Notes originally for UNIX, changed some

- Linux is largely similar
- There is more variation among shells c/zsh etc.
- Good to appreciate both, as both will be encountered
- Mac OS X is still UNIX
  - BSD (Berkeley Software Distribution)
  - Commercial spin off from Berkeley
- Scripts were originally tested on OS X years ago
  - There was no easy syntax highlighting compatible with Micro$oft Powerpoint at the time, so just screenshots
- And it was originally presented to a group (Higher Dip.)
  - in their first year of programming
  - So a lot about simple basic thinking about planning programs
    - Left for reference, reminder...

Just a little more pain, then the gain!

- 2-4 lectures of script programming, then on a roll!
- Initial emphasis on daft traps to save confusion, frustration and pain, but which generally can be ignored when you know to avoid them.
- Why… because the old scripts are still in use…
  - So need to look at some of the old conventions and pitfalls
  - Developed locally on a Mac at home (*within city limits, but crap Eir comms up to last year*), so UNIX quirks not Linux
    - M$ did not have syntax colouring for code at the time, so screenshots
    - But adds to you polyglot ability - or part of a broader appreciation!?
- What you've already done:-

Some basic commands	How to use the manual
Redirection & pipes	Regular expressions
Awk	sed, grep, find etc.

- What we're about to do
  - How to write programs the easy way, combining all of the above for ease!!
- What then?
  - Anything & everything: admin, from scripts to GUI's
  - But Inter Process signaling is done in following modules, and can be challenging, even for the experts. May view briefly.

The who.. who is logged on.. Etc..

```
#!/bin/bash
# the simple command per line way... go for it!
# can have many commands on a line, just separate with semicolons ;
echo "Today's date"
date   # just prints today's date
echo "Who is logged on now!"
who    # just prints who is currently on
echo -e "\n\n\n ************************ \n\n\n"
# or jam all commands on a single line ... needs more care,
# can also use semicolons, if not using cmds within o/p string
echo -e " `date` \n \t\tWho is logged on now!\n`who`\n\n "
# should only show others if all logged on the same machine.
# but may be reconfigured ... like nearly everything else!
```


Old ways – for short code sequences with little reuse or OO

- Define the problem
- Outline the solution
- Develop the outline into an algorithm
  - Found this helps many develop their algorithms:
  - How would you do it manually with a paper system?
    - values as cards/blocks for moving on a grid –

    (Still has speed advantage over GUI's e.g. ATC – carrier deck)
  - Test (i.e. desk check on paper) for correctness
- Code the algorithm into a specific language
  - Optionally Test (i.e. desk check on paper) for correctness
- Run the program on the computer (test case date / computer)
  - Debug the code for correctness –
  - may need to repeat much of previous steps
  - Document and maintain the program

Problems…& (problems with) programming paradigms!

- Study problem until you thoroughly understand it - might have to analyse and think
  - May involve writing a problem description yourself – surely not…
- Problematic Paradigms…
  - (Old non-OO) Programs = Algorithms + data structures – way to go for small programs
  - Primary emphasis on algorithm working on data
  - Note : algorithms & data are still inextricably linked.
  - Primarily a functional logical division
  - Easier & quicker for smaller programs
- (Newer OO) Programs = Objects + (assoc.) methods. - thought it was way to go for big ones
  - Primary emphasis on Objects : data with permitted operations
  - Methods still inextricably linked to data, but only within object
  - Provides better independent encapsulation of data & method
  - Object reusability.. well suited to 'programming in the large'
  - But extra design effort in deciding object division & structure
  - And extra redesign effort in refactoring object division & structure.
  - - now they realise it's a NO-GO! - more bureaucratic rubbish to refactor – a no brainer
- Scripts : just Interactive Hacking & jamming : convert this data to that at CLI, store & go-on!

Outline a Solution – the older, simpler, shorter – non-OO way!

- Divide the problem into three separate components
  - Inputs
  - Outputs
  - Processing steps needed to produce the outputs
- Once the problem is understood, break the problem into smaller tasks or steps and outline a solution
- Outline may include:
  - Major processing steps
  - Major subtasks (if any)
  - Major control structures (loops)
  - Major variables and data structures
  - Mainline logic

Develop Outline into an Algorithm

- Expand the outline that was developed into an algorithm
- So what's an algorithm?
  - A set of precise steps which describe exactly the tasks to be performed and the order in which they are to be carried out
  - An algorithm must:
    - Be lucid, precise, and unambiguous
    - Give the correct solution in all cases
    - Eventually end! (…but what about the halting problem!?)
- Algorithm may be expressed and developed in pseudo-code, a free-form highly structured natural human language, which reflects :-
  - The main concepts and steps of the algorithm at the relevant level of detail appropriate to the design / presentation stage
  - The final target implementation language
  - And human readable natural language
- Pseudocode is a program expressed informally in your own words, while retaining logical structure

Pseudo-Code

- Procedure is written in structured English, (or any natural human language) with little regard to the final programming language
- Each instruction is written on a separate line
- Keywords, parenthesis and indentation etc. used to signify control structures (if-then-else, do-while, while, until)
- Each set of instructions is written from top to bottom
  - with only one entry and exit – to avoid spaghetti logic junctions
  - Exceptions and breakouts are handy,
    - just jump right to exit point.
- statements may be
  - grouped logically
  - combined into modules
  - given a name,

  so they can be called by name!
- Simple design generally omits, but doesn't exclude object-oriented approach

Desk check logic – with a peer ?

Probably the only point worth making to those who already program!

The most critical step, and most often neglected
- (although documenting is a close second :)

❑Desk check the algorithm to find logic errors early
- Errors found now are (relatively) easy to correct
- After your code is written, it becomes much more difficult
- Probably quicker overall than using a debugger
  - Prejudice : “I’m right, or at least thinking right!”
  - Context : “ can’t see the wood for the trees!” - with bugs (on bugs!)
  - “fog of war” : risk of confusion, crossed lines of logic & value setting

Then the obvious – after many years ‘two heads are better than one!’

❑Joint desk check, aka ‘code review’ with a colleague
- you play computer, (preferably with an understanding colleague!?)
  - walking through the algorithm with test data - just as a computer would,
  - keeping track on a sheet of paper of
    - Logic flow
    - major variables

❑Pair programming
- Do it together on the computer, checking logic & decisions, either chaotic or correct
- spreading skills & knowledge, learning from each other, all rise, and nobody critical

Code the Algorithm, run the code.

- ❑ Code the Algorithm
  - After the previous steps have been successfully completed, code the resultant algorithm in the desired implementation language
- ❑ Run the program on the computer
  - Fix the syntax .. .spellings, typos, mismatched parentheses & words
  - Test the logic … again.. Using …
    - Trace the logic – through the major modules… does it flow right?
    - Test the values – using debuggers with breakpoints, etc.
    - Assume it’s wrong until proven right...
    - Check results and False assumptions: it can be wrong, even if
      - – It runs
      - – Gives reasonable looking results
  - Using test data and test cases, verify proper operation
  - Push it to the limits… daft input values & sequences, numeric ranges
  - If errors are found, debug, fix, and re-execute the test
- ❑ This is the usually the most rewarding step of the entire process

Document and Maintain !!?? ;-)))

- ❑ A vital step in the development process, although often overlooked (or avoided!)
- ❑ Even the best programmer forgets what he wrote and how it works after time
- ❑ Documentation includes:
  - Internal documentation (module headers and code comments)
  - External documentation (test data, design documentation, user documentation)
- ❑ suggestion… (Knuth also: ‘Literate programming!”
  - Code your algorithm documentation first
    - as comments
    - And assertions – write a function if language doesn’t do assertions!
  - And then write the real code within the algorithm documentation
  - Can become a web of confusion, but tools exist to manage… incl Leo

Preliminaries :– incompatibilities + sensitivies => incomprehensibilities

- ❑ You are not expected to remember all of this,
  - Most of the time you will not need or encounter this mess
  - But you might know what to check if stuck.
- ❑ Incompatibilities
  - Flavours of ‘nix
    - BSD
    - Linux
    - Solaris
- ❑ Shells & commands
  - Even / especially the common & simple (just like natural language!) : echo, printf, quotes, arguments.
- ❑ Regex
- ❑ All of the above...

Unix Shells are:-

- Interactive - the user interface or set of programs used to interact with Unix and process commands
- Interpreters – interpret and execute one command at a time, parsing & passing arguments, etc.
- **Incompatible : C & Bourne derived families**
- Common shells are:
  - Bourne – standard across all 'nix es
  - C
  - Korn
- When a shell is executed, it
  - inherits environmental variables set by the previous shell
  - stops the previous shell
  - becomes the current shell

Shell's can…

- have new ones installed by admin
  - If binary files available,
  - Or if source code available, then download, compile & install
  - Else if code is proprietary & unavailable, then no go
- Be identified & indicated by the SHELL variable
  - echo $SHELL
- be invoked and changed by user
  - By invoking shell name e.g. **tcsh** or **exec tcsh**
  - By using **ch**ange **sh**ell command **chsh tcsh**
- Shell '***sh***' & '***c***' derived families are incompatible
- Lots new shells in dev.:- fish, dash, - still with issues

Bourne Shell (sh)

- Written by Dr Steven Bourne of Bell Labs
- Both a command interpreter and a high-level programming language
- Typically the default Unix shell
- Fast
  - Approximately 20 times faster than C shell because it is simpler and doesn't carry as much user-friendly baggage
- ***sh*** is used in these notes, but ***bash*** in final code.
- ***bash*** is
  - more common, and used in the labs.
  - Mostly backwardly compatible with ***sh***
  - So whatever is in the notes for ***sh*** should work for ***bash***

Main (incompatible) shell families – C & Bourne derived

- Bourne ***sh*** derived backwardly compatible:
  - ksh - Korn shell - again after developer
  - pdksh - public domain Korn shell
    - Command line editing
    - Built-in arithmetic data types
  - zsh - Z shell
  - bash - Bourne again shell
    - the most common & powerful of all, since it incorporates some better 'C-family' shell features
- C-family : syntax like C language, hence name
  - csh - C-shell -
  - tcsh - most commonly used extension
    - Has lots of user-friendly features, which make it slower

Main Linux (not Unix) shells

- bash - default prompt is $
  - Most popular & powerful Linux shell
  - bourne again shell - after original
- csh & tcsh - default prompt %
- Shell '***sh***' & '***c***' derived families are incompatible
  - Each has strengths & weaknesses
    - So depends on what you want to do…
  - Most common extensions:-
    - bash
      - more widely available
      - Includes some csh & tcsh features
      - Quite powerful and popular
    - tsch
      - handier for loops

Default Prompts …(PS1 - PS4) can be changed…

- The defaults are
  - PS1 : $ usual
  - PS2 : > when a command awaits more input
  - PS3 : prompt for interactive menu input
  - PS4 : + indicates indirection level in tracing

The default primary 'cs1' prompt is : urid1@cs1:~$

Which can be shown by 'echo $PS1' to be: \u@\h:\w\$

\u => user id

\h => hostname up to first '.'

\w => working directory, $HOME abbreviated to '~'

- Lots more variations available incl. colour
- But more advanced options exist; see : UI_construct_prompt

prompting can be done in any way imaginable,
be it plain text, through dialog boxes or from a cell phone.

Scripting & PATH variable

- The Bourne shell is what we will use for scripting
  - It's faster (remember? Why?)
  - It's portable
    - Virtually every Unix system has *sh* installed
  - It has a rich set of programming constructs
- The PATH variable
  - holds the directories listed in sequence, separated by :
  - which are searched in sequence to find the executables (fixes priority of same name cmds)
  - which run the commands!
  - e.g. in cs1 : $ echo $PATH
    /users/2020/abc123/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/users/software/utils

NB changing the sequence changes the search order, so that placing your own directory earlier, can have your scripts override system ones.

DANGER : faulty scripts are a hackers highway!?

3 basic ways to run a script…

1 **Bash – for a quick-fix!? i.e. $bash myscript**

man entry on 'bash' – since

- used be one of the best references (not tutorial) on scripting.

Bash is interactive;

- runs a new shell (handy but excessive) with commands following (even if in a file).

(So no need to change file modes/permissions or place in a dir in PATH*

Otherwise … for any of the following...

Get ready to go…make the script executable

chmod u+x myscr… or chmod 700 myscr

(you could make it 755, 751, 711, 710 etc. or even 001, but couldn't edit it!)

And then

2 **Either** … **get on your marks i.e. $./myscript** (**./** - like a starting block?)

- Precede the filename with **./** - tells shell to use current dir/scriptname

3 * **Or** … **get on your PATH** – next slide

3 – script bins for users to use/share

- Just for yourself ~/bin
  - Usually need to make it first : mkdir ~/bin
  - $HOME inserts users HOME pathname e.g. $HOME/bin
    - And cd without an arg defaults to $HOME

- For future reference,
  when you've admin rights/approval, system privilege & responsibility,
  look up the manual for your implementation but generally as follows
  (although you may not have superuser privileges) :

  - For yourself & other general local users
    - */ usr /* local / bin
  - For superuser & root
    - *I usr / local / sbin*

Put a script in ~/bin in your PATH

- Ignore all legacy dirs in the cs PATH for ease
  while most appear empty, avoid changes mid-term
- So this is the start of PATH before
  - $ echo $PATH
    - /usr/local/bin:/usr/bin: ... ...
- Modification by prepending $HOME/bin
  - $ export PATH="$HOME/bin:$PATH"
- So that afterwards $HOME/bin is prepended
  - $ echo $PATH
    - /users/2020/abc123/bin:/usr/local/bin:/usr/bin: ... ...
- the shell will preferentially run executables first in PATH
  - First encountered is run, with no more searching PATH

Put a script in ~/bin in your CS acct PATH

- Remember to ensure the script is executable
  - Chmod u+x chmod 7?? etc (NB ? Indicates your choice, not just a ' ? ' !
  - (depending on access rights you want to give – read, write=*hack*, execute!?)
- Create a bin in your local home directory,
  - $ mkdir ~/bin
  - Place a (fairly well tested/developed) script there
- Put ~/bin in your PATH (- the sequence of directories checked by shell for a cmd)
  - Prepend (i.e. put first) to your home bin into your path
  - This means that any commands earlier in PATH will take precedence,
  - (as that determines the search sequence, once found it stops searching further)
  - strongly recommend NOT using standard command names to avoid chaos and non portable scripts... unless you want to run them instead of system ones.
- Either interactively $ export PATH="$HOME/bin:$PATH"
- Or in a shell startup or login script … a following slide

Bourne Shell – files, from the bash manual – changes over time

/bin/bash
    bash shell executable - i.e. the shell program
Systemwide
  /etc/profile
      systemwide initialization file, executed for login shells
  /etc/bash.bashrc
      systemwide per-interactive-shell startup file
  /etc/bash.bash.logout
      systemwide login shell cleanup file, executed when a login shell exits
Personal
  ~/.bash_profile
      personal initialization file, executed for login shells
  ~/.bashrc
      individual per-interactive-shell startup file
  ~/.bash_logout
      individual login shell cleanup file, executed when a login shell exits
  ~/.inputrc
      Individual readline initialization file, settings for interactive shells:- bash, py

Personal login/startup file use & sequence

- Obviously, to login the system has started
- and when a user logs in runs the system-wide startup files
- then moves on to personal startup files which are
  - Searched in this order, depends on distro & config (others exist!)
  - And the first found is executed, search stopped, so others ignored

  ~/.bash_profile ~/.bash_login ~/.profile
- but each new non-login bash shell which runs or starts (from) a cmd, runs ~/.bashrc
  - Often now the only personal startup file... e.g. Ubu20:04
- But in csgate, modify the ~/.bash_profile as it's first on hit list!

  Remember to restart or rerun your profile to reset PATH

  $ source ~/.bash_profile
- Make a backup before editing... cp .bash_profile .bash_profile.bak

What is going on?

1. Comments in code
   a) So the reader / programmer / tester / maintainer has some idea
2. Output as code executes – echo, printf
   a) So that the following have some idea
      a) Initially the developer / tester / maintainer
      b) Finally the user
3. Trace & Debugging – more later (~23 slides)
   - -nxv (syntax, execute, verbose)

   a) At command line
   b) Using **set** within the program => effectively breakpoints
   c) At **shebang** line – during initial development
      - Just follow shell invocation with them… #!/bin/sh -vxn

Comments & the whole '#shebang!"

- # is the comment symbol in the shell
- # can occur anywhere on a line
  - The shell ignores anything following a # until the end-of-line
- One special exception … '#shebang!' #-hash !-bang
  - #! on the **first line** is used to tell the shell what program to use to interpret a script file
  - Examples:
    - #!/bin/sh
      - use Bourne shell to execute script
      - Will generally execute most bash scripts also since invoking sh often defaults to bash anyway..
    - #!/usr/bin/bash - use bash shell to execute script
      - Can also use python, perl, awk, and so on

I/O - echo…– it echoes the keyboard on screen

- Should use print(f) standard POSIX but everyone uses echo
- prints its arguments separated by single spaces followed by a newline.
- If an unquoted variable contains characters present in **I**nternal **F**ield **S**eparator $IFS, then the variable will be split on those characters:

  $ list="a b c d e f g h"
  $ echo $list
  a b c d e f g h

  NB no spaces around assignment
- If the variable is
  - double quoted, all internal characters will be preserved:
    - $ echo "$list" a b c d e f g h
  - single quoted, just the string within the quotes is printed
    - $ echo '$list' $list
- Will see more about **$IFS** (~+20 slides), but can save lots of code to have the shell accept or reject certain separators.
- Try viewing with echo **$IFS** to see what you get !!! (Hint: Pipe to xxd)

Echo – **BUT** like sound each echo is slightly different

- echo is the primary way to perform output from the shell
  but it is non-standard POSIX and implementation dependent
  bash (Ubuntu 10.04 & 20.04 LTS) it throws a NEWLINE by default.
  - echo 'A newline automatically follows …unless…'

  it is overridden by
  - -n as an argument with echo;
    echo –n 'not throwing a newline here either'\
  - \c at the end of the output string: (also discards rest of line)
    if interpretation of backslash escapes is enabled…:
    echo 'no newline here\c'
- Some useful arguments…(is there ever a useless one!?)
  - -e enable interpretation of backslash escapes
  - -E disable interpretation of backslash escapes (default)
- Syntax: *echo arguments*
  - arguments - may be variables or explicit strings .. as above..
- To suppress output from shell commands in a script
  redirect output to:- > /dev/null
  - it's the bin/trash device

```
$echo echo
echo
$

$ echo -n 'echo e\cho'
echo e\cho$
$ echo -n echo e\cho
echo echo$

$ echo echo e\cho
echo echo
$
$echo -e echo e\cho
echo echo
$
$ echo -e 'echo e\cho'
echo e$

$ echo a b c d
a b c d
$ echo a b  >/dev/null
$
```


Echo – but each echo is slightly different -!?

- In the early days of Unix, two different versions of echo appeared.
  - One version (AT&T) converted escape sequences,
    - e.g. **\t** and **\n**, into characters they represent in the C language; **\c** suppressed the newline, and discarded any further characters.
  - The other (BSD) did not convert escape sequences
    - used the **–n** option to suppress the trailing newline.
- Bash – does both in one way or another
  - -e option activates escape sequences such as \c but by default uses –n to suppress newlines
  - xpg_echo option (X/Open Portability Guide)
    - Which can be turned on or off : shopt –s xpg_echo
      - » Either interactively in the shell
      - » Or automatically in a script
    - Or the shell can be compiled with either option

so echo'es in bash (as in real life) are inconsistent!

POSIX (Portable OS Intervace based on UNIX) standards

(M$ complies to some extent for interoperability, but really for $!?)

The POSIX standard for echo states:

"Implementations shall not support any options"
and "If the first operand is –n,c
or if any of the operands contain a backslash ('\') character,
the results are implementation-defined."

- Can't rely on echo's behaving as one or the other.
- best not to depend on echo in these situations – but we all do
  - unless you know exactly what echo is going to print,
  - and you know that it will not contain any problem characters.
- The preferred command is C-program language (not shell) derived *printf*
  - As we used with awk earlier.
- for exact information check manual (man printf) as implementations vary, e.g. printf in bash in BSD & Linux differ.

Printf – recommended but tradition!

- derived from C-language
- may be built into the shell itself,
- or it may be an external command
- Recommended best-practice standard, since echo isn't…!!!???
- but tradition and convenience of echo … it just echoes … on and on…!

Like the C-language function on which it is based, printf takes

- a format operand that describes how the remaining arguments are to be printed,
- and any number of optional arguments.

The format string may contain

- literal characters,
- escape sequences,
- and conversion specifiers..

Printf in bash

- Escape sequences - the most common ones being
  - \n for newline,
  - \t for tab,
  - \r for carriage return) in format

  will be converted to their respective characters.
- Conversion specifiers,
  - %s, %b, %d, %x, and %o,

  are replaced by the corresponding argument on the command line.
- Some implementations support other non-standard specifiers.
- When there are more arguments than specifiers, the format string is reused until all the arguments have been consumed

Format specifiers %s, %b, %d, %x, %o

- The %s specifier interprets its argument as a string and prints it literally:

```
$ printf "%s\n" "qwer\ty" 1234+5678
qwer\ty
1234+5678
```

- The %b specifier is like %s, but converts escape sequences in the argument:

```
$ printf "%b\n" "qwer\ty" "asdf\nghj”
qwer    y
asdf
ghj
```

- The %d, %x, and %o specifiers print their arguments as decimal, hexadecimal, and octal numbers, respectively.

```
$ printf "%d %x %o\n" 15 15 15
15 f 17
```


Specifiers : width flags

- conversion specifiers may be
  - preceded by flags for width specification,
  - optionally preceded by a minus sign indicating that the conversion is to be printed flush left, instead of flush right, in the specified output field, aka ‘left justified’

```
$ printf "%7d:\n%7s:\n%-7s:\n" 23 Cord Auburn
23:
   Cord:
Auburn :
```

- In a numeric field, a 0 (zero) before the width flag indicates padding with zeroes:

```
$ printf "%07d\n" 13
0000013
```

Floating point and exponential forms exist in C and BSD

Printf – BSD supports C-like real number format

fF The argument is printed in the style `[-]ddd.ddd'
where the number of d's after the decimal point
is equal to the precision specification for the argument.
If the precision is missing, 6 digits are given;
if the precision is explicitly 0, no digits and no decimal point are printed.
The values infinity and NaN are printed as `inf' and `nan', respectively.

eE The argument is printed in the style e `[-d.ddd+-dd]'
where there is one digit before the decimal point
and the number after is equal to the precision specification for the argument;
when the precision is missing, 6 digits are produced.
The values infinity and NaN are printed as `inf' and `nan', respectively.

gG The argument is printed in style f (F) or in style e (E)
whichever gives full precision in minimum space.

Shell Variables

- To store values in a shell variable, write the name of the variable followed by an = followed by the value
  - count=1
  - my_dir=$HOME/choose_dir

ASSIGN HAS NO SPACES ON EITHER SIDE HERE!

- Spaces are NOT allowed on either side of the =
  - Will result in parsing & code errors!
- Also, the shell has no concept of data types
- No matter what assignment you make, the shell considers the value as a string of characters
  - (arithmetic needs conversion functions...see later!)
- Variables don't need to be declared, they're simply assigned values when you want to use then

Referring to Variables - $$$'s for value

- to get what you pay for, first need $ up front!
- In order to refer to the value of a variable, preface the variable name with a $
  - echo $count - displays the value of count
    - echo count – literally displays the word: count
  - echo $HOME (and environment variable storing users home!)
    – displays value of variable HOME :- /users/your_id
  - echo HOME – literally displays the string HOME
- To ensure a variable is not accidentally modified, specify it as *readonly*. Further attempts to modify it will result in an error from the shell.
  - my_dir=$HOME/choose_dir
  - readonly my_dir

File Name Substitution & Variables

- If you define a variable as x=*,
  - ls $x will produce a directory list, and currently recursive at that
- How does it really work...what did the shell store in x ? Was it * or the list of files in your current directory?
  (kind of silly question, if you know about lang. translation!)
- The $x ensures value of x, (which is '*') is used, not just 'x'
- The shell stored the * in x, the shell does not perform filename substitution when assigning values to variables
- What actually happens is:
  - The shell scans ls $x, substituting * for $x
  - It then rescans the line and substitutes all the files in the current directory for the *
    - (And must repeat for recursive listing)

Quoting – necessary in shell scripting – but 4 types

- BACKTICKS:- when not overridden/ignored by single quotes
  - `command` executes command and inserts standard output at that point
- SINGLE QUOTES:- '…'
  - removes special meaning of all enclosed characters, including backticks.
- DOUBLE QUOTES "…"
  - removes the special meaning of all enclosed characters EXCEPT $, `, and \
- EXCEPT BACKSLASH STEPS BACK..
  - \c removes the special meaning of the character that follows the \;
    - inside double quotes it removes the special meaning of $, `, ", NEWLINE, and \
    - but is otherwise not interpreted…e.g. within single quotes.

- So filelist="`ls`", issues the ls cmd, on:-
- Whereas filelist='`ls`', assigns the string, incl. backticks `ls` to filelist
- And filelist="\`ls\`" should also behave exactly like previous line.
- So many options exist for confusion & accidental issue or not of commands.
- Probably best to verify outcomes with $echo $filelist

Check quote is not a back-tick – hard to debug a tick on your back

Daft memory aid: Back ticks : can bite you(r) back,
even with 2 (clearly poor coats) but not one good coat,

- echo Your current directory is `pwd`
  - (NB ` - backtick (NOT a single quote ‘ ’ !!!)
  - Outputs “Your current directory is /users/cs_year_group/ur_id_abc123"
  - Outputs, literally after :
    Your current directory is /users/cs_year_group/ur_id_abc123
- echo There are `who | wc -l` users logged on
  - Outputs "There are 13 users logged on"
- Since single quotes override everything, the following output should make sense:
  - echo '`who | wc -l` tells how many users are logged on'
  - Outputs "who | wc -l` tells how many users are logged on"
- But back ticks are interpreted inside “double quotes”
  - echo "You have `ls | wc -l` files in your directory"
  - Outputs "You have 24 files in your directory" (Why? “ don’t affect `)
  - Backticks are left of ‘z’ on Mac, left of ‘1’ on other keyboards

Variables in the Environment

- This course not sufficiently advanced to use environments much, but fyi…
- When variables are assigned, they are local to the current shell
- Since scripts are executed in a sub-shell,
  (a new process, with a different context, like a function) these *local variables* aren't visible to the script
- To make them visible (inherited by) subsequent sub-shells, they must be *exported* (like passing values)
  - e.g. $export x
- The *env* command lists all currently defined exported variables, and will show value for x as a2c
- (exec has workarounds + complications – avoid for now)

Environment

- Create a file called vartest.scr that contains:

  echo :$x:
- Save it and make it executable (chmod 744 vartest.scr)
- Now, assign a2c to x

  x=a2c
- Then execute vartest.scr – what happens?

```
cs1> cat vartest.scr
echo :$x:
cs1> x=a2c
cs1> bash vartest.scr
::
cs1>
```

It prints the colons,
But not value of x variable
Has not been passed to the bash(ed) subshell process
Which executes the script
But by issuing a cmd : $export x
Before bash rectifies this
with :a2c: being printed out.

Extract for env

```
Cs1>x=a2c
Cs1>export x
cs1> env
TERM=xterm
SHELL=/bin/bash
XDG_SESSION_COOKIE=ec93...  ... ...166
SSH_CLIENT=143.239.74.157 50944 22
SSH_TTY=/dev/pts/0
LC_ALL=en_GB.UTF-8
USER=jsad1
MAIL=/usr/spool/mail/jsad1
PATH=/users/2019/jsad1/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/games:/
users/software/utils
PWD=/users/2019/jsad1
LANG=en_GB.UTF-8
SHLVL=1
HOME=/users/2019/jsad1
LANGUAGE=en_GB:en
LOGNAME=jsad1
SSH_CONNECTION=143.239.74.157 50944 143.239.75.218 22
x=a2c
_=/usr/bin/env
cs1>
```

X variable value is listed after export, before env cmd

Locked terminal/process - breakout

More later – just to help explain next slide for jambusting

- If a command/script will run for a long time,
  - it will lock up the terminal and not give a prompt
    - So you can't do anything but wait ... waste time…and grow old...
    - or you could snooze it Ctrl+z (repeat to resume);
    - Or cut it Ctrl+c (c for cut)
- But if still locked (in an unresponsive state) and can't get a command prompt then
  - Open a new terminal (so can work within that)
    - Current terminal (The + tab icon at the top of the terminal window)
    - Or tmux to multiplex one, especially if remote ssh
  - then kill the process kill -9 PID find the PID from ps, jobs etc
  - PID's are unique, whereas terminal processes may have similar names, but can pgrep and pkill processes by name… nearby slides.

Foreground – background – gaining ground!

- If a command/script will run for a long time,
  - run the command in the background, within current terminal...
    - Just follow the command with &
      - e.g. sleep 5& … will sleep a process 5 secs
    - it will run away in the background and finish when it's done..
      - With no message unless it fails or is programmed to
      - Until finished when it responds with its PID
  - But better still it will restore a command line input prompt
    - So you can carry on interactively in the meantime
  - can always resurrect it with a foreground cmd : fg PID
  - Or restart in background, if snoozed (^z) using the background cmd : bg PID … just as if it was started with & appended
  - PID's with: jobs or ps

Clearing processes – by name or number

- Haven't time to do processes right now in this course
  - Single thread programs are enough for now, comms. More confusion
  - Processes spawn others, a process tree,
    - similar to filesystem communicating via signals,
  - Here : just the basics to find and kill a process
  - Check manual/online/books for more info
- By process name – or partial name - pgrep
  - eg pgrep -l chrome ... will list PID & name of proesses with chrome as part of the name
  - Can kill by name with
    - - pkill name... but (partial) names are not unique PIDs are.
    - - killall ... will kill all with (partial) match
- By number
  - Easiest is to use ps -au for limited
  - Then kill -9 PID .... the -9 (signal) makes sure it is finishes..
- NB if you shut a terminal window, it stops all processes, even those in the background unless a disown command is used to disconnect stdio.
- Enough for this course!

Main Pre-defined Variables – some implementation dependent

- $# - number of arguments on command line
- $0 - name of program being executed – so user remembers
- $1, $2, etc … first and second arguments
- $* - collectively references all of the positional parameters as a single string.. ($1, $2, $3,…)
- $$ - PID of current process
- $? - exit status of last command NOT executed in background
- $! - PID of last command executed in background
- "$@" - just like $* except it returns arguments as though individually quoted, as separate strings ie; "$1 "$2" "$3"…"$9" even with null IFS

However, since echo output is non-standard except for basic strings, the values output for $@ and $* may be unreliable.

- It is best to use printf especially when IFS is modified in any way.
- NB: "$@" Quotes are necessary, without them, $@ acts just like $*,

- more in about 8 slides...

Passing Arguments – often a wise move!?

- When the shell invokes your script, it assigns the command line arguments to a set of variables
  - $0, $1, $2,…$9
  - $0 is the script name
  - $1 is the first argument, $2 the second, up through $9
  - You can then refer to these variables in your script
- If more than 9 arguments are used, you must access them using the *shift* command
  - *shift* simply does a left shift on all the arguments, discarding $1 and making $1 = $2, $2 = $3, etc
  - Note, this means that if you still need $1 you must save it in another variable BEFORE performing the shift

first_script.sh (current bash supports echo –e)

```
#! /bin/sh
greeting="Hi!"
echo -e " \n\n"
echo " ******** ************ ************ ************ ******** ********"
echo " ********      $greeting - We're now up and running!?     ********"
echo " ******** ************ ************ ************ ******** ********"
echo -e " \n\n"

echo -e "The script name is:- \t\t $0"
echo -e "The parameter list is:- \t $@"

echo -e "\nParameters :-"
echo -e "First:-  \t $1"
echo -e "Second:- \t $2"
echo -e "Third:-  \t $3"

echo -e "\n\n\nHere are some environment variables"
echo -e "My userid      \t $ME"
echo -e "HOME directory         \t $HOME"
echo -e "PATH           \n$PATH \n\n"
exit 0
# Excersise:- extend to display any (reasonable) no. of parameters
# Hint: $1-$9 stores 9 args ($0 being the script name)
# For more than 9 args, use shift, which downshifts args one index.
# so that $1 stores $2, $2 stores $3 etc with previous values overwritten.
# NB $1 is lost, so store locally (and repeatedly) if needed again.
```


bash first_script.sh a b c d

```
******** ************ ************ ************ ******** ********
********     Hi! - We're now up and running!?     ********
******** ************ ************ ************ ******** ********
The script name is:-  first_script.sh
The parameter list is:-       a b c d

Parameters :-
First:-       a
Second:-      b
Third:-       c

Here are some environment variables
HOME directory      /users/csdipact2012/sajd
PATH
/users/guest/sajd/bin:/users/coursework/cs1100/bin:/users/software/utils:/
usr/local/bin:/usr/bin:/usr/X11R6/bin:/bin:/opt/gnome2/bin:/opt/gnome/bin:/
opt/kde3/bin:/usr/lib/java/bin:
```


One way of processing files in a script : step_file.sh
(note: no format options used, so echo is ok!)

```
#!/bin/bash
filelist="`ls`"
for file in $filelist
do
        echo $file
done
exit 0
```

```
#!/bin/bash
for arg in *
do
        echo $arg
done
exit 0
```

```
cs1> bash step_file.sh
first_script.sh
loopy_args.sh
more_loopy_args.sh
step_file.sh
textap
```

Note special directory listing convenience

For arg in *

Some shell implementations support such code taking an argument list from piped output of a previous command.

The IFS – Internal Field Separator…

- Internal Field Separator special variable, which the system uses to separate fields i.e. “$*” is equivalent to "$1c$2c...", where c is the first character of the value of the IFS variable.
- The IFS can save lots of programming to separate and parse fields…
  - Eg if you needed to separate the directories listed in the internal PATH variable…which are separated by ‘:’
    - set the IFS equal to : & system will separate them automatically; just remember to unset after.
  - Or
    - use sed to replace : in the string to be parsed e.g. PATH with the default system IFS
  - Or
    - write a parse routine in some language.

Resetting IFS for parsing convenience

IFS being set to colon : the field separator in PATH variable.

```
#!/bin/bash

echo -e  "\n\nPATH in it's internal form using ':' as separators"
echo $PATH

# could be done using sed with s/:/some other separator/
# but resetting IFS is handier and more general

IFS=":" # setting the IFS to separators in PATH
echo -e "\n\nPATH parsed and separated by system with IFS set to :"
for file in $PATH
do
        echo $file
done
echo -e " ********** "
unset IFS       # resetting IFS back to default value

exit 0
```


PATH Output with IFS=”:”

```
PATH (has changed since!) in it's internal form using ':' as separators
 /users/csdipact2013/jsad1/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/
bin:/sbin:/bin:/usr/games

PATH parsed and separated by system with IFS set to :
/users/csdipact2013/jsad1/bin
/usr/local/sbin
/usr/local/bin
/usr/sbin
/usr/bin
/sbin
/bin
/usr/games
 **********
```


$* and $@ - skip a few!

With no positional parameters, neither is expanded, so are null & removed;
Without double quotes, both expand to positional parameters, starting from $1.
Otherwise

For $*
- within double quotes (**“$*”** - when $* is enclosed in double quotes):-
  - Expands to a single string;
    - with each parameter separated by the first character of the IFS.
  - If IFS is unset to default values, the parameters are separated by spaces.
  - If IFS is null, the parameters are joined without intervening separators.

For $@
- within double quotes (**“$@”** when $@ is enclosed in double quotes):-
  - Expands each parameter to a separate word
    - i.e. “$@” is equivalent to "$1”, “$2”, ...",
  - If the double-quoted expansion occurs within a word
  - e.g. if “$@”=”esting pro”, then “inter”$@”jection” = “interesting projection”
    - The first parameter is joined to the first part of the original word
    - The final parameter is joined to the last part of the original word

"$@" Quotes are necessary, without them, $@ acts just like $*

```
#!/bin/bash
echo Number of Aguments passed is $#
for arg in $*
do
        echo $arg
done
exit 0
```

```
#!/bin/bash
echo Number of Aguments passed is $#
for arg in $@
do
        echo $arg
done
exit 0
```

Difference:- ‘a b’ treated as a single arg by $*, but not by unquoted $@

```
$ bash args.sh a b c
Number of Aguments passed is 3
a
b
c
```

```
$ bash args.sh ‘a b’ c
Number of Aguments passed is 2
a
b
c
```


"$@" Quotes are necessary, without them, $@ acts just like $*

```
#!/bin/bash
echo Number of Aguments passed is $#
for arg in "$@"
do
        echo $arg
done
exit 0
```

No Difference:- ‘a b’ treated as a single arg by $*, and by ~~un~~quoted “$@”

```
$ bash args.sh a b c
Number of Aguments passed is 3
a
b
c
```

```
$ bash args2.sh 'a b' c
Number of Aguments passed is 2
a b
c
```


args_aat_star_script

```
#!/bin/bash
echo -e '\n*****************************'
set one two tri
echo parameters set to one two tri
# token delimiter: IFS = Internal Field Separator
IFS=''  # a blank empty IFS
echo -e '\n\t IFS set to blank\n' #  echo -e is partly supported
echo -ne 'displayed using $@ \t'
echo "$@"
echo -ne 'displayed using $* \t'
echo "$*"
unset IFS # IFS unset = reset back to normal
echo -e ' \n\t normal IFS \n' # but space needed for \n
echo -ne 'displayed using $@ \t'
echo "$@"
echo -ne 'displayed using $* \t'
echo "$*"
echo -e '\n*** S@ is preferable & independent of IFS ***'
exit 0
```


bash args_aat_star_script

```
cs1> bash aat_star_args

*****************************
parameters set to one two tri

        IFS set to blank

displayed using $@      one two tri
displayed using $*      onetwotri

        normal IFS

displayed using $@      one two tri
displayed using $*      one two tri

*** S@ is preferable & independent of IFS ***
cs1>
```

Set – values to flags and positional variables

- unset, restores IFS to default values from assignment IFS='whatever'
- **set is not the opposite of unset, but used to set variables, flags etc**
- set merely assigns values to positional variables, since there is no way to directly assign values to the positional variables the obvious approach is logically ridiculous
  - e.g. to set the first positional variable to the 100th, the following is tempting:- 1=$100
    but omitting the $ on lhs of assigment, renders it ridiculous 1=1 !
  - or likewise 3=filename doesn't work
- set gets around this by indirectly assigning values to positional variables ($1, $2,..) by the sequence following set… unless nam
- "set alpha bravo charlie" (sonic alphabet for clear letter comms) gives :-
  - echo $1 $2 $3
  - alpha bravo charlie
- When using set, $# and $* also work properly

Debugging Scripts with -n -x

Two debugging aids are available within the shell: (also at CLI - cmd line)

- -n for syntax checking before running
  - helps avoid trying to run sytactic (not semantic) nonsense!
- -v for verbose..
  - Prints commands before executing them
    - So you know the kind of nonsense you're trying to run
- -x for eXecute|terminate bugs ;-) trace of executing statements
  - shows stepwise expansion (&substitution) of commands before executing them, otherwise similar to –v option
  - So your screen is absolutely full of nonsense!
- set can be used to turn these debugging flags on and off, effectively setting breakpoints as in a normal debugger, whereby detailed debugging can be restricted to problem area

Using –x trace debug option > bash -x

```
#!/bin/bash
filelist="`ls`"
for file in $filelist
do
        echo $file
done
exit 0
```

```
cs1> bash -x step_file.sh
++ ls
+ filelist='first_script.sh
loopy_args.sh
more_loopy_args.sh
step_file.sh
textap'
```

```
+ for file in '$filelist'
+ echo first_script.sh
first_script.sh
+ for file in '$filelist'
+ echo loopy_args.sh
loopy_args.sh
… …  and so on  … …
+ for file in '$filelist'
+ echo textap
textap
+ for file in '$filelist'
+ echo textap~
textap~
+ exit 0
```


Set – for debugging and breakpoint flags

Debugging and breakpoints - for tests confined to a code segment

- Assuming code has been
  - Either designed & developed properly and carefully, ;-)
  - Or at least compiled from interactively che/acked statements,

- then starting with high level trace (or echo!) to check logic flow…!
(whether indicated through trace info or developing new code extensions, but new code may only unearth a hidden bug in old code, presumed right)

- **set –v –x** for verbose or expanded mode debugging
- **set +v +x** to turn it off again, and restrict debugging to a smaller section.
- **Set** won't run under root, but shouldn't be debugging scripts as root!
  - Set has been known to run under sudo with root privs. & risks

Or the whole shebang… modify the shebang line when developing code

» **#!/bin/bash –x**

Handy when developing new scripts, avoids always –x in command line, But need to comment or edit it out later…

Set – handy for setting debug flags

Safe escape on error rather than compound it…avoid danger!

set –e …

- should possibly be the second line in all scripts after #!

- it will cause script to exit on any shell command exiting with an error (a non-zero return code)
  - hopefully minimising collateral damage!?
- Except for commands used in tests for flow control statements
    - (i.e. loops (for, while, until, etc) , if, case :
  - Which use non-zero code for control-flow test/decision
  - Here if the flow-control test fails, set -e will abort
  - Need another approach
    - Either clear before & reset immediate after flow test statement
    - Or use another approach… echo, printf or debug (-x)

Set –e Hazard for root privileges!

If a script is run with root privileges, even using sudo, and fails with –e set, it will exit to the user giving root access!
- a hazard for all: system, root & user not knowing!?

(sudo is a way of allowing a user to run a script with root privileges without giving him root access or the root password)

So make sure any such script is correct, and will not fail on any input scenario, when run at any stage with –e privileges.

If admin then be careful about who has sudo privileges, that they are trustworthy
- both ethically – that they won't deliberately do something nasty
- and technically – that they won't accidentally do something nasty

Rampant delusion : my program is right, the computer is wrong!

Input Interactive & Line Input from file

- Data is obtained from the user by using *read*
  - Syntax: *read variables*
  - Example:
    - *read x y*
    - *read text*
  - read gets input from STDIN so it can be redirected
    - *read text < data*
- This doesn't work in all version of sh so an alternative is the *line* command
- line can read/write an entire line from/to STDIN/STDOUT
  - text=`line < data`
  - Check out more on file descriptors if needed.
- Don't confuse with the readline library – basically bash cmd line,
  - Complex; for lots more facilities and info… see the manual.

A List of commands…

- is a sequence of
  - One or more commands
  - Separated by
    - Newlines … the usual;
    - Semicolons … several commands on a line;
    - Ampersands – run in background
    - Control operators - &&, || etc conjunctions

    (will be discussed at length in following section on tests)
- May be used as
  - The body of any loop
  - The condition in a while or until loop
- Has the exit code of the last command in the list.