COMP519: Web Programming
Autumn 2014

Perl Tutorial: The very beginning

- A basic Perl Program
  - The first line
  - Comments and statements
  - Simple printing
- Running the program
- Scalar Variables
  - Operations and assignment
  - Interpolation
  - Exercise
- Array variables
  - Array assignments
  - Displaying arrays
  - Exercise

Perl

- Perl was created in the mid 1980's by Larry Wall.
- The language was first dubbed 'Perl' and later it was decided that name stands for 'Practical Extraction and Reporting Language'. (So it's a 'retronym' not an acronym.)
- It's been said that Perl fills a gap between high-level and low-level programming languages.

- Perl is especially useful for processing strings and other textual data.

- Regular expressions provide extremely flexible methods for processing text.

- Perl can be used for CGI programming (as we will see later).

Example programs given in these notes can be found in the directory
http://www.csc.liv.ac.uk/~martin/teaching/comp519/PERL

A Basic Program

Here is a basic Perl program that we'll use to get started.

```
#!/usr/local/bin/perl

# basic01.pl COMP519
# Program to do the obvious

print "Hello world. \n"; # Print a message
```

Every Perl program typically starts off with #!/usr/local/bin/perl
(or possibly #!/usr/bin/perl depending upon local server configuration).

This is often referred to as the “shebang” line and directs the operating system to the location of the Perl interpreter. (There are no spaces in this line.)

Single line comments can be added with the # symbol. (The first line above is not a comment, however.)

A Perl statement must end with a semicolon ;

The print function outputs some information. "\n" gives a ‘carriage return’.

Running the Program

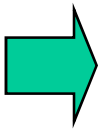
Work under UNIX (Linux) or a suitable editor that can save files in ‘UNIX mode’.

Use a text editor, e.g. vi or Xemacs with Perl mode (use `M-x perl-mode').

Make sure the file is executable by using the command

chmod a+rx progname

To run, type at the prompt



```
bash-2.05b$ perl basic01.pl
Hello world.
bash-2.05b$
```

perl progname

or ./progname (provided the “shebang” line is present)

To run with warnings, use the command

perl -w progname

To run with a debugger, use the command

perl -d progname

Make sure your program works before proceeding.

Scalar Variables

Scalar variables in Perl must start with $ and can store both strings and numbers.

These can interchange, so Perl is not a strongly typed language:

$priority = 9; (but you can later assign a string if you want)

$priority = 'high';

$priority = “high”;

Perl also accepts numbers as strings, like this (but strings are not the same as numbers):

$priority = '9';

$default = '0009';

Variable names = numbers, letters and underscores, but they should not start with a number. Also, Perl is case sensitive, so $a and $A are different.

$_ is a very special Perl variable, and we'll see more about it later.

Operations and Assignment

For arithmetic, Perl includes:

```
$a = 1 + 2;    #  Add 1 and 2 and store in $a
$a = 3 - 4;    # Subtract 4 from 3 and store in $a
$a = 5 * 6;    # Multiply 5 and 6
$a = 7 / 8;    # Divide 7 by 8 to give 0.875
$a = 9 ** 10;  # Nine to the power of 10
$a = 5 % 2;    # Remainder of 5 divided by 2
++$a;          # Increment $a and then return it
$a++;          # Return $a and then increment it
--$a;          # Decrement $a and then return it
$a--;          # Return $a and then decrement it
```

For strings, Perl includes:

```
$a = $b . $c;    # Concatenate $b and $c
$a = $b x $c;    #  $b repeated $c times
                 #  where $c is an integer
```

To assign, Perl includes:

```
$a = $b;       #  Assign $b to $a
$a += $b;      #  Add $b to $a
$a -= $b;      #  Subtract $b from $a
$a .= $b;      #  Append string $b
               #        onto $a
```

Note that when Perl (like most programming languages) assigns a value with $a = $b it makes a copy of $b and then assigns that to $a. Therefore the next time you change $b it will not alter $a.

Other operators can be found on the perlop manual page. Type man perlop at the Unix (Linux) prompt.

Interpolation (substitution) of variables

```
#!/usr/local/bin/perl
# basic02.pl COMP519
$a = 'apples';
$b = 'pears';
print '$a and $b';
print "\n";
```

```
bash-2.05b$ perl basic02.pl
$a and $b
bash-2.05b$
```

```
#!/usr/local/bin/perl
# basic03.pl COMP519
$a = 'apples';
$b = 'pears';
print "$a and $b \n";
```

```
bash-2.05b$ perl basic03.pl
apples and pears
bash-2.05b$
```

Note that print '$a and $b'; literally prints the string $a and $b.

But we can use the double quotes like print "$a and $b"; to get the output

apples and pears

as in the programs above.

The double quotes force *interpolation* of any codes, including interpreting variables and escape sequences (meaning that variables are replaced by their values).

The newline code \n and the tab code \t are interpolated as well inside of double quoted strings.

Exercise

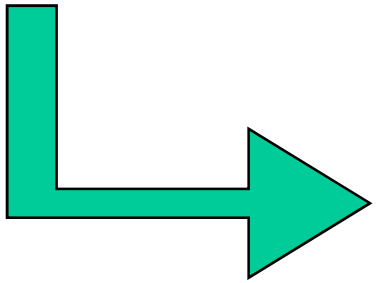
This exercise is to rewrite the Hello world program so that

(a) the string is assigned to a variable and

(b) this variable is then printed with a newline character.

(c) Use the double quotes and don't use the concatenation operator.

```
#!/usr/local/bin/perl
# basic04.pl COMP519
$message = 'Hello world.';
print "$message \n"; # Print a message
```



```
bash-2.05b$  ./basic04.pl
Hello world.
bash-2.05b$
```

Examples

```
#!/usr/local/bin/perl
# basic05.pl COMP519
print '007',' has been portrayed by at least ', 004, ' actors. ';
print "\n";
print 7+3, '  ', 7*3, '  ', 7/3, '  ', 7%3, '  ', 7**3, ' ';
print "\n";
$x = 7;
print $x;
print "\n";
print '   Doesn\'t resolve variables like $x and backslashes \n. ';
print "\n";
print "   Does resolve $x and backslash\n";
$y = "A line containing $x and ending with line feed.\n";
print $y;
$y = "Con" . "cat" . "enation!\n";
print $y;
```

```
bash-2.05b$ ./basic05.pl
007 has been portrayed by at least 4 actors.
10  21  2.33333333333333  1  343
7
   Doesn't resolve variables like $x and backslashes \n.
   Does resolve 7 and backslash
A line containing 7 and ending with line feed.
Concatenation!
bash-2.05b$
```

String Functions

Here are some built-in Perl functions for operating with strings:

Name	Parameters	Result
chomp	(a string)	the string with terminating newline characters removed
length	(a string)	the number of characters in the string
lc	(a string)	the string with uppercase letters converted to lower
uc	(a string)	the string with lowercase letters converted to upper
hex	(a string)	the decimal value of the hexadecimal number in the string
join	(a char. and a list of strings)	the strings concatenated together with the character inserted between them
q	/a string/	places single quotes around the string
qq	/a string/	places double quotes around the string

Array Variables

an array is a list of scalars, i.e. numbers and strings.

Any Perl array variable is prefixed by an @ symbol.

```
@food  = ("apples", "pears", "eels");  # a three element list
@music = ("whistle", "flute");         # a two element list
```

```
print $food[2];                  # prints eels
```

Notice that the @ has changed to a $ because “eels” (a single element of the array) is a scalar!

```
@music = ("whistle", "flute");
@moremusic = ("organ", @music, "harp");    # combines both
@moremusic = ("organ", "whistle", "flute", "harp"); # is the same
@moremusic = qq(organ whistle flute harp);          # is the same
```

Note: Perl only has single dimensional arrays (you can get more dimensions by using references, but I (probably) won’t discuss that in this course).

Push and Pop Functions

`push` is a neater way of adding array elements (to the end)

```
@food  = ("apples", "pears", "eels");

push(@food, "eggs");              # pushes eggs onto the end of @food
push(@food, "eggs", "lard");      # pushes two items
push(@food, ("eggs", "lard"));    # pushes two items (same as above)
push(@food, @morefood);             # pushes the items of  @morefood
                                    # onto the end of list @food
$length = push(@food, @morefood);   # pushes @morefood onto the @food
                                    #   list and $length contains the
                                    #   length of the new list
```

`pop` is a way of removing the last item from a list.

```
@food  = ("apples", "pears", "eels");
$grub = pop(@food);                    # Now $grub = "eels“
```

`shift` and `unshift` work in a similar manner but operate on the front of the list

Array Assignments

Can make an assignment to a scalar variable in different contexts.

```
$f= $#food;          # assigns the index of the last element of @food array
$f = @food;          # assigns the length of @food to the variable $f
$f = "@food";        # turns the list into a string with a space between each element.
                     # This space can be replaced by any other character (or string)
                     # by changing the value of the special (built-in) $" variable.
```

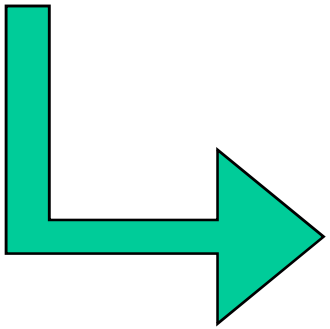
Can make multiple assignments to scalar variables

```
($a, $b) = ($c, $d);                  # Same as $a=$c; $b=$d;
($a, $b) = @food;                     #  $a and $b are the first two items of @food
($a, @somefood) = @food;              # $a is the first item of @food and
                                      # @somefood is a list of the remaining
(@somefood, $a) = @food;              # @somefood is @food and
                                      # $a is undefined, that form is best avoided
($a, $b) = ($b, $a);            # swaps the values of $a and $b  (since
                                #       Perl creates the whole list before the
                                #       assignment statement)
```

Displaying Arrays

Context is important in determining the output!

```
#!/usr/local/bin/perl
# basic06.pl COMP519
@food  = ("apples", "pears", "eels");
print @food; # By itself (no spaces appear between array elements)
print "\n";
print "@food"; # Embedded in double quotes (spaces appear)
print "\n";
$f=@food; # In a scalar context, gives the length of the array
print "$f \n";
```



```
bash-2.05b$ perl basic06.pl
applespearseels
apples pears eels
3
```

The special $_ variable

- Perl has many special (built-in) variables.

- The most commonly used one is the scalar variable that has the name $_

- This is the default variable that is used when an argument is not provided to a function.

The special $_ variable (cont.)

- For example, the function call

```
print;
```

is equivalent to the function call

```
print $_;
```

and the function call

```
chomp;
```

is equivalent to

```
chomp $_;      or     chomp($_);
```

- This built-in variable is a shortcut provided to the programmer and is often heavily used in Perl programs (which can therefore be confusing to new programmers).