Introduction to shell scripting

- Filename expansion
- Quoting and backslash escapes
- Shell/environment variables
- Commands
- Variables
- Comparisons
- Operators
- Iterations

filename expansion

- "shorthand" for referencing multiple **existing** files on a command line

*	any number of characters
?	exactly one of any character
[abc]	any character in the set [abc]
[!abc]	any character **not** in the set [abc]

- these can be combined together as seen on the next slide

filename expansion (2)

- examples:
  - count lines in all files having the suffix ".c"

    % wc -l *.c
  - list detailed information about all files with a single character file extension

    % ls -l *.?
  - send to the printer all files with names beginning in Chap* and chap* files

    % lpr [Cc]hap*

filename expansion (3)

- * matches any sequence of characters (except those with an initial period)

```
% rm *.o                    # remove all files ending in '.o'
% rm *                      # remove all files in directory
% rm ../*-old*.c
```

- ? matches any single character (except an initial period)

```
% rm test.?     # remove test.c and test.o (etc.)
```

- So to delete a file of the form ".filename" you can't use wildcards

```
% rm .viminfo
```

How do we delete a file named *?

quoting

- controls bash's interpretation of certain characters
- what if you wanted to pass **'>'** as an argument to a command?
- **strong quotes** – All characters inside a pair of single quotes (') are preserved.
- **weak quotes** – Some characters ($,`) inside a pair of double quotes (") are expanded (interpreted) by the shell.
- **backquotes** – substitute result of evaluation as a command

quoting

```
% echo $TERM *
ansi file1 file2 file3

% echo '$TERM' '*'
$TERM *

% echo "$TERM" "*"
ansi *

% echo `date`
Wed 14 Sep 2022 14:54:10 PDT
```

backslash escaping

- Characters used by **bash** which may need to be escaped:

  ~, `, #, $, &, *, (,), \, [,], {, }, ;, ', ", <, >, /, ?, !

- single characters can be "protected" from expansion by prefixing with a backslash ("\")

  cmd * is the same as typing cmd '*'

- protecting special characters in such a manner is an example of **backslash escaping**

```
% cp ~bob/junk \* # make copy of junk named '*'
% rm '*'          # remove '*' (not "delete all files")
```

- Single quotes around a string turn off the special meanings of most characters

```
% rm 'dead letter'
% cp ~bob/junk '*' # same as up above
```

command substitution

- backquotes (`) are used to substitute the result of evaluating a command into a quoted string or shell variable:

```
% echo "Current date is: `date` "
Current date is: Wed 14 Sep 2022 14:54:10 PDT
% BOOTTIME=`date`
% echo $BOOTTIME
Wed 14 Sep 2022 14:55:29 PDT
```

- standards-compliant (i.e. POSIX) style avoids backticks:

```
% echo "Current date is: $(date)"
Current date is: Wed 14 Sep 2022 14:56:24 PDT
```

shell variables

- a running shell carries with it a **dictionary** of variables with values
- some are **built in** and some are **user defined**
- used to customize the shell
- use env to display the values of your **environment variables**
- use set to display the values of your **environment + shell variables**

```
% env
PWD=/home/bgates
GS_FONTPATH=/usr/local/fonts/type1
XAUTHORITY=/home/dtrump/.Xauthority
TERM=ansi
HOSTNAME=c70
. . .
```

shell variables (2)

- many variables are automatically assigned values at login time
- variables may be re-assigned values at the shell prompt
- new variables may be added, and variables can be discarded
- assigning or creating a variable – and notice absence of spaces around the "=" symbol:

```
% somevar="value"
```

- to delete a variable:

```
% unset somevar
```

- To use the value of a shell variable use the `$` prefix:

```
% echo $somevar
```

PATH shell variable

- helps the shell find the commands you want to execute
- its value is a list of directories separated by ':' symbol
- when we intend to run a program, the directory of its executable should be in the PATH in order to be found quickly
- Example: assume that program cmd is located in directory "`/usr2/bin`"

```
% echo $PATH
PATH=/usr/bin:/usr/sbin:/etc
% cmd
bash: cmd: command not found
% PATH="$PATH:/usr2/bin"
% echo $PATH
PATH=/usr/bin:/usr/sbin:/etc:/usr2/bin
% cmd
(... now runs ...)
```

- the shell searches sequentially in the order directories are listed

environment variables

- some shell variables are exported to every subshell
  - when executing a command, the shell often launches another instance of the shell; this is called a **subshell**

    % (date ; who ; pwd) > logfile
  - the subshell executes as an entirely different process
  - the subshell "inherits" the environment variables of its "parent" (main shell)
- "exporting" shell variables (`var`) to the environment

  % export *var*

  % export *var*=value
- example:

  % export EDITOR=vim

shell scripting

- Why write a shell script?
  - Sometimes it makes sense to wrap a repeated task into a command.
  - Sequences of operations can be placed in a script and executed like a command.
- Not everything is sensible for a script, though
  - For some problems it would make more sense for a full program
  - Instances: resource-intensive tasks, complex applications, mission critical apps, etc.

Some simple scripts

- For what appears below, ensure the file containing the script is executable

hello.sh

```
#!/bin/bash

echo 'Hello, world!'
```

status.sh

```
#!/bin/bash

uptime
users
```

- The very first line is called a "shebang" path
  - What follows the "#!" ("shebang") is the command that interprets everything else that follows.

Echoing command-line arguments

- Obtaining command-line argument is relatively straightforward

status.sh

```
#!/bin/bash

echo "First command-line arg" $1
echo "Second command-line arg" $2
```

- Notice that echo takes multiple strings
  - But these are not separated by commas
  - (Shell syntax is close enough to regular programming syntax to be confusing.)

Selection

- The numeric representation of true and false are inverted
  - True == 0
  - False == anything else
- Common tests involve file operators
- Note the spacing in the test expression!

```
#!/bin/bash

if [ -e /home/zastre/seng265/assign1 ] ; then
    echo 'Hooray! The file exists!'
else
    echo 'Boo! The file ain't yet there…'
fi

# Other tests:
# -f True if file is a regular file
# -d True if file is a directory
# -w True if file exists and is writable
# -O True if the account running script owns the
file
```

String and arithmetic relationals

- String operators compare lexical order (i.e., dictionary order)
- Arithmetic operators only work on integers
- Note spacing variants…

```
#!/bin/bash

if [ "abc" != "ABC" ] ; then
    echo 'Case does matter here' ; fi

if [ 12 \< 2 ] ; then
    echo 'Why is 12 less than 2??!' ; fi

a=12
b=2
if [ "$a" -lt "$b" ]
then
    echo 'Something is wrong with numbers here...'
else
    echo "Ah ha! $a can be either a string or integer"
fi
```

More generate tests

- Sometimes we could use a good old logical OR or logical AND
- Some C-like expressive power is possible
- (Note slight syntax variation with the "if" statement)

```
#!/bin/bash

if [[ -e ~/.bashrc && ! -d ~/.bashrc ]]
then
    echo 'Definitely a config file to process'
fi
```

Arithmetic? Not so good…

- Bash usually treats variables as strings
- We can force bash to treat a variable's value as an integer

```
#!/bin/bash

x=1
x=$x+1
echo $x    # Still a string

y=1
(( y=y+1 ))
echo $y    # This gets it...
```

Iteration

- List-like values are common in the shell
  - Arguments passed to command
  - Files expanded by wildcards
- A list literal is simply spelled out

```
#!/bin/bash

for x in 1 2 a
do
    echo $x
done


for x in *
do
    echo $x
done
```

Iteration

- The output from commands may be used to create a list
  - Note that the text from the command will be tokenized into a sequence of individual words…
- We can also write old-style for loops
- Use "set –x" to have the shell print out commands as they are executed
  - Handy for debugging

```
#!/bin/bash

set -x

for i in $(date) ; do
    echo item: $i
done

for (( i=0; i<5; i++ )) ; do
    echo item: $i
    echo in$i.txt
done
```

Iteration

- Also: while loops
  - Recall: quantity in the square brackets is tested
  - Top-tested

```
#!/bin/bash

COUNTER=0
while [ $COUNTER -lt 5 ] ; do
    echo The counter is $COUNTER
    let COUNTER=COUNTER+1
done

COUNTER=8
until [ $COUNTER -lt 2 ] ; do
    echo COUNTER $COUNTER
    let COUNTER-=1
done
```

Lots more to shell scripting…

- … parameter expansion …
- … regular expressions …
- … but we have enough for now.
- Recall: selection and loops control by a test
  - If a program runs to completion without errors, it returns 0;
  - otherwise it should return something other than zero
  - This can be used to phrase a conditional (i.e., for driving a test plan involving sets of inputs and outputs)