

HEP Computing Part II Scripting Marcella Bona

Lectures 2

© Adrian Bevan

Lecture 2

- Introduction to scripts: what are they, how do you write and run them?
- Using bash
- Special Characters.

Scripting

This section covers the idea of putting together use of commands into a unit/block → *i.e. a script*:

- how to write your own scripts
- understand other people's scripts
- know where to get more information

The benefit of scripting is the automation of common repetitive Tasks; so you get

- greater productivity
- time to spend elsewhere doing more interesting things

There are many different ways to get the same result – as long as the script works there is no right or wrong solution – just do/use what works for you.

What should you learn?

Aim to learn two things:

A basic grasp of a **shell scripting** language and something like **PERL**



`bash` – default shell

Other shells exist,
you may come
across some of
these: `tcsh`, `ksh`,
...



PERL

now python is more used
so we'll have two hours
on it. However you can still
encounter perl script.

So what is a script?

A set of commands in a file that are executed sequentially

Basics:

- Start file with a '#!'
- follow this with the path to the program [e.g. bash, tcsh, PERL, python...] use `which` to find a program if you don't know where it is!
e.g. `which perl`
- after this comes the commands that are run
- comments start with '#' and continue to the end of the line
- `chmod u+x scriptname` to change the file permissions so you can run the script
- run using `./scriptname` or if you have '.' in your path just use `scriptname`

Aside: File Permissions

you can check to see if a file is executable using `ls -l`

r = readable
w = writeable
x = executable

```
bfa ~/Lectures/scripts > ls -l
total 24
-rwxr----- 1 bona  zp   1632 Aug 20 13:39 jot*
-rwxr----- 1 bona  zp   231 Aug 19 13:42 loopTest.sh*
-rwxr----- 1 bona  zp   240 Aug 19 13:45 loopTest.tcsh*
-rwxr----- 1 bona  zp   197 Aug 21 15:28 sanr*
-rw-r----- 1 bona  zp   241 Aug 21 15:28 sanr2
-rwxr----- 1 bona  zp   673 Aug 20 13:43 texIt*
```

file size

timestamp of modification

file is executable

User permissions

Group permissions

'Other' permissions
(any user)

group of user
who owns file

user name
of file owner

filename

Aside: Making a file executable

The command `chmod` can be used to change permissions on files and directories. To make the file executable to the user just type:

```
chmod u+x forlooptest.sh
```

```
bfa ~/Lectures/scripts > ls -l forlooptest.sh
-rw-r----- 1 bona  zp    240 Aug 19 13:45 forlooptest.sh
```

 You can only read and write this file

```
bfa ~/Lectures/scripts > chmod u+x forlooptest.sh
bfa ~/Lectures/scripts > ls -l forlooptest.sh
-rwxr----- 1 bona  zp    240 Aug 19 13:45 forlooptest.sh*
```

 Now this can also be executed

The "Hello World" example:

```
#!/bin/bash  
echo "Hello World"
```

```
#!/bin/tcsh  
echo "Hello World"
```

```
#!/usr/local/bin/perl -w  
print "Hello World\n";
```

You **DO** want this with perl!
it gives warnings when you
start to do things wrong

General Comments

bash: have configuration files that are run

- at login
- start of a shell
- log out

These files have different names as given later on. In addition to this, there is a history file that records what commands you have used recently.

I'll assume that your default shell is the bash shell. There are other shells that you may encounter – these all have similar functionality (e.g. ksh, tcsh) but slightly different syntax.

bash – an introduction

bash is a robust shell for general use.

<http://www.gnu.org/software/bash/>

- `#!/bin/bash` - invoke the shell
[if this doesn't work try 'which bash' to find it]
- use `sh/bash` in a production environment to do anything *REALLY* serious

Config files

`.bash_profile`

run at

login (your basic shell setup). Changes will be picked up the next time you log into a machine.

`.bashrc`

shell start up [may not exist, run via the `.bash_profile`].

`.bash_logout`

logout.

Other useful files:

`.(bash_)history`

record of last session's commands.

→ use the `history` command to see the history of shell commands that you have used

For these examples, there is a condition in square brackets: [some condition]

```
if [ condition ]
then
    # do something
else
    # do something else
fi
```

```
for I in <list>
do
    echo $I
done
```

loops

```
while [ condition ] ; do
    #do stuff
done
```

set the value
(can use in scripts)

can export value
after setting it.

set and export variable
to the environment

environment
& variables

```
var=value
export var
export var=value
```

```
until [condition]; do
    # do something
done
```

For these examples, there is a condition in square brackets: [some condition]

```
if [ "$MYVAR" == "yes" ]
then
  echo "yes"
else
  echo "no"
fi
```

```
export MYLIST="a b c d"
for I in $MYLIST
do
  echo $I
done
```

loops

```
while [ "foo" != "bar" ] ; do
  echo "ere I am J.H."
done
```

```
until [ "foo" == "bar" ] ; do
  echo "ere I am J.H."
done
```

environment
& variables

```
var=value
export var

export var=value
```

Some Important environment variables

\$PATH	path to search for apps
\$LD_LIBRARY_PATH	path to search for libs
\$HOME	your home directory
\$USER	your UID
\$ROOTSYS	root install directory
\$SHELL	the shell you're using
\$EDITOR	e.g. emacs
\$PRINTER	e.g. PRINTER=ds

access variables with \$ prefix

Example script: looping in bash

```
#!/bin/bash
# forlooptest.sh
#
# simple example of looping over
# more than one directory
# and performing an action on the directory
```

The shell

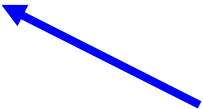


```
for thisdir in `ls -F | grep '/'`
do
    echo "looking at directory $thisdir"
    ls $thisdir
done
```

Run the command in `` to get the list of directories.



Loop over the directories and list the content of each one



forlooptest.sh

Using a script to do work for you

- If I run `forlooptest.sh` from my home directory I get the following output

```
~ > ./Lectures/scripts/forlooptest.sh

looking at directory bin
NNMakeAllCfgFiles*      diagonalize*
icrootqsub*
NNMakeCfgFiles*        makeFlatFile*
NNMakeInputFiles*      fitForPulls.cc
makeSimFitFile*
calculateLikelihoodUL.c  getPullTable*
clean_package*         getPulls*
combineInQuadrature*   getSigTable*
cullDeadRootFiles      getSignificance*

looking at directory scripts
Acrobat.ps              dos2unix*           myCronJobs
CronJpsitollKanga      excludeFile.txt     myCronJobs-
old*
CronTest*               finalize            myPs2Gif*
RandomHacks/            findDeadNodes*     niceJobs*
RecursiveFileSearch.pm  findMissingLines*  opr/
etc.
```

same as doing the commands

```
echo "looking at.."
ls bin
echo "looking at.."
ls scripts
echo "looking at.."
ls analysis
echo "looking at.."
ls tex
```

Getting the output of a command

- If you try to set an environment variable in the following way

```
export TEST=date
```

the value assigned to TEST will be the string 'date'.

- You can use backticks: ``<command>`` to access the output obtained when executing a command in a script.

```
# set MYDATE to have the value
# of the date command's output
export MYDATE=`date`
# set TODAY to be the day of the week
# based on the date command
export TODAY=`date | awk '{print $1}'`
```

You can inspect the values set for the environment variables by typing

```
echo $MYDATE
echo $TODAY
```

Backticks can be extremely useful in scripts!

Shell Scripting Examples

Here are a few examples to work through. You will learn how to

- 1) write your own script and make sure you can run it
- 2) know how to get the output of another command into your script
- 3) loop over a list in a script
- 4) access the command line arguments given to a script

- Write and get working the 'hello world' shell script example shown previously
- Write a script to loop over a list of variables and echo each value (hint see examples).
- Write a script to take arguments from the command line and write an output file containing these.

hint: the variable \$0 is the script name used and \$1, \$2, ... \$n are the n arguments supplied to the command line. see the echo command

Then manipulate the output of the `date` command into a timestamp for the name of the log file

hint: can use `awk` and backticks e.g.: `export mydate=`date``

Examples: 1) Hello World

- Open a file called hello.pl
- enter the following into the file:

check this matches the output of
the command:

which perl

```
#!/usr/bin/perl -w  
print "hello world\n";
```

- Change the permissions on the file so that you can run this:

```
chmod u+x hello.pl
```

- Now you can run the script using:

```
./hello.pl
```

→ tcsh and bash examples can be taken from page 8

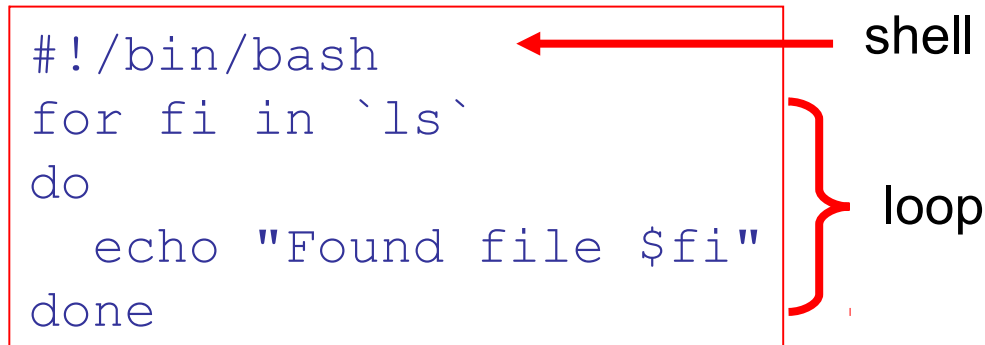
→ As an aside – this is simple & you can do this on the command line;
try typing the following command:

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

Examples: 2) Getting information from the system: bash

- Write a new script called test1.sh and start this off in the usual way:

```
#!/bin/bash
for fi in `ls`
do
    echo "Found file $fi"
done
```



- change permissions so you can run the script and use the command `ls` on the current directory

```
chmod u+x test1.sh
```

and get the output to print. It should look something like:

```
Found file hello.txt
Found file jot
Found File loopTest.sh
.
.
```

listing the content of your current directory

Examples: 3) more looping

- Use a while loop to count from 1 to 10 in a script.

index variable

```
mycounter=1
```

start index of loop

```
while [ $mycounter -lt 11 ]; do
```

loop condition

```
    echo The counter currently has the value $mycounter
    let mycounter=mycounter+1
```

The let command carries out arithmetic operations on variables

```
done
```

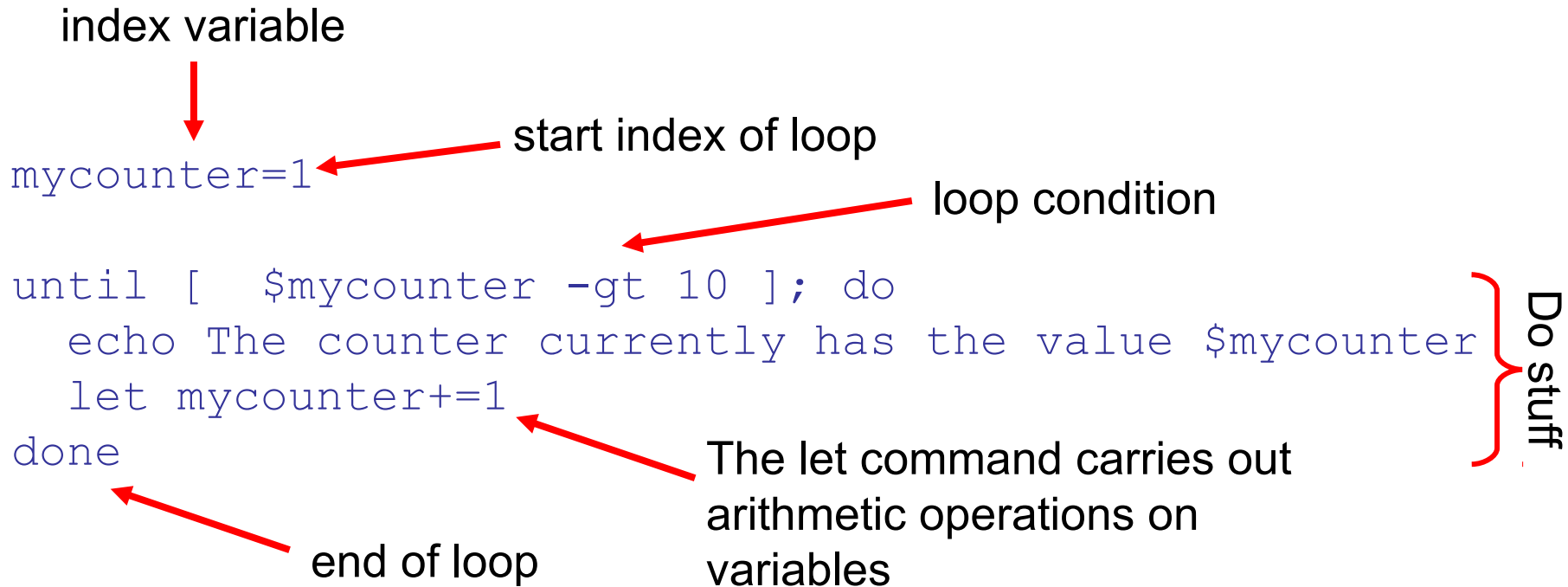
end of loop

Do stuff

```
whilelooptest.sh
```

Examples: 4) more looping

- Use an until loop to count from 1 to 10 in a script.



Very similar to the while loop.

`untillooptest.sh`

Examples: 5) Using command line arguments

printargs.pl

Aim: Want to parse arguments to a script e.g.:

```
./myScript a b c
```

so that the script can use 'a', 'b' and 'c' to do stuff

- Start off in the usual way – open a new file and enter:

```
#!/bin/bash
```

```
#!/usr/bin/perl -w  
use strict;
```

- Then change the permission to run the file and add the commands to print out the input arguments:

```
echo "1st Argument $1"  
echo "2nd Argument $2"  
echo "3rd Argument $3"
```

```
foreach my $iarg (@ARGV)  
{  
    print "$iarg\n";  
}
```

The command line arguments are `$n` for shell scripts

Similar to C, C++, use variable ARGV to get arguments. ARGV[0] is the first argument and \$0 is the script name.

Examples – use of scripts etc.

You want to run `forlooptest.sh` and put the output into a file:

```
forlooptest.sh > test.txt
```

redirect the output of the command
`forlooptest.sh` into the file `test.txt`

You want to print the day of the week , month of the year and the year only from the `date` command:

```
bfa ~ > date
Mon Oct 18 15:52:28 BST 2010
bfa ~ > date | awk '{print $1 " " $2 " " $3 " " $6}'
Mon Oct 18 2010
```

The pipe '`|`' means take the output of the first command and pass it to the second command

You want to append one file to the end of another:

```
forlooptest.sh > test.txt  
date | awk '{print $1 " " $2 " " $3 " " $6}' >> test.txt  
cat somefile.txt >> test.txt
```



The >> operator appends information to the file test.txt

You can see that the special characters

|, > and >>

that you've just been introduced to are quite useful in writing log files of events that happen when commands are being executed. There are a number of these listed on next page.

Special Characters & useful syntax

To get the most out of scripting you'll need some background information

→	>	redirect output	←	redirect into a file
→	>>	append to output file		
	<	redirect input		
	<<	'here document' (redirect input)		
→		pipe output	←	take the output of one command and pass it to another
→	&	run process in bkgnd		
→	;	separate commands on one line		
→	?	match single character		
→	*	match any character(s)		
→	`<command>`	substitute for output of <command>: "back-ticks"		
	\$\$	process ID number of a script		
	\$0	command name		
→	\$n	argument n		
→	\$var	variable		
→	#	comment		

Useful resources are
UNIX Power Tools
LINUX in a Nutshell
but these are more in depth
than you'll need for quite a while

Wildcards and pattern matching

```
*      match all
?      match to any single character
```

e.g.

`ls *.txt` list all files with a `.txt` extension

```
[tersk01] ~ > ls *.txt
12seriesCheck.txt      markus-tagging.txt      sxf.float.txt
7bbgndresults.txt     pipi.txt                systematics.txt
bad-521.txt            productionMC.txt        tau.txt
correlations.txt       quinn.txt               test.txt
crossFeed.txt          rad_ll.txt              twoBodyModes.txt
dataCardVMassHelData.txt* resultSummary.txt       unblindResults.txt
deChecks.txt           rr.txt                  validation_25_06_03.txt
ee.txt                 rr_to_do.txt            validation_27_06_03.txt
```

`ls ?r*.txt` list all files with extension `.txt` and `'r'` as second character in the file name

```
[tersk01] ~ > ls ?r*.txt
crossFeed.txt      productionMC.txt  rr.txt      rr_to_do.txt
```

Regular Expressions

"Regular Expression" is commonly abbreviated as regex, regexp, re or RE.

A RE is a text string pattern matching method with wild cards.

REs can be used on the command line and in UNIX filters like sed, awk, grep, egrep and perl, and in text editors like vi, emacs and in high level languages like C, Fortran, and Java.

Eleven characters with special meanings:

the opening square bracket [, the backslash \, the caret ^, the dollar sign \$, the period or dot ., the vertical bar or pipe symbol |, the question mark ?, the asterisk or star *, the plus sign +, the opening round bracket (and the closing round bracket).

These special characters are called "metacharacters".

If you want to use any of these characters as a literal in a regex, you need to escape them with a backslash.

Regular Expressions II

A "character class" matches only one out of several characters.

To match an a or an e, use [ae]: gr[ae]y to match either gray or grey.

A character class matches only a single character. gr[ae]y will not match graay or graey. The order of the characters inside a character class does not matter.

You can use a hyphen inside a character class to specify a range of characters. **[0-9] matches a single digit between 0 and 9.**

You can use more than one range. [0-9a-fA-F] matches a single hexadecimal digit, case insensitively. You can combine ranges and single characters. [0-9a-fxA-FX] matches a hexadecimal digit or the letter X.

Typing a caret after the opening square bracket will negate the character class. **The result is that the character class will match any character that is not in the character class.**

q[[^]x] matches qu in question. It does not match Iraq since there is no character after the q for the negated character class to match.

Regular Expressions III

\d matches a single character that is a **digit**,

\w matches a "**word** character" (alphanumeric characters plus underscore),

\s matches a **whitespace** character (includes tabs and line breaks).

The actual characters matched by the shorthands depends on the software you're using. Usually, non-English letters and numbers are included.

You can use special character sequences to put non-printable characters in your regular expression:

\t to match a **tab** character, **\r** for carriage **return** and **\n** for **line** feed.

Remember that Windows text files use `\r\n` to terminate lines, while UNIX text files use `\n`.

The dot matches a single character, except line break characters.

It is short for `[^\n]` (UNIX regex flavours) or `[^\r\n]` (Windows regex flavours). `gr.y` matches `gray`, `grey`, `gr%y`, etc..

Regular Expressions IV

Anchors do not match any characters. They match a **position**.

^ matches at the **start** of the string, and **\$** matches at the **end** of the string.

\b matches at a word **boundary**. A word boundary is a position between a character that can be matched by **\w** and a character that cannot be matched by **\w**. **\b** also matches at the start and/or end of the string if the first and/or last characters in the string are word characters.

\B matches at every position where **\b** **cannot** match.

Alternation is the regular expression equivalent of "**or**".

cat|dog will match **cat** in **About cats and dogs**. If the regex is applied again, it will match **dog**. You can add as many alternatives as you want, e.g.: **cat|dog|mouse|fish**.

The **question mark ?** makes the **preceding token** in the regular expression **optional**. *E.g.: **colou?r** matches **colour** or **color**.*

Regular Expressions V

The **asterisk or star** tells the engine to attempt to match the **preceding** token zero or more times.

The **plus** tells the engine to attempt to match the **preceding** token once or more.

<[A-Za-z][A-Za-z0-9]> matches an HTML tag without any attributes. <[A-Za-z0-9]+> is easier to write but matches invalid tags such as <1>.*

Use **curly braces** to specify a specific amount of **repetition**.

Use `\b[1-9][0-9]{3}\b` to match a number between 1000 and 9999. `\b[1-9][0-9]{2,4}\b` matches a number between 100 and 99999.

Place **round brackets** around multiple tokens to group them together.

You can then apply a quantifier to the group. E.g. `Set(Value)?` matches `Set` or `SetValue`. Round brackets create a capturing group.

Regular Expressions VI

example with emacs query-replace-regexp:

searching for repeated words: *the expression "[a-z][a-z]" will match any two lower case letters. If you wanted to search for lines that had two adjoining identical letters, you need a way of remembering what you found, and seeing if the same pattern occurred again.*

You can mark part of a pattern using "(" and ")". You can recall the remembered pattern with "\" followed by a single digit.

Therefore, to search for two identical letters, use "\"([a-z])\1".

You can have 9 different remembered patterns. Each occurrence of "(" starts a new pattern.

The regular expression that would match a 5 letter palindrome, (e.g. "radar"), would be \"([a-z])\"([a-z])\1\2\1

back-up - PERL

PERL – an introduction

- PERL is more powerful than either tcsh or bash
- Supports Object Oriented programming paradigm
- large community base – modules
 - CGI – web forms/html generation etc
 - DB connectivity: MySQL etc
 - POSIX / Networking
 -
 -
 - you name it – there is probably something there to help you
- REGEXP engine – powerful pattern matching/substitution
- In a nutshell – PERL is a language to glue everything else together for you

<http://www.perl.com>
<http://www.perl.org>
<http://www.perlmonks.org>

The Aim of this part of the course is to give you a crash course in PERL.
In particular the following topics introduced are

variable types

accessing the system

what a simple PERL script looks like

Getting at the command line arguments

printing in PERL

Some example scripts

What a simple PERL script looks like

```
#!/usr/bin/perl -w
use strict;

#scalars
my $var = 5;
my $name = "wibble";

#array
my @arr = (1.0, 2.0, 3.0);

foreach (@arr)
{
    my $num = $_ + $var;
    print "$_ \t $num\n";
}
```

Gives warnings – you **SHOULD ALWAYS USE THIS**

checks for declared types – like FORTRAN's 'implicit none'

declare variable using 'my'

semi-colon terminates line [not necessary for last line in a block]

special character

similar special print characters to C/C++
\t = tab, \n = new line character

Basic types

Scalar variable

- Starts with a '\$'
- can be a number or a string ...

```
$num = 5;  
$name = "wibble";
```

Array variable

- Starts with a '@'
- null initialiser: @arr = ();
- can push onto/pop off of a list:

```
push(@arr, $var); pop(@arr);
```

```
my @arr1 = ( 3, 4, 5, 6, 7 );  
my @arr2 = ( "spam", "larch", "parrot" );  
print "$arr[2]\n";  
$arr1[0] = 1;
```

- counting of array index starts from 0 just like C/C++ etc.

Hash Variables

this is the PERL equivalent of a **MAP**

an “associative container” – look up wrt. STL

- Starts with a `%`
- associate a key with a value

```
my %options = (  
    "parrot" => "The dead parrot sketch",  
    "larch"  => "A tree",  
    "spam"  => "random stuff"  
);
```

keys values

```
foreach (sort keys %options)  
{  
    print "\t$_ $options{$_}\n";  
}  
print "\tkey = parrot value = $options{parrot}\n";
```

get the keys for this hash

access the value corresponding the key parrot

Printing in PERL

There are a couple of commands for printing in PERL

```
print "some info\n";  
printf "some info: %5.2f\n", $var;
```

print to screen

and you can easily print to a file:

```
open(OUT, ">outputfile.txt");  
  
print OUT "some info\n";  
printf OUT "some info: %5.2f\n", $var;  
  
close (OUT);
```

formatted
print statement

Some format characters for print (similar to C)

't' = tab

'n' = new line

'a' = a system beep

Running system commands in perl

System calls in Perl use sh as the default shell

```
exec "sleep 5; ping somehost"
```

fork a process to run the command and carry on executing the script **WITHOUT** waiting for the outcome of the command

```
system "ping somehost"
```

Execute the command **AND WAIT** for the system to return control to the script

```
my @data = `grep somestring myFile.txt`;
```

Like system – but get output redirected into a local variable (as an array) – same as for tcsh/bash etc

you can then remove the end of line characters from the array variable

```
chomp (@data);
```


Getting input from the command line:

- The easy way is to pop inputs off of the bottom of @ARGV:

> ./mmyScript wibble hat (the command)

```
#!/usr/local/bin/perl -w (The script)
use strict;
my $in1 = shift;
my $in2 = shift;
print "$in1\n\t$in2\n";
```

> ./mmyScript wibble hat (The output)

```
wibble
    hat
```

a more robust way to deal with
command line input is to
use `Getopt::Long`;

Examples: 2) Getting information from the system: PERL (This is the PERL solution for example #2)

- Write a new script called test1.pl and start this off in the usual way:

```
#!/usr/bin/perl -w
use strict;
```

- change permissions so you can run the script and use the command `ls` on the current directory and get the output to print:

run command and get the output in a local variable

just like sh or tcsh – use back-ticks!

```
my @data = `ls`;
chomp (@data);

foreach my $line (@data)
{
    print " $line\n";
}
```

Get rid of new line characters in data (don't need this here, but it is useful to point out now)

The for loop to print out each line that you got back from `ls`

PERL Exercises

- 5) Write a script to add together two numbers and print the output.
 - Extend this to take two input numbers instead of having this hard-coded in
- 6) Write a script to count the number of lines in a file
 - Extend this to print the file with the line number prepending the line
- 7) Write a script to execute a command on each file in a directory and loop on this printing the file name and number of lines per file as you go.
- 8) Write a script to run loopTest.csh and print the last line of the output to the screen. [n.b. you can use **backticks** for this if you are really lazy].

Example 5: A perl script to add together two numbers

```
#!/usr/local/bin/perl -w
use strict;
my $num1 = 5.2;
my $num2 = 7.3;
print $num1+$num2, "\n";
```

hardcode numbers
into script



```
#!/usr/local/bin/perl -w
use strict;
my $num1 = shift || die;
my $num2 = shift || die;
print $num1 + $num2, "\n";
```

get numbers
to add from
command line

Two new concepts:

`die` → if you get to this part of the script then perl dies...

`||` → this is an OR. If there are <2 arguments passed to the script, it dies

Example 8: Print the last line from the result of running ls.

```
#!/usr/local/bin/perl -w
use strict;
my @data = `ls`;
chomp(@data);

print "the last line output from ls is\n";
print "$data[-1]\n";
```

This is a foolproof way of getting the last element of an array in perl: use the array element `[-1]`

It only fails if the array is null.

- Now that you have done the examples, you should note that exercise 6 was a waste of your time ...

- there is a unix command called `wc`.

```
wc -l <filename>
```

prints out the number of lines in the file...

- This is a common lesson to learn ... if you are trying to do something that is and obvious generic problem ... then most probably either

- there is (at least) a (single) command to do this already
- there is going to be more than one way to solve the problem
- someone you work with knows/has a solution to the problem already