Perl Short Course: Introduction

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

- This short course gives a practical introduction to Larry Wall's well established and immensely flexible **Perl** programming language.

- It consists of a series of 6 one-hour lectures (usually arranged as 3 two-hour sessions), with slides and practical examples. There are also a number of longer practical examples for you to try.

- The course materials (notes and examples) for the course can be found at:

  `http://www.doc.ic.ac.uk/~dcw/perl2010/`

  - This first lecture will give a general introduction to Perl, a fast and shallow scan across most of the important features.

  - The second, third and fourth lectures will go over Perl in more detail, introducing most of the interesting features of the language.

  - The fifth and sixth lectures will describe the Perl module archive (CPAN) and some of it's modules, and tell you how to construct your own modules and classes.

- 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.**

Why Perl?

- We should start by asking: *Do we need yet another computer language?* There are hundreds of computer languages around already – do we really need yet another?

- The next obvious question is: *Should I bother learning it?* I hope to convince you the answer to both questions is Yes - that there are very good reasons for learning one more language.

- We shall see that 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.

- Perl gives us great **leverage** in writing programs when compared to languages like C, C++ (good as they are!) or Java. Instead of starting from the architectural level (highly machine-efficient but not programmer-efficient) Perl lets you program at a level much higher up, with more powerful tools.

- Perl is an immensely pragmatic language, borrowing the best features from many other languages - forming a coherent whole, more powerful than the sum of its parts.

- But first, let's look at what came before Perl.

Shell Programming

Perl grew out of Unix shell programming, so we'd better look briefly at that:

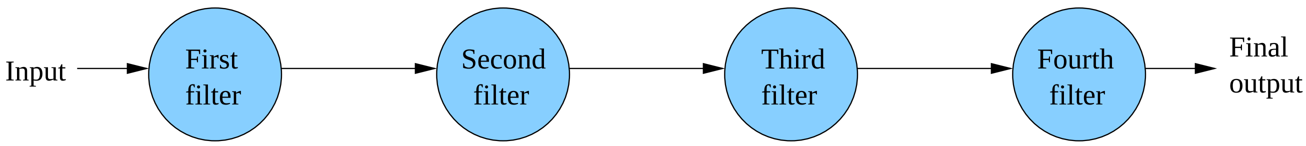
- The term *Shell Programming* describes the way in which a series of Unix commands, including simple loops and conditionals, may be placed into a file and made executable, forming a *Shell Script*.

- Much of the power of Unix comes from the toolkit of small but powerful programs, called *filters*, that can be used in shell scripts.

- Each filter reads input from the standard input *(stdin)* and/or named files, performs some simple sorting, searching or summarising operation on the data, and then writes the resulting output to the standard output *(stdout)*.

- Filters can be linked together into a *pipeline*, in which the *output* from one filter can be provided as the *input* to another filter.

- Thus, input flows into the pipeline, is processed by each stage of the pipeline, and the final output emerges from the end of the pipeline.



Some of the most important filters are:

- **wc** - read each file of input, counting words, characters and lines.

- **tr** - translate every occurrence of any character in one set of characters into the corresponding character in another set. Given only one set, deletes all characters in the first set.

- **sort** - sort the entire input, line by line, in a variety of sorting orders (ascending, descending, numeric, alphabetic etc).

- **grep** - extract all the lines from the input that match a particular pattern called a **regular expression**.

- **sed** - a stream editor that lets you perform a variety of structural changes to the input stream. You can delete particular lines (either by range of line numbers, or all those lines matching a regular expression), search and replace lines, add new lines, extract particular lines etc.

- **awk** - an interpreted pattern-matching language which treats each line as a record, allows you to select a set of records, and then select and summarise some fields.

  For example, one could select a group of records, select two columns from those records, and total up the columns.

Disadvantages of Shell Programming

For all the advantages that shell programming has, there are some disadvantages:

- Every time a script wants to do some arithmetic, even to increment a variable, it has to invoke an external program (called **expr**) to do it.
- In general, a script may invoke hundreds of external commands. Each one takes lots of CPU time, memory and disk accesses to load and start running.
- Therefore, a shell script can be much slower to run than an equivalent C program. Of course, it is often faster to write, because it is constructed out of prewritten filters!
- Quoting rules in the shell are rather tricky, especially when you embed complex **awk** or **sed** programs (with their own quoting rules) into shell scripts. On occasion, working out how to escape a quoted string inside a filter can defeat even the experts.
- Finally, even though the basic Bourne shell is portable among hundreds of Unixes, many of the detailed filter commands vary slightly from Unix to Unix - and thus writing a totally portable Unix shell script is very tricky (but not impossible).

One step beyond the Shell

So; how about something a bit more like a full-blown programming language than the shell is, but still interpreted, and merging in all the best bits?

Such a language should have:

- Built-in arithmetic and more powerful data types (arrays, lists and associative arrays) than the shell.

- The capabilities of filters like **grep, sed, awk** and **sort** built-in to the language, so that they may be used without needing to call external programs. **grep, sed** and **awk** all had slightly different basic metaphors - these should be unified.

- Full *regular expressions* built in and available for use in a variety of novel and powerful ways, as well as the familiar **grep, sed** and **awk** idioms.

- The ability to read or write data from/to external command pipelines as easily as reading or writing data from files.

- The ability to manipulate files, processes and command line arguments simply and efficiently, in order to be able to build our own filters.

- Plus: threads, portable graphics, OOP, distributed programming and more modules than you can count.

- *Well, this is Perl!*

Perl Components

Perl borrows the best from several different languages, and binds them together into a seamless whole:

- Fundamental control structures are taken from C and the shell.

- Expression syntax comes from C, with several operators brought in from the shell – such as a set of file test operators.

- Regular expressions come from **sed** and **grep**, and Perl extends them in a gradually increasing number of ways to make them even more powerful.

  In fact, the **Perl Compatible Regular Expressions (PCRE)** library is rapidly becoming the new standard regular expression library..

- The ability to run external Unix commands or pipelines, supply them with programmatically generated input and retrieve their output, comes direct from the shell.

- Easy file handling comes from C, **awk** and the shell.

- Most crucially, Perl does all *storage management* for us - just like the shell, **awk,** and **Java**, but unlike **C** and many other C-family languages.

Perl does Storage Management

- There is growing evidence that we humans just shouldn't do storage management, because we always get it wrong! Some experts have been quoted as suggesting that 99% of serious programming errors are caused by bad storage management.

- Overflowing arrays, retaining pointers to deallocated memory, dereferencing NULL pointers, forgetting to deallocate memory in a loop - all these are fundamental storage management problems.

- The most obvious example is **strings**: in C these are simply character arrays - the most efficient and low-level representation available. So whenever we declare a string variable, we have to decide the maximum length of string it can hold. But if we're reading data from a file, we simply don't know how long it might be!

- Of course, most people just "over-engineer" their string variables - declare them as ludicrously large. A few people then write overflow prevention code. A tiny number of people write **correct** overflow prevention code.

- Does all this matter? Yes! There are numerous examples where storage management bugs hit real systems and products - hundreds of recent worms and viruses depend critically on buffer overflows, from **Nimda** and **Sasser** right back to the original **Internet Worm**.

The main ways Perl does storage management are as follows:

- Strings are not treated as arrays of characters as they are in most C-family languages (ok, not Java!) - rather as a basic type. 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. All elements that have not had a value assigned to them have the undefined value, effectively zero.

- Perl provides arbitrary unlimited length Prolog-style lists, which are actually arrays. Perl provides many powerful list/array operators.

- These lists can also act as Prolog-style **tuples** - an unnamed collection of data like a record.

- Perl also provides associative arrays, ie. arrays where the index can be an arbitrary string - think of this as a set of (key,value) pairs. These are called **hashes** in Perl speak.

- To do anything more complex, eg. multi-dimensional arrays, Perl provides **references** - an ability for one variable to refer to another variable. These are rather like pointers, but are more structured.

Your First Perl Program (eg1)

- Ok, let's jump in and get started. Following a long tradition, our first Perl program will be the classic "hello world" program.

- Using any editor (vi, pico, nedit, emacs), we create a file called **eg1**, containing the following lines:

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

- Then, we syntax check the program:

  ```
  perl -cw eg1
  ```

- Finally, we run the program by typing:

  ```
  perl eg1
  ```

- What can we see immediately about Perl from this example?
  - Lines beginning with `#` are comments, ignored by Perl.
  - Statement are terminated with semi-colons.
  - A string is placed in double quotes, and can contain C-style special characters such as `\n`.

Personalising the Message (eg2)

- Suppose we now create `eg2` containing the following:

  ```
  #
  # eg2: greetings!
  #
  print "Please enter your name: ";
  $name = <STDIN>;
  print "\nhello $name!\n";
  ```

- What's going on here?
  - `$name` is a *scalar variable*. It can hold a number (integer or real) or an arbitrary length string. The `$` is always present on scalar variables.
  - `<STDIN>` means *read one line from stdin*.
  - The second print statement prints the result of the string `"\nhello $name!\n"` after *variable interpolation*. This means that the current value of `$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*.

- To get round this common problem, add:

  ```
  chomp $name;
  ```

  immediately after reading `$name`. This deletes the newline at the end.

- Rerun and check that the `!` is now on the same line as the name.

Let's capitalise the first letter of the name (eg3)

- 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*?

Adding Special Cases (eg4)

- Suppose we now want a special-case - if the name is your first name, print out a special message, otherwise print the generic message:

- Replace the final print with the following code:

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

- Let's just refresh our memories - the complete program is now **eg4**:

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

Adding some Fuzziness (eg5)

- 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` etc are all recognized, but `Dunc`, `Dunk` and the regrettable `Dunky Babe` are not.

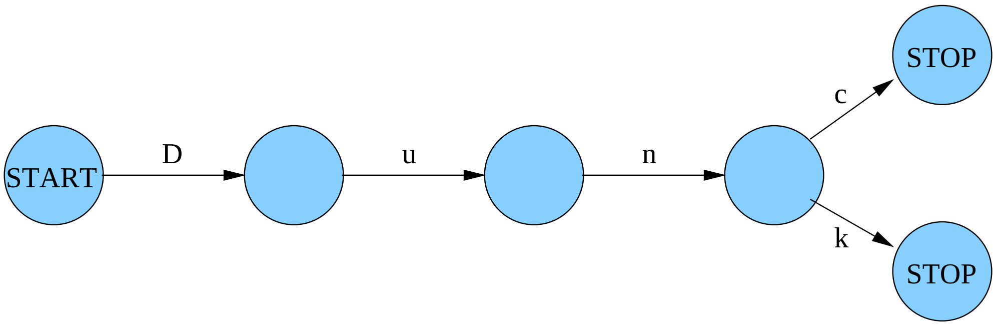
- Replace the **if** condition with:

  ```
  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 *regexp* - as found in the Unix utilities **sed, grep** and **awk**.

- **regexps** 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:



- You may like to make a similar change involving misspellings of your first name and try it out.

Adding a Secret Word (eg6)

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

  ```
  $secretword = "Klingon";
  ```

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

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

- The basic structure is simply an infinite **while** loop, meaning "repeat forever". Inside the loop, we obtain a line of input, store it in `$guess` and then `chomp` it as usual.

- Then, we break out of the loop if the guess is exactly the same as the secret word. If the guess is wrong, we print the `"Wrong.."` message and continue round the loop. (Notice how **if** is placed *after* the command **last**, acting as a modifier.)

- You should recognise this as the classic *N-and-a-half loop*. Perl has (probably) the cleanest implementation of the N-and-a-half loop that I've ever seen!

More than One Secret Word (eg7)

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

- Replace the assignment to `$secretword` at the top with the following:

```
@secretword = ( "Klingon", "Romulan", "Vulcan" );
```

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

```
$correct = 0;
foreach $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 really want to sequence through a list: We just wanted to be able to directly look up the answer to the question *"Is this string a valid secret word?"*, by associating a true value with each secret word and false with every other word.

Using a Hash (eg8)

- Perl supports *hashes* - or *associative arrays* - that can store an arbitrary number of **(key, value)** pairs, i.e. associating an arbitrary value with a key.

- Replace the `@secretword` initialization with:

```
%issecretword = ();
$issecretword{"Klingon"} = 1;
$issecretword{"Romulan"} = 1;
$issecretword{"Vulcan"} = 1;
```

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

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

- Hashes take quite a bit of getting used to! Not many languages support them - in C, you'd probably have implemented the ordered list as an array and searched through one by one.

- Once you've got used to hashes, you never want to be without them! Amazingly, you find that a lot of the data structures you ever need to build – using lots of pointers etc in C – can all be done with the combination of lists and hashes.

Permanent Storage (eg9)

- It's a bit 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( IN, "secretwords" ) || die;
while( $line = <IN> )
{
        chomp $line;
        $issecretword{$line} = 1;
}
close( IN );
```

- This shows us Perl's classic idiom *read and process every line from a file*:
  - We open a text file called `secretwords`, exiting if the open fails.
  - While the file contains another line of text, we read it and chomp it.
  - Then we process the line - in this case, we use the line as the key into an hash, setting that element to 1.
  - When we have read the last line from the file, we quit the **while** loop and close the file.

Better Storage (eg10)

- 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 associates an arbitrary string with an arbitrary key.

- Perl provides an almost trivial interface to DBM files, because the concepts behind DBM map perfectly onto a Perl hash.

- Now, for the first time, we need two programs: one to initialise the DBM file, and our existing program (modified a bit) to read the DBM file:

- First, the creation program **mksecret** is as follows:

```
#
#       mksecret: create the secret words DBM file
#
dbmopen( %secret, "secretwords", 0666 ) || die;

$secret{"Romulan"} = 1;
$secret{"Klingon"} = 1;
$secret{"Vulcan"} = 1;

dbmclose( %secret );
```

- Back in the main program - now called **eg10** - remove all the file reading code (up to `CLOSE(IN)`) and replace it with:

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

- Right at the end of the program, add:

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

- Now, the file reading is done entirely automatically for you.

- What's best of all is that the DBM file is not just a plain text file that has to be read in its entirety. Retrieving a single data value given the key value from a very large DBM file containing millions of (key, value) pairs is usually done using only two disk accesses.

- And we can simply go ahead and use this highly efficient system - which gives us *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( %issecretword, "secretwords", 0666 ) || die;

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

dbmclose( %issecretword );
```