Shells & Shell Programming (Part A)

Software Tools

EECS2031 Winter 2018

Manos Papagelis

Thanks to Karen Reid and Alan J Rosenthal for material in these slides

SHELLS

What is a Shell

- A shell is a command line interpreter that is the interface between the user and the OS.
- The shell:
  - analyzes each command
  - determines what actions are to be performed
  - performs the actions
- Example:

```
wc -l file1 > file2
```

Which shell?

- sh – Bourne shell
  - Most common, other shells are a superset
  - Good for programming
- csh or tcsh – command-line default on EECS labs
  - C-like syntax
  - Best for interactive use.
- bash – default on Linux (Bourne again shell)
  - Based on sh, with some csh features.
- korn – written by David Korn
  - Based on sh – Some claim best for programming.
  - Commercial product.

`bash` versus `sh`

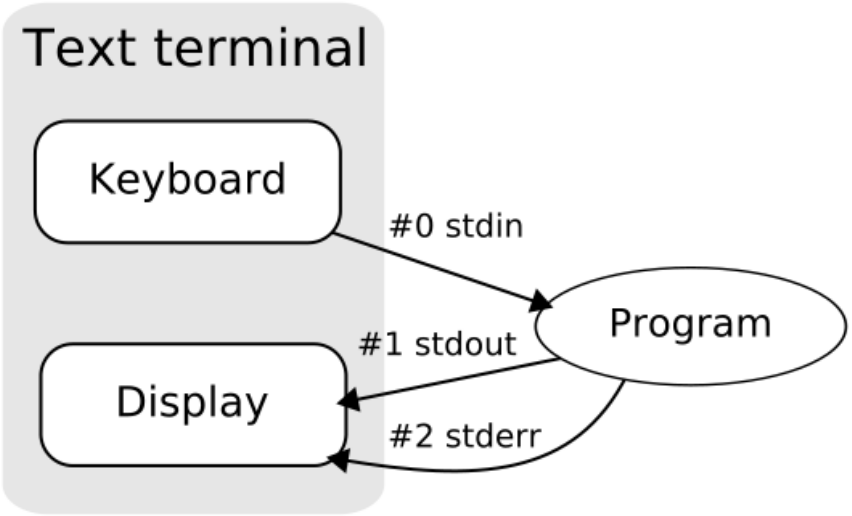
- On *EECS labs*, when you run `sh`, you are actually running `bash`.
- `bash` is a superset of `sh`.
- For EECS2031, you will be learning only the features of the language that belong to `sh`.

Changing your shell

- I recommend changing your working shell on EECS to bash
  - It will make it easier to test your shell programs.
  - You will only need to learn one set of syntax.
- What to do:
  - echo $SHELL (to check your current shell)
  - `chsh <userid> bash`
  - Logout and log back in.
  - `.profile` is executed every time you log in, so put your environment variables there

Standard Streams

- Preconnected input and output channels between a computer program and its environment. There are 3 I/O connections:
  - standard input (stdin)
  - standard output (stdout)
  - standard error (stderr)



Common shell facilities

- Input-output redirection

```
prog < infile > outfile
ls >& outfile        # csh and bash stdout and stderr
ls > outfile 2>&1        # sh stdout and stderr
```

  - More redirection examples:

    https://www.tutorialspoint.com/unix/unix-io-redirections.htm

    https://en.wikipedia.org/wiki/Redirection_(computing)

- Pipelining commands
  - send the output of a command to the input of another

```
ls -l | wc
ps –aux | grep papaggel | sort
```

Job Control

- A job is a program whose execution has been initiated by the user
- At any moment, a job can be **running** or **suspended**
- Foreground job:
  - a program which has control of the terminal
- Background job:
  - runs concurrently with the parent shell and does not take control of the keyboard
- Start a job in the background by appending `&`
- Commands: `^Z, jobs, fg, bg, kill`
- More information:
  - https://linuxconfig.org/understanding-foreground-and-background-linux-processes

File Name Expansion

```
ls *.c
rm file[1-6].?
cd ~/bin
ls ~papaggel
ls *.[^oa]   – ^ in csh, ! in sh
```

- **`*`** stands in for **0 or more** characters
- **`?`** stands in for **exactly one** character
- **`[`**`1-6`**`]`** stands in for **one of** 1, 2, 3, 4, 5, 6
- `[`**`^`**`oa]` stands in for any char **except** o or a
- **`~`** stands in for your home directory
- **`~papaggel`** stands in for my home directory

SHELL PROGRAMMING

Shell Programming (Bourne shell)

- Commands run from a file in a **subshell**
- A great way to automate a repeated sequence of commands.
- File starts with `#!/bin/sh`
  - absolute path to the shell program
  - not the same on every machine
  - for bash it is `#!/bin/bash`
- Can also write programs interactively by starting a new shell at the command line.
  - Tip: this is a good way to test your shell programs

Example: at the command line

```
% sh
sh-3.2$ echo "Hello World"
Hello World
sh-3.2$ exit
Exit
%
```

Example: in a file

- In a file named “hello_world.sh” write:

```
#!/bin/sh
echo “Hello World!”
```

- make the file executable:

```
chmod 711 hello_world.sh
```

- run the script:

```
./hello_world.sh
```

Shell scripts

Like any programming language:

- Variables
- control structures (if, for, while, …)
- Parameters
- subroutines (functions)
- Plus shell conveniences (I/O redirection, pipes, built-in commands)

Shell scripts advantages

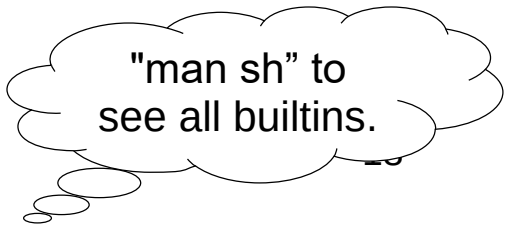
- saves typing if you need to perform the same thing over and over
- faster
- you can make it quite complex and debug it before using

Oh! - you mean just like a program! :-)

Commands

- You can run any program in a shell script by calling it as you would on the command line
- When you run a program like `grep` or `ls` in a shell script, a new process is created
- There are also some built-in commands where no new process is created

  - `echo`
  - `set`
  - `read`
  - `exit`
  - `test`
  - `shift`
  - `wait`



Variables

- local variables – spaces matter
  - `name=value` – assignment
  - `$name` – replaced by value of name
  - variables can have a single value or list of values.
- Single value:

```
bindir="/usr/bin"
```

- List of values (separated by spaces):

```
searchdirs="~/tests $HOME/test2 ."
```

Example:

($ or % is the default sh prompt)

```
$ bindir="/usr/bin"
$ searchdirs="~/tests $HOME/test2 ."
$ echo $searchdirs
~/tests /u/reid/test2 .
$ echo $bindir
/usr/bin
```

STRING REPLACEMENT & QUOTING

String Replacement

- Scripting languages are all about replacing text or strings (unlike other languages such as C or Java which are all about data structures)
- Variables are placeholders where we will substitute the value of the variable
- Example:

```
iters="1 2 3 4"
for i in $iters; do
   echo $i
done
```

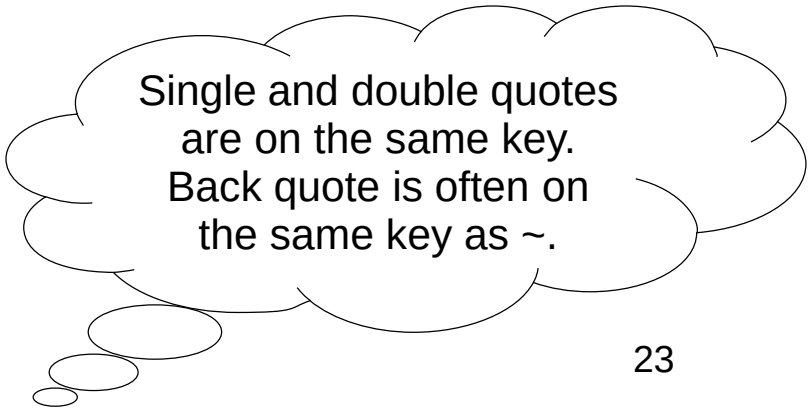
=

```
for i in 1 2 3 4; do
   echo $i
done
```

Quoting

- Double quotes ("") prevent wildcard replacement only.
- Single quotes (‘’) prevent wildcard replacement, variable substitution and command substitution.
- Back quotes (``) cause command substitution.

Practice and pay attention.



Double quotes (" ")

Use *double quotes: " " to prevent wildcard interpretation of *,?, etc.*

$ ***ls***

a b c cap.sh whale.sh

$ echo *

a b c cap.sh whale.sh

$ echo "*"

*

$ echo ?

a b c

$ echo "?”

?

Single quotes (' ')

Use *single quotes: ' ' to prevent pretty much everything:*

- *wildcards*
- *variable value substitution*
- *command substitution (see backquotes, a few slides down)*

$ ***echo * $shell***

a b c cap.sh whale.sh /bin/tcsh

$ echo '* $shell'

** $shell*

$ echo `whoami`

papange1

$ echo 'whoami`'

`whoami`

Backquotes (` `)

- backquote ` - usually on the same key as ~: the leftmost on the top row on QWERTY boards
- Known as command substitution - the meaning is take the expression inside the ` `, execute it, and substitute the result for the expression
- Command substitution causes another process to be created

Backquotes (` `)

$ whoami

papaggel

$ grep `whoami` /etc/passwd

papaggel:x:18084:2000:Manos Papagelis:/cs/home/papaggel:/cs/local/bin/bash

$ grep papaggel /etc/passwd

papaggel:x:18084:2000:Manos Papagelis:/cs/home/papaggel:/cs/local/bin/bash

Quoting example

" – double quotes
’ – single quote
` – back quote

```
$ echo Today is date
Today is date
$ echo Today is `date`
Today is Thu Sep 19 12:28:55 EST 2002
$ echo "Today is `date`"
Today is Thu Sep 19 12:28:55 EST 2002
$ echo ’Today is `date`’
Today is `date`
```

Another Quoting Example

- What do the following statements produce if the current directory contains the following non-executable files?

```
Assume there exist files: a b c
$ echo *
$ echo ls *
$ echo `ls *`
$ echo "ls *"
$ echo 'ls *'
$ echo `*`
```

```
" – double quotes
' – single quote
` – back quote
```

Answers

$ a b c

$ ls a b c

$ a b c

$ ls *

$ ls *

$ will try to run a (depends on permissions, etc.)

Quoting Summary

- "" double quotes:

  prevents wildcards in file names (*, ?) only
- ' ' single quotes:

  prevents all replacements: wildcard, variable substitution, command substitution
- ` ` backquotes:

  simply causes command substitution, does not override anything