Perl Short Course

Second Session

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

- In this second session, we'll go over Perl in more detail, going through most important parts of the language, with lots of practical examples for you to try later.

- We'll broadly follow the order of material presented in Randal Schwartz's excellent introduction **Learning Perl** (the Llama book).

- We'll look at **scalar data** (numbers, strings, variables) and **expressions**, **control structures** and **input/output**.

- Next session, we'll look at more complex bits of the language: **arrays and lists** and **hashes**, **special variables** and **regular expressions**.

- In the remaining sessions, we'll look at **functions**, **references**, **modules**, **objects and classes**, take a quick tour of some of Perl's standard library and find out how to write modules.

A Note on Perl History

- **Perl 1 - Perl 3**, 1987-89: Larry Wall designed the **Practical Extraction and Report Language** for his own amusement, and released one version of Perl per year. Note: Perl was 20 years old on Dec 18th 2007!

- **Perl 4**, 1991-1996: the first widely used version of Perl. The final release was **Perl 4.036**, development then stopped.

- **Perl 5**, 1994-now: Larry Wall completely rewrote the perl interpreter, and added significantly to the Perl language. **Perl 5** has been maintained, patched and extended for 15 years, with new features being added every year or two.

- **Perl 5.10**, 2003-now: The Perl 5 developers worked on a new major version of Perl 5 for several years (in the experimental **Perl 5.9** series) and released **Perl 5.10.0** in Dec 2007. Perl 5.10 is now installed on DoC Ubuntu Jaunty and Windows Vista PCs. To find out the version installed, do **perl -v**.

- **Perl 6**, 2001-????: The Perl developers are also working on a fundamental redesign of Perl - **Perl 6** - very different internally from Perl 5. However, it never seems to get finished! Will it ever?

Scalar Variables and Values

- A scalar variable is written as `$variablename` - the `$` is **always** present. A variable name starts with a letter, and then may continue with any number of letters, digits or underscores! Every character in the name is significant, as is case.

- You can also write a scalar variable as `${variable}`, just like in the Unix shell.

- A scalar variable can store any single scalar value:
  - An integer.
  - A real number (floating point).
  - An arbitrarily long string.
  - A reference to another variable.

- Integer literals may be written in decimal, octal or hexadecimal:

  eg. `-129, 0377 and 0xffe0.`

- Real literals may be written in *exponential notation*, with or without a *decimal point*:

  eg. `6.27, -2.4e30, 7e-5.`

- A string literal can be either delimited with single-quotes or double-quotes. Entirely different rules apply inside the two kinds of string.

Single-Quoted Strings

- A single-quoted string literal, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.

- The shortest possible single-quoted string is just `''`.

- The quotes themselves are not actually part of the string - rather, they *delimit* both ends of the string.

- Each character inside the quotes is placed into the string exactly as-is - even a newline!

- There are only two exceptions to this:
  - A single-quote on it's own would terminate the current string - so to embed a single quote inside a single-quoted string, write `\'`.
  - A backslash itself may be embedded inside a single-quoted string by writing `\\`.

- Apart from backslashed single quotes and backslashed backslashes, nothing is modified inside a single quoted string.

- In particular, `$` symbols are embedded as-is, and even the C-style backslash escapes such as `\n` do not function inside a single-quoted string.

Double-Quoted Strings

- A double-quoted string literal, like `"hello $name\n"`, is much more powerful. Again, it is an arbitrary sequence of characters, but this time inside double quotes. As before, it can be split across lines (ie. contain internal literal newlines!).

- However, the backslash can now specify several important control or *escape* operations as follows:

`\n`	Newline
`\t`	Tab
`\r`	Carriage Return
`\a`	Ring bell
`\075`	Any octal ASCII value (in the above example, 7*8+5 = 61)
`\x6d`	Any hexadecimal ASCII value (in the above example, 6*16+13 = 109)
`\\`	Backslash
`\$`	Dollar
`\"`	A double-quote
`\l`	Lower-case the next letter
`\u`	Upper-case the next letter
`\L`	Start lower-casing the rest of the string
`\U`	Start upper-casing the rest of the string
`\E`	Stop `\L` or `\U`

- In addition, `$` has a special meaning when followed by a sequence of characters looking like a variable name.

- Perl gathers the longest possible sequence of variable name characters up, and replaces the entire sequence with the value of that variable.

- This process is called **variable interpolation**.

- So, for instance, `"hello $name\n"` is a string literal, comprising:
  - the character sequence `'h','e','l','l','o',' '`
  - the current value of the variable `$name` (if `$name` is not in use, then the value is the empty string),
  - the ASCII newline character (symbolically written as `\n`).

- Suppose we wish to print the value of a variable (eg. `$n`), immediately followed by a sequence of characters (eg. `"th"`) that could also form part of the variable name. If we just write:

```
"you're the $nth person today..\n"
```

  Perl will look for a variable called `nth` instead, and end up interpolating the empty string!

- To prevent this, use `{}` around the variable name:

```
"you're the ${n}th person today..\n"
```


Arbitrarily Delimited Strings

- In double-quoted strings, you still need to backslash double quotes to embed them into a string. You can also write a string obeying all the double-quoted rules, but choose your own quote character (delimiter):

```
$contentslink = qq!
<a href="$v.html">
   <img src="$v.png" alt="Back to \u$v">
</a>
!;
```

- After the `qq`, the next character (the exclamation mark) is taken as the quote character. All characters after this are placed into the string (with interpolations and backslashes working exactly as in a double-quoted string) until the next exclamation mark is encountered.

- If the opening quote is an open bracket of some kind (round, curly or square), Perl uses the appropriate closing bracket as the closing quote - We could write the above as:

```
$contentslink = qq(
<a href="$v.html">
   <img src="$v.png" alt="Back to \u$v">
</a>
);
```


Expressions

- Expressions in Perl are very like those of C, but without the pointer operators.

- Some operators expect numbers, others expect strings. Perl is sufficiently liberal to convert one to the other when needed. Thus a string such as `"17.3e7"` will convert to the number `17.3e7` when needed, and a number such as `31.53` will convert itself to the string `"31.53"` when needed.

Numeric Operators

- Perl provides all the obvious +, -, *, / arithmetic operators just as you'd expect. For instance:

```
25 + 7              3 * 12.1
4.7 - 3.1           14 / 2.1
```

- Unlike C, *all operators operate in floating point*.

- Unlike both C and Pascal, Perl provides the *exponentation operator* (written **). For example: `3.25 ** 4` (ie. 3.25^4).

- Like C, Perl provides a *modulus* operator: `10%3` gives the remainder when 10 is divided by 3. Both values are truncated to integers before this operator is applied.

- Perl provides a set of numeric comparison operators just like C: `<`, `<=`, `==`, `>=`, `>` and `!=` (not equals).

String Operators

- Strings can be *concatenated* with the fullstop operator (**.**) - for example:

```
'hello ' . $name . "\n"
```

(Warning: No space is added automatically between the strings!)

- Concatenation is not used as much as you would expect - interpolation is often used instead. The above example would normally just be written as:

```
"hello $name\n"
```

- Strings may be compared (alphabetically) via a separate set of comparison operators: **lt, le, eq, ge, gt** and **ne**.

  One of the commonest mistakes is to compare numbers with **eq** or strings with **==**. Strings and numbers are converted back and forth very easily.

  Consider `"0" == "000"` (both are converted to numbers and compared numerically, hence the comparison succeeds) and `0 eq "000"` (0 is converted to the string `"0"` which is not the same string as `"000"`!)

- Another useful string operator is the *repetition* operator (written **x**):

  `"fred" x 3` gives the result `"fredfredfred"`

  The right-hand argument is truncated to an integer before the replication occurs.

Operator Precedence and Associativity

- Each operator (in any language) has two properties: *precedence* and *associativity*.

- *Precedence* governs what happens when several operators compete for operands:
  - Consider $3 + 4 * 5$. Is that $(3 + 4) * 5 = 35$, or is it $3 + (4 * 5) = 23$ as our maths intuition suggests?
  - In fact, the latter is the case (your intuition is almost always respected!). This means that ***** has a *higher precedence* than **+**.

- *Associativity* governs what happens when operators of the same precedence compete for operands:
  - Consider $2 * *3 * *2$. Is that $(2^3)^2$ or is it 2^{3^2}? They have considerably different answers: the former gives $8^2 = 64$ whereas the latter gives $2^9 = 512$.
  - So does ****** associate to the *left* or to the *right*? (the answer is *right*).

- The full precedence and associativity tables may be found in either the Camel or Llama Books.

- If your intuition fails you, *just put things in* (). This causes the innermost expression to be evaluated first.

Assignment

- Perhaps the most vital operator of all is assignment (=). For example:

```
$x = 37;
$y = $x * 7 + 5;
$z = $z * 3;
```

  - The first simply sets `$x` to 37.
  - The second calculates `$x * 7 + 5` (using the current value of `$x`) and then stores the result in `$y`.
  - The last example takes the current value of `$z`, multiplies it by three, and stores the result back in `$z`.

- An assignment also has a value, which means it can be used like:

```
$y = 5 * ($a = 7 + $x);
$x = $y = 17;
```

  - The first example means: evaluate `7+$x` and store the result in `$a`, then multiply `$a` by 5 and store the final result in `$y`.
  - The second example sets both `$x` and `$y` to 17.

Binary Assignment Operators

- Assignments like `$z = $z * 3` are so common that Perl (just like C) provides a set of assignment operators that modify a single variable.

- Nearly any binary operator that computes a value can have an = sign appended on the end. For instance:

```
$z *= 3;                    $sum += $element;
```

  These read very intuitively as *multiply* `$z` *by 3*, and *add* `$element` *to* `$sum`.

- The same works for strings: `$z .= $a;` appends the contents of `$a` onto the end of `$z`.

- Just like basic assignment, all the binary assignment operators return a value too. So the following is valid:

```
$a = 3;
$b = ($a += 4) * 7;
```

- **However**, never change the same variable inside two parallel branches of the same expression. The following is extremely unclear and should be avoided:

```
$a = 3;
$b = ($a += 4) * ($a -= 2);
```

Autoincrement and Autodecrement

- We have already seen that the common operations `$a = $a + 1` and `$a = $a - 1` can be written as `$a += 1` and `$a -= 1` In fact, there's an even easier way of doing it (taken straight from C):

```
$a++;                    $a--;
```

  These mean simply *increment* and *decrement* `$a`.

- Perl has a prefix and postfix form of these operators: `++$a`, `$a++`, `--$a` and `$a--`. Both forms increment (or decrement) the variable in the same way.

- The difference lies in what value is returned as the expression result. Consider the following two examples, first:

Perl	Simplified	Effect
`$a = 7;`	`a=7`	a:7
`$b = ++$a;`	`a++; b=a`	a:8, b=8
`$c = $b--;`	`c=b; b--`	c=8, b=7

- Second:

Perl	Simplified	Effect
`$a = 7;`	`a=7`	a:7
`$b = $a++;`	`b=a; a++`	b:7, a:8
`$c = --$b;`	`b--; c=b`	b:6, c:6

  In both cases, `$a` ends up as 8.

Boolean Expressions

- Expressions used in a boolean context (which we'll see in a moment) are evaluated in the usual way as numbers or strings.

- Then, when the single result value is produced (a number or a string), a *zero value or empty string* is interpreted as *false*, and *anything else is true!*

  There are a few boolean operators that we haven't mentioned so far:

- `!` is a unary operator meaning *not*.

- `||` and `&&` come straight from C – `||` is pronounced *or* and `&&` is pronounced *and*.

- Just like C and Java (but unlike Pascal), these operators are *short-circuit*:
  - Consider:

```
$a < 7 || $b > 6
```

    If `$a` is less than 7, then the whole expression is bound to be true. There is no point in evaluating the rest of the expression - so Perl doesn't!
  - Similarly, consider:

```
$s eq "hello" && $t ne "bonjour"
```

    If `$s` is not `"hello"` then the whole expression must be false. Again, there's absolutely no point in evaluating the second half.

Some Simple I/O

Computation is useless without being able to input parameters and output results. For now, lets consider getting input from the keyboard, and reporting results to the screen.

Reading a line of input:

- The <> operator (pronounced *diamond* or *getline*) fetches a line of input from a file. `STDIN` is just such a file, and represents the keyboard. So:

```
$name = <STDIN>;
```

  reads an entire line of input, up to and including a newline, and then stores the input in the variable (including the newline).

Chop and Chomp

- After reading a line, you usually want to get rid of the trailing newline. Two operators:

- Perl 4 provided `chop($name)` which simply removes the last character from the named variable and returns the character removed.

- Perl 5 added `chomp($name)` which only deletes the last character if it's a newline!

- Use `chomp` everywhere; it's safer and more portable (it will remove the newline from the string even if the newline convention on the OS you're using is not a single character!).

Printing the results:

- There is a nice simple way of printing some results:

```
print expression;
```

- Combined with the variable interpolation we've seen in strings, this gives us all the power we need for now. This gives us the simplest debugging strategy in Perl, as in:

```
print "debug: x=$x, y=$y, z=$z\n";
```

- Here's an example (**eg1**), allowing you to investigate different expressions:

```
print "Please enter x: ";
$x = <STDIN>;
chomp $x;
print "Please enter y: ";
$y = <STDIN>;
chomp $y;
$z = $x + $y;
print "\n$x + $y = $z\n";
```

- You may like to spend some time trying some examples of expressions, starting with the above example and incorporating different, more complex, expressions.

- Try various += and ++ types of operators to get clear exactly what they do.

Statements and Control Structures

- Perl allows any expression to be made into a statement simply by appending a ;. That is, 3; is a valid statement, albeit a rather silly one.

  It means *evaluate the expression* (giving 3 as the result) and then *discard the final result*.

- Usually, the expressions that we tend to make into statements are ones that have side-effects:

  – Assignments (the basic =, all the binary assignment operators like *=, and the ++ and -- operators.

  – Procedure/subroutine calls. We'll see these in the fourth session.

- Perl also provides *modifiers* for single statements:

```
<statement> if <expr> ;
<statement> unless <expr> ;
<statement> while <expr> ;
<statement> until <expr> ;
```

  (Perl allows unless to be used as a synonym for if !, ie. *if not true*, and until as a synonym for while !).

- A sequence of statements may be enclosed inside {} braces forming a group called a *block*. There is no ; after the } of a block.

- Perl provides a conventional **if.. elsif.. elsif.... else** statement, to choose between two or more alternatives:

```
if( $i < 20 || $j > 7.4 )
{
        print "case one\n";
} elsif( $i > 40 && $j > 0 )
{
        print "case two\n";
} else
{
        print "case three\n";
}
```

  Note that the brackets – both round and curly – are compulsory on an **if** - and the loop below.

- In a rare omission, Perl did **NOT** provide a special case/switch statement for multi-way comparisons, at least until Perl 5.10. However, an add-on module exists that adds this facility.

- Perl also provides a conventional *test at the top* while loop:

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


- In fact, the above algorithm happens to find the square root of a number!

- Perl provides a *test at the bottom* loop, two forms:

```
do {                          do {
        $y *= $x;                     $y *= $x;
        $x += 2;                      $x += 2;
} while( $x < 15 );           } until( $x >= 15 );
```

- A C-style *counting loop*:

```
for( $sum = 0, $i = 1; $i <= 10; $i++ )
{
        $sum += $i;
}
```

- A very useful *for each element in a list/array* loop:

```
foreach $x ( 1, 3, 7, 5, 4, 54 )
{
        $total += $x;
}
```

  (We'll see more about arrays next session).

- Exercise: Try writing a little Perl program that reads in a number and computes the square root using the while loop on the previous page, printing out a message like "The square root of $x is $w" at the end.

More I/O - End of Input

- We've seen that $line = <STDIN> reads a whole line of input from stdin (normally keyboard).

- But what if there's no more input? Files have a concept of *end of file (eof)* - after you have read the last line of data!

- So does the keyboard (slightly less obviously). On Unix systems, keyboard end-of-input is signalled by the user typing CTRL-D on a line on its own. ie. one line ends with newline as normal and the very next character encountered is a CTRL-D.

- No CTRL-D character (ASCII code 4) is delivered; CTRL-D is interpreted by the terminal driver as a signal to end the keyboard input for that process.

- When eof occurs, <> returns the *empty string*. This never otherwise occurs - because of the trailing newline that <> leaves in any non-eof line read.

- So comparing the result of <> against the empty string will reliably determine eof:

```
if( $name eq "" )...
```

- More typically, you test for *not eof*, which is written if($name ne "") or if($name).

For Each Line Idiom

- The fact that an assignment returns the assigned value, and that any non-empty string is interpreted as true, means that we can even combine the line reading, assignment, and test for not eof into:

```
if( $line = <STDIN> )   # if we've read a line..
```

- Using a `while` not an `if` leads us directly to a very common Perl *idiom*: how to process every line of stdin:

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

- Let's see an simple example of that (**eg2**):

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

- We fill the line processing slot with `$sum += $line`, and add `print "total: $sum\n"` at the end.

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 (overwriting the previous contents):

```
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 **eg2**, 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`.