with several operators brought in from the shell – eg file test operators.
- **Powerful data types** built-in: dynamic **arrays, lists, stacks, queues, tuples** and **hashes (dictionaries)**, very easy to use.
- **Regular expressions** built-in: coming from **sed** and **grep**, but significantly extended to make them even more powerful.
- **Build filters** easily: **manipulate files, processes** and **command line arguments** simply.
- **Storage management**: done for us - like most scripting languages, Java (but very unlike C!).
- Plus: threads, portable graphics, OOP, functional programming, network programming and more modules than you can count.

The main ways Perl does storage management are as follows:

- **Strings are a basic type** - not arrays of characters as they are in C. Strings **grow as needed** - and they can be enormous!
- **Arrays grow as needed** - simply assign to an element and the array extends to include that element automatically.
- Arrays also provide **unlimited length Prolog/Haskell-style lists**. Many powerful list operators.
- Arrays can also act as **stacks**, **queues** and **tuples**.
- Perl also provides **hashes**, ie. arrays where the index is an arbitrary string - i.e. a collection of (key,value) pairs indexed by key. Called Dictionaries in Python and Java.
- To do multi-dimensional arrays, arrays of hashes, pointers to functions etc, Perl provides **references** - an ability for one variable to refer to another variable; like C pointers.

- Ok, let's try some programs. Our first Perl program must be the classic "hello world" program. We can write this as a **Perl one-liner** (-e meaning "program follows as next argument"):

```
perl -e 'print "hello world\n"'
```

- Optionally, Perl's -l flag automatically adds newlines:

```
perl -le 'print "hello world"'
```

- However, if we want to create this as a proper Perl script, use an editor (vi, pico, nedit, emacs) to create the file **eg1** containing the following lines:

```
#
# eg1: a first Perl program
#
print "hello world\n";
```

- Then, we syntax check the program and run it:

```
perl -cw eg1                    (syntax check)
perl eg1                        (run it)
```


- Suppose we now create eg2 containing the following:

```
print "Please enter your name: ";
my $name = <STDIN>;
print "\nhello $name!\n";
```

- What's going on here?
  - `my $name` declares a *scalar variable*. It can hold a number (integer or real), or an arbitrary length string.
  - `<STDIN>` means *read one line from stdin*.
  - The line read is then assigned to `$name`.
  - The second print statement prints the result of the string `"\nhello $name!\n"` after *variable interpolation*: the current value of `$name` is interpolated into the string in place of the character sequence `$name`.
  - For instance, if `$name = "duncan"` then the string would be `"\nhello duncan!\n"`.
- Once again, we syntax check eg2 and then run it.
- Was there anything that surprised you when the program ran?

Why was the ! on a separate line?

- Because the <> operator reads a line and returns it *including the newline at the end* (for a good reason, explained later).
- To fix this, add:

```
chomp $name;
```

  immediately after reading `$name`. This deletes a trailing newline.
- Rerun and check that the ! is now on the same line as the name.
- Suppose we now wish to lowercase the whole name, and then capitalise the first letter:
- We simply add (after the `chomp`):

```
$name = ucfirst(lc($name));
```

- Recheck it and rerun it now.
- How long would that have taken to write in C *and know it's bug-free*?

- Suppose we now want a special-case - if the name is your first name, print out a special message:
- Embed the final print inside the else part of the following new if statement:

```
if( $name eq "Duncan" )
{
      print "\nwotcha Dunky babe!\n";
} else
{
      print "\nhello $name!\n";
}
```

- Syntax check, run it again a few times. Check it works.
- You may wish to try this with your own name (and stupid greeting) instead..

- Let's just refresh our memories - the complete program is now:

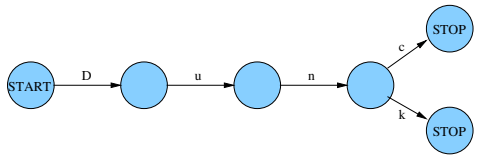
```
#
# eg4: special case greeting
#
print "Please enter your name: ";
my $name = <STDIN>;
chomp $name;
$name = ucfirst(lc($name));
if( $name eq "Duncan" )
{
        print "\nwotcha Dunky babe!\n";
} else
{
        print "\nhello $name!\n";
}
```


- So far, the name entered (after capitalization changes) must be your first name *exactly* for the special case to apply.
- For example, `duncan`, `DUNCAN`, `Duncan`, `DunCAn` are accepted, but `Dunc`, `Dunk` and the regrettable `Dunky Babe` are not.
- Replace the **if** line with:

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

What on earth does this mean?

- This is an example of matching a string against a *regular expression* - known as a *regex* - as found in the Unix filters **sed, grep** and **awk**.
- **regexes** are explained in more detail later, so for now let's just say that it succeeds if `$name` starts with the string `"Dun"`, immediately followed by *either* `"c"` or `"k"`. Graphically:



- Now, suppose that we require a secret word from anyone other than `"Dunc"`. Right at the top, below the comment lines, add:

```
my $secretword = "Davros";
```

- In the **else** part, add the following:

```
print "What is the secret word: ";
while(1)
{
      my $guess = <STDIN>;
      chomp $guess;
last if $guess eq $secretword;
      print "Wrong - guess again: ";
}
```

- This is an infinite **while** loop. Inside, we obtain a line of input, store it in `$guess` and `chomp` it as usual.
- We break out of the loop (**last**) if the guess is exactly the same as the secret word. If the guess is wrong, we continue round the loop. Notice the cute **last if condition** syntax.

- Suppose we now want several secret words. We want a **list**:

```
my @secretword = ( "Davros", "Zygon", "Cyberman" );
```

- Replace the **last if** line with:

```
my $correct = 0;
foreach my $i (@secretword)
{
      $correct = 1 if $i eq $guess;
}
last if $correct;
```

- Now any of the secret words will be accepted.
- But there's something slightly strange about what we just did: We stored the words in an ordered list and sequenced through it.
- But we didn't want to sequence through a list: We wanted a *set of secret words* and to test *set membership*.

- A Perl *hash* stores an arbitrary number of **(key, value)** pairs, indexing the keys.
- We can use a **hash** (where all the values are 1) as a **set**.
- Replace the `@secretword` initialization with:

```
my %issecretword = ();
$issecretword{"Davros"}   = 1;
$issecretword{"Zygon"}    = 1;
$issecretword{"Cyberman"} = 1;
```

- Now replace the entire `$correct` loop we just added with:

```
last if $issecretword{$guess};
```

- Hashes take quite a bit of getting used to, but are very powerful! Without them (eg in C), you'd probably have used an array and linear search. Ruby has them, Java and Python have them but call them dictionaries.
- All data structures you would build using pointers in C can be built with some combination of lists and hashes.

- Stupid to store the secret words in plain view inside the Perl script. Let's store them in plain view in a text file instead:-)
- Replace the initialisation of the `%issecretword` elements with:

```
open( my $in, '<', "secretwords" ) || die;
while( my $line = <$in> )
{
      chomp $line;
      $issecretword{$line} = 1;
}
close( $in );
```

- This shows us Perl's *foreach line in a file* idiom:
  - Open a text file called `secretwords`, exiting if the open fails.
  - While the file contains another line of text, read it.
  - Then process the line - in this case, add the line to the sethash.
  - When the last line has been dealt with, quit the **while** loop and close the file.

- There is an even better way of storing the secret words on disk and retrieving them: Unix provides a highly efficient storage system called DBM that stores arbitrary **(key,value)** string pairs.
- The concepts of DBM map perfectly onto a Perl hash.
- Need two programs: one to initialise the DBM file, and our existing program (modified) to read the DBM file:
- First, the creation program **mksecret** is as follows:

```
#
#       mksecret: create the secret words DBM file
#
dbmopen( my %secret, "secretwords", 0666 ) || die;
$secret{"Davros"}   = 1;
$secret{"Zygon"}    = 1;
$secret{"Cyberman"} = 1;
dbmclose  %secret ;
```

- Back in the main program - now called **eg10** - replace all the file reading code with:

```
dbmopen(my %issecretword, "secretwords", 0666) || die;
```

- At the end, add:

```
dbmclose( %issecretword );
```

- Now, the I/O is done entirely automatically for you.
- More efficiently too: the DBM file is not a plain text file. In a large DBM file containing millions of **(key, value)** pairs, retrieving the value corresponding to a specific key is usually done using **only two disk accesses**!
- By using this highly efficient system, we get *persistent storage* for Perl programs, for free!
- This is an example of what I meant by leverage.

- Let's close by showing the final version **eg10**:

```
#
# eg10: secret words from a dbm file
#
dbmopen( my %issecretword, "secretwords", 0666 ) || die;

print "Please enter your name: ";
my $name = <STDIN>;
chomp $name;
$name = ucfirst(lc($name));
if( $name =~ /^Dun[ck]/ )
{
        print "\nwotcha Dunky babe!\n";
} else
{
        print "\nhello $name!\n";
        print "Please enter one of the secret words: ";
        while(1)
        {
                my $guess = <STDIN>;
                chomp $guess;
        last if $issecretword{$guess};
                print "Wrong - guess again: ";
        }
}

dbmclose( %issecretword );
```