

Introduction to Perl: Second Lecture

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

Dept of Computing,
Imperial College London

January 2015

- A scalar variable is declared via `my $variablename`, the `$` 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.
- Can write as `${variablename}`, just like the Unix shell.
- Stores any single scalar value:
 - An integer.
 - A real number (floating point).
 - An arbitrarily long string.
 - A reference to some other piece of data.
- 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. Different rules apply inside the two kinds of string:

- In this second session, we'll start going over Perl in more detail, following order of material from **Learning Perl**.
- We'll look at **scalar data** (numbers, strings, variables), **expressions**, **control structures** and **input/output**.

A Note on Perl History

- **Perl 4**, 1991-1996: first widely used version of Perl.
- **Perl 5**, 1994-now: complete interpreter rewrite, significant language extensions. Maintained, patched and extended for over 20 years.
- Since 2010: developers do annual releases of Perl 5, eg **Perl 5.16 in Spring 2012**, **Perl 5.18 in Spring 2013**. **Perl 5.20 in Spring 2014**. (DoC Linux machines: 5.18; DoC Windows: 5.16.3).
- **Perl 6**, 2001-????: fundamental redesign of Perl, a new programming language! BUT: never finished! Will it ever be?

- A single-quoted string, eg `'hello there'`, is an arbitrarily-long sequence of characters placed inside single quote marks.
- The quotes are not part of the string - rather, they *delimit* both ends of the string.
- The shortest (empty) single-quoted string is `''`.
- Each character inside the quotes is placed into the string exactly as-is - even a literal newline!
- There are only two exceptions to this:
 - A single-quote on it's own would terminate the current string - to embed a single quote in a single-quoted string, write `\'`.
 - A backslash may be embedded in a single-quoted string as `\\`.
- Apart from these exceptions, nothing is modified inside a single quoted string.
- In particular, `$` symbols are embedded as-is, and C-style escapes like `\n` do not function in a single-quoted string.

- A double-quoted string literal, like "hello \$name\n", is much more powerful. Again, it is an arbitrary sequence of characters which can be split across lines.
- However, the backslash can now specify several important control or *escape* operations as follows:

\n	Newline
\t	Tab
\r	Carriage Return
\a	Ring bell
\072	Any octal ASCII value (7*8+2 = 58 = ':')
\x6d	Any hexadecimal ASCII value (6*16+13 = 109 = 'm')
\\	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

- For example, "hello \Uthere\E how are you" generates "hello THERE how are you".
- Also in a double-quoted string...

- ...\$ has a special meaning when followed by a variable name: Perl gathers the **longest possible** sequence of variable name characters, and replaces \$varname with the value of that variable: *Variable Interpolation*.
- Example: "hello \$name\n" is a string, comprising:
 - the character sequence 'hello ',
 - the current value of the variable \$name (if \$name is not in use, then the value is the undefined value, treated as the empty string),
 - the ASCII newline character (symbolically written as \n).
- Suppose we wish to print the value of \$n immediately followed by a sequence of characters that could also form part of the variable name. If we write:


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

 Perl looks for a variable called \$nth, and replaces it with the empty string - swallowing th!
- Fix: use {} around the variable name:


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

- In double-quoted strings, backslash double quotes to embed them into a string. Avoid this: choose your own quote character:

```
my $contentslink = qq!  
<a href="$v.html">  
    
</a>  
!;
```

- After qq, the next character (!) 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 ! is encountered.
- But now - you don't need to backslash double-quotes.
- 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:

```
my $contentslink = qq(  
<a href="$v.html">  
    
</a>  
);
```

- Expressions in Perl are very like those of C, but without the pointer operators.
- Some operators expect numbers, others expect strings. Perl converts one to the other when needed. Thus the string "17.3e7" converts to the number 17.3e7 when needed, and the number 31.53 converts to the string "31.53" when needed.
- Perl provides all the obvious +, -, *, / arithmetic operators. Unlike C, *all operators operate in floating point*.
- Plus, the *exponentiation operator* (written **). For example: 3.25 ** 4 (ie. 3.25⁴).
- The *modulus operator*: 10%3 gives the remainder when 10 is divided by 3. Both values are truncated to integers before this operator is applied.
- A set of *numeric comparison operators* just like C: <, <=, ==, >=, > and != (not equals).

- Alphabetical *string comparison* via a separate set of comparison operators: **lt**, **le**, **eq**, **ge**, **gt** and **ne**.
- A common mistake is comparing numbers with **eq** or strings with **==**, allowed because strings and numbers are converted back and forth freely.
- For example, "0" == "000" is true, but 0 eq "000" is false. (The string "0" which is not the same as "000"!)
 - *String concatenation* done with the fullstop operator (**.**), eg:


```
"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. Write the above as:


```
"hello $name\n"
```
- Useful *string repetition* operator:


```
"fred" x 3
```

 gives "fredfredfred" (RHS truncated to an integer first).

- 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 you can **nest** or **chain** assignments:


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

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

- Any binary operator can have an **=** sign appended. For instance:

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

Read these as: *multiply z by 3*, *add element to sum* and *append \$a's value onto the end of \$z*.

- All binary assignment operators return a value too. So can do:

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

- Some avoid such nesting, viewing it as hard to understand. Personally, I'm fine with modest amounts of nesting: Your choice.

Autoincrement and Autodecrement

- To *increment* or *decrement* \$a, write:

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

- Prefix and postfix form of these operators: ++\$a, \$a++, --\$a and \$a--, work exactly like C and Java.
- Probably a bad idea to embed these in much larger expressions.

- Expressions used in a boolean context are **first** evaluated as numbers or strings.
- When the result has been calculated: *zero*, *the empty string* or *the undefined value* are *false*, *anything else is true!*

There are a few extra boolean operators:

- **!** is a unary operator meaning *not*.
- **||** and **&&** come straight from C – these can be written as 'or' and 'and' in modern Perl.
- Just like C and Java, they *short-circuit*:
 - Consider:


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

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


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

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

- Any expression can become a statement by appending ‘;’.
- So `3+4;` means *evaluate expression* (result: 7) and *discard the result*. A pointless statement!
- Usually, expression-statements have side-effects:
 - Assignments (the basic =, binary assignment operators like *=, and ++ and -- operators).
 - Function/procedure calls (eg. print - or your own functions).
- Perl provides *single statement modifiers*:

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

‘unless’ is a synonym for ‘if !’; ‘until’ a synonym for ‘while !’.

- A sequence of statements may be enclosed inside {} braces forming a *block*. NB: 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.

- Historically, Perl did **NOT** provide a special case/switch statement for multi-way comparisons. Perl 5.10 added ‘given/when’, but you have to ask for 5.10 features to be enabled:

```
use v5.10; # at the top of the program, enable 5.10 features
....
given( $x )
{
    when( "hello" ) { print "hello\n"; }
    when( /^Dun[ck]/ ) { print "dunc\n"; }
    default { print "unknown\n"; }
}
```

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

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

- The above algorithm happens to find the square root of \$x!
- Perl provides a *test at the bottom* loop, two forms:

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

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

- A C-style *counting loop*:

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

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

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

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

Lets consider getting input from the keyboard, and reporting results to the screen.

- The <> operator (pronounced *diamond* or *readline*) fetches a line of input from a filehandle - STDIN is a filehandle, which represents the standard input (normally 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).

- After reading a line, you usually want to get rid of the trailing newline. Two operators:
 - Perl 4 provided `chop($name)` which removes the last character from the named variable and returns the character removed.
 - Perl 5 added `chomp($name)` which deletes a trailing newline or does nothing.
- 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!).

- `print()`, combined with the variable interpolation in double-quoted strings, gives us a lot of formatting control - although Perl also has a C-style `printf()` function (see `perldoc -f printf`) to give you even better control such as 3 decimal places for `$z`:

```
print "debug: x=$x, y=$y, z=$z\n";
printf "debug: x=$x, y=$y, z=%.3f\n", $z;
```

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

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

- You may like to spend some time trying some examples of string, numeric and boolean expressions.
- Perl 5.10 added a new function 'say' - basically 'print' with a newline added - again, you have to ask for it:

```
use v5.10; # at the top of the program, enable 5.10 features
....
say "debug: x=$x, y=$y, z=$z";
```

- We've seen that `$line = <STDIN>` reads a whole line of input from `stdin`.
- 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: On Unix systems, end-of-input is signalled by the user typing CTRL-D on a line on its own.
- No CTRL-D character (ASCII code 4) is delivered; CTRL-D is a terminal control signal that input has ended.
- When eof occurs, `<>` returns the *empty string*. This never otherwise occurs - why? Because `<>` leaves the newline in!
- So test for eof by comparing the result against the empty string:

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

- More typically, test for *not eof*: `if($line ne "")`
- Or simply `if($line)`.
- Can combine line reading, assignment, and test for not eof into one:

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

- Can even declare `$line` at the same time:

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

- Replacing 'if' with 'while' gives us the **for each line from stdin** idiom:

```
while( my $line = <STDIN> ) # for each line from stdin
{
    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.

- Above the while loop, we initialize: `my $sum = 0;`
- Processing is: `$sum += $line;`
- At the end, add `print "total: $sum\n"`.
- Giving us:

```
my $sum = 0;
while( my $line = <STDIN> ) # for each line from stdin
{
    chomp $line;
    $sum += $line;
}
print "total: $sum\n";
```

- Perl can read the contents of named files (or Unix pipelines), create/overwrite a named file with new contents, and append new output on the end of a file.
- Suppose we wish to read a file called `fred`. The first step is to create a filehandle like `STDIN` but connected to the file "fred":

```
open( my $in, '<', "fred" );
```

If the file "fred" doesn't already exist in the current directory, `open()` fails, setting `$in` to undefined (boolean false) and returning 0 (another boolean false). So you might handle an error by:

```
open( my $in, '<', "fred" );    or    unless( open( my $in, '<', "fred" ) )
unless( $in )
{
    # ... handle error...
}
# ... handle error...
}
```

- A common way of handling the error is to print an error message (on `STDERR`) and exit - use `die`:

```
unless( open( my $in, '<', "fred" ) )
{
    die "can't open fred\n";
}
```

- Better still, use the *Die unless Happy* idiom:

```
die "can't open fred\n" unless open( my $in, '<', "fred" );
```

- Best of all, we can use the common *Do or die* idiom. This wouldn't work without short-circuit evaluation:

```
open( my $in, '<', "fred" ) || die "can't open fred!\n";
```

- Whichever variant we use, after a successful `open()`, `$in` is a filehandle, just like `STDIN`, from which we can read data.
- `my $line = <$in>` reads the next line of input from `fred`. When there is no more input, `$line` will contain the empty string.
- A typical *read every line* program looks like **eg3**:

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

- Aside: As well as `die` there's a function `warn` which prints a message to `STDERR` but doesn't exit.

- To open a file for writing (overwriting the old contents):

```
open( my $out, '>', "bob" ) || die "sob.. can't create bob\n";
```

- Now write new data into `bob` by any number of:

```
print $out "...some data...\n"; or $out->print( "...some data...\n" );
```

- As with reading, don't forget `close($out)` when you finish writing data.

- We can append data to `bob`, instead of overwriting:

```
open( my $out, '>>', "bob" ) || die "sob.. can't append to bob\n";
```

- You can also open a pipe to/from a Unix pipeline for reading/writing:

- `open(my $in, '|-', "ls | sort -r");`

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 `<$in>`.

- `open(my $out, '|-', "expand");`

We can write data to the pipeline: `expand` will see a stream of data coming from its `STDIN`, but it'll be whatever we write to `$out`.

- You can also do more advanced things like open files for random access I/O, open pipelines feeding data in and receiving data back etc. See

`perldoc -f open` and `perldoc IPC::Open2` for more details.