

# CSE 374 Lecture 4

Shell Variables and Scripting

*Feel free to ask questions until lecture starts...*

# Any questions before we get started?



Tc 0

Powered by  **Poll Everywhere**

Start the presentation to see live content. For screen share software, share the entire screen. Get help at [pollev.com/app](https://pollev.com/app)

# Today

1. Alias
2. Scripting
3. Source / executable

## Office hours this week:

Monday	Tuesday	Wednesday	Thursday	Friday
09:30-10:20 Lecture CSE2 G10 <i>I/O Redirection and alias</i>	14:30-15:30 OH Evan Gates Center 152	09:30-10:20 Lecture CSE2 G10 <i>Introduction to scripting</i>	11:00-12:00 OH Emma CSE1 5th floor Breakout	09:30-10:20 Lecture CSE2 G10 <i>Scripting Continued</i>
10:30-11:30 OH Alex X CSE1 4th floor Breakout		10:30-11:30 OH Aditya CSE1 2nd floor Breakout	15:00-16:00 OH Alex L TBD	15:00-16:30 OH Adrian and Aurora CSE1 5th floor Breakout
12:30-1:30 OH Ray CSE1 4th floor Breakout				23:59 PRACTICE HW0 due; Shell Access Spec
23:59 PRACTICE HW-Intro				

# HW0 & HW1

Please remember that these two assignments should be done using Seaside.

*If your homework passes the autograder this is sufficient....*

To move files from seaside, the 'scp' command works well. Remember the command is `scp <copyfrom> <copyto>`, and pay attention to which computer you are executing it on.

But also, if you are disciplined about doing your work in your `~/cse374` drive, you can access it with windows file explorer.

# Note about mntclassdir

TL;DR

Use `mntclassdir cse374` every time you log in.

Do all your work in `~/cse374`

Use `udrive` access from your home machine instead of `scp`

# Passwords, and managing Passwords

Linux systems have consistent password management.

- `/etc/passwd` file contains user info
  - Username
  - Password
  - Userid, groupid
  - Shell
  - Home directory
- `/etc/shadow` stores encrypted passwords

Change your password on Linux:

```
> passwd
```

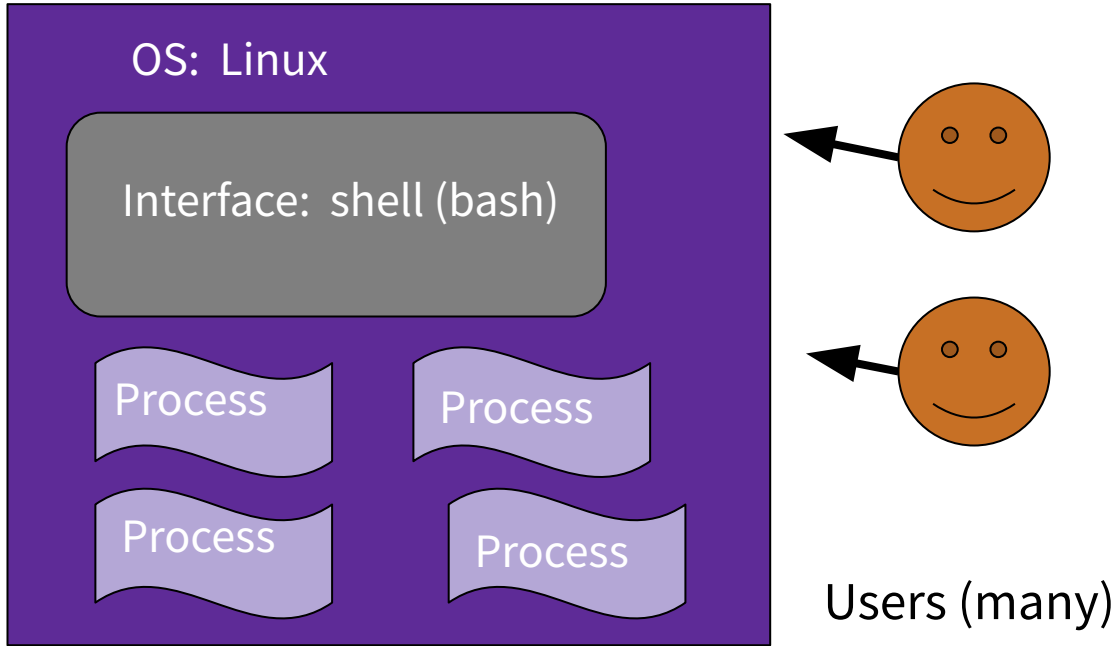
Prompts for previous password, then new password

Passwd also has facilities for those with sudo access to update other user accounts and password management

Seaside is a little different - passwords are obtained from the UWNetID servers (no `/etc/passwd` entries).

Passwd will work, and propagate changes through UWNetID servers.

# Computer Model



- ❖ Computers do two things
  - Store data (filesystem)
  - Manipulate data (processes)
- ❖ Shell is a process that allows the user to interact with the above.
- ❖ But, the shell also allows programming in its special language.

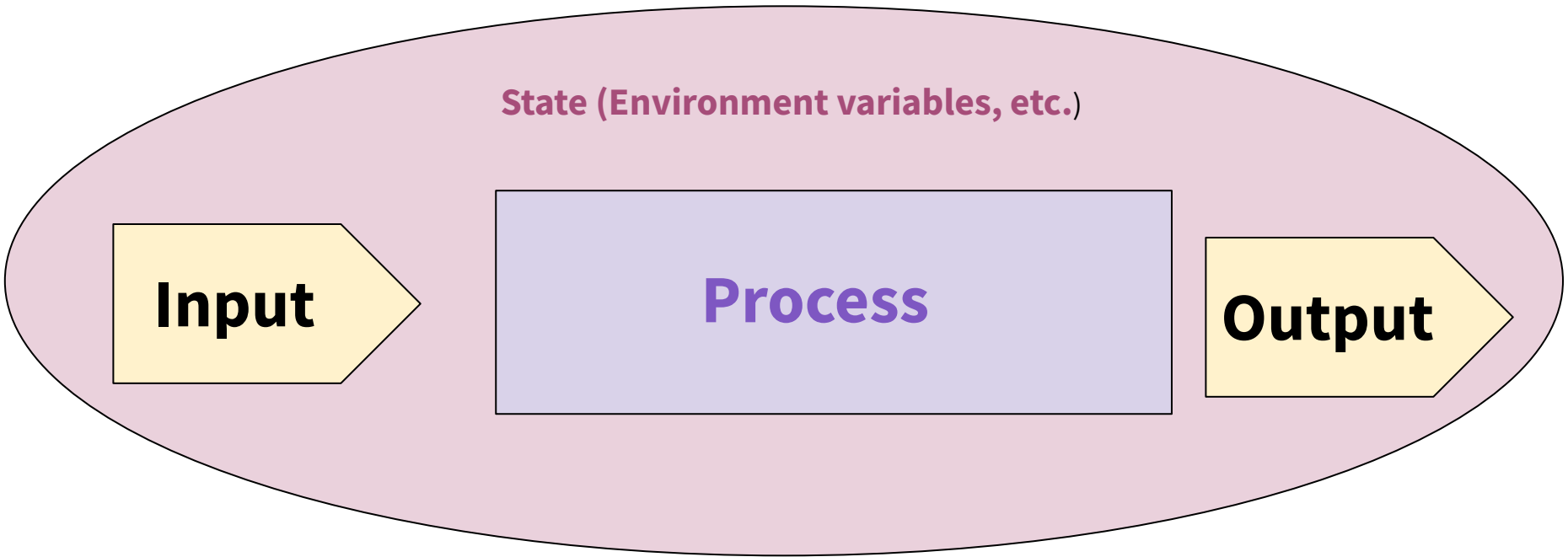
# Bash (shell) Language

- Bash acts as a language interpreter
  - Commands are subroutines with arguments
  - Bash interprets the arguments & calls subroutine
  - Bash also has its own variables and logic



# Bash (shell) Language

- Bash acts as a language interpreter
  - Commands are subroutines with arguments
  - Bash interprets the arguments & calls subroutine
  - Bash also has its own variables and logic



*BASH applies its own processing to the I/O text - 'globbing'*

# Special Characters

- Directory Shortcuts
  - ~uname or ~
  - ./ or ../
- Wildcards - *Globbering*
  - 0 or more chars: \*
  - Exactly 1 char: ?
  - Specified chars: [a-f]

## History, or '!'

```
[mh75@cancun cse374]$ cat dog > dig
[mh75@cancun cse374]$ echo shovel >> dig
[mh75@cancun cse374]$ cat dig > dug
[mh75@cancun cse374]$ cat dig > digger
[mh75@cancun cse374]$ ls
cat  dig  digger  dinosaur  dog  dug
[mh75@cancun cse374]$ ls c*
cat
[mh75@cancun cse374]$ ls d*
dig  digger  dinosaur  dog  dug
[mh75@cancun cse374]$ ls d[ou]*
dog  dug
[mh75@cancun cse374]$ ls d??
dig  dog  dug
[mh75@cancun cse374]$ !ls
ls d??
dig  dog  dug
```

# Special Characters

! > < & | \* ~ [] “ ‘ ` \$ /

\ is escape  
character



“string”



‘string’

What do they all  
mean?

Would substitute  
things like \$VAR

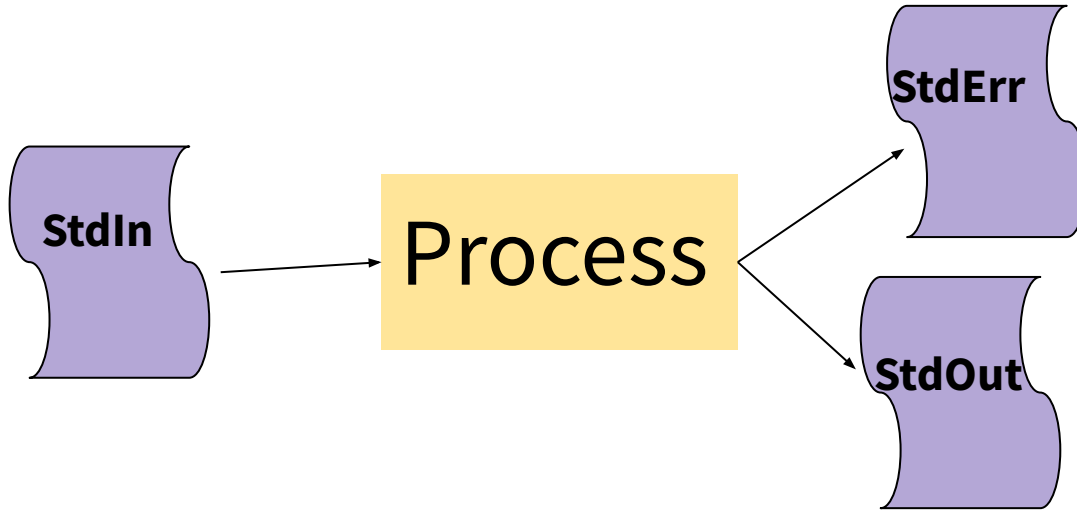
Suppresses  
substitutions

# Shell Behavior

All redirection & string expansion or substitutions are done by the shell, before the command.

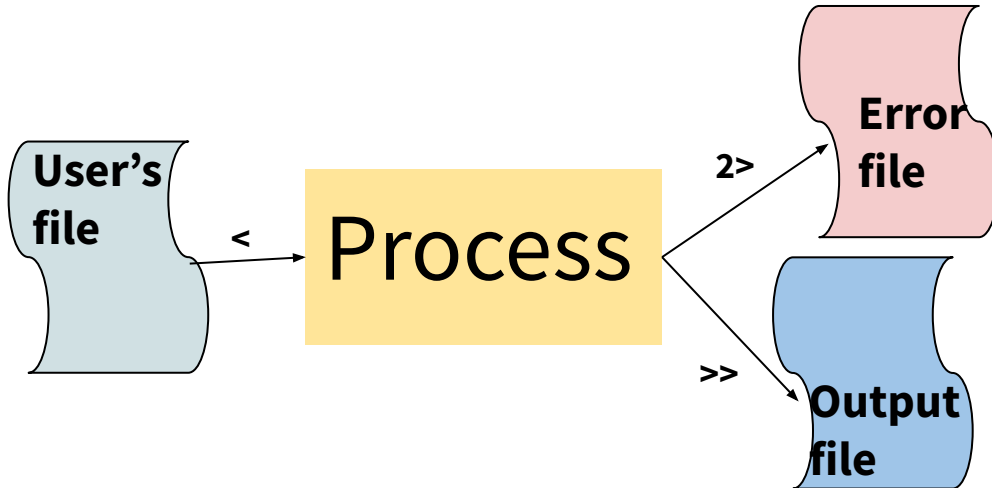
Command only sees resulting I/O streams.

Processes all can take INPUT from one source, the default being StdIn.



Processes have two OUTPUT destinations, the default being StdOut and StdErr. You can think of these as two potential files to which a processes can write.

But, instead of using StdIn you can use any file, and 'redirect' it in by using the '<' symbol (pointing towards process).



You can also write to different files instead of StdErr or StdOut. The '>' symbol means to put in an new file, while '>>' means to append to the end of a file. The '2' specifies that you want iostream '2', or the error stream.

# I/O Streams

- All bash commands have three streams
  - 0- stdin [keyboard]
  - 1- stdout [screen]
  - 2-stderr [screen]
- Can redirect streams
  - < yourInput
  - > yourOutput
  - >> appendYourOutput
  - 2> yourError
  - &> yourOutput&Error
  - And more...
- Special File /dev/null
  - Is EOF if input
  - Data is discarded if output
- Can combine one cmd to the next
  - Cmd1 | cmd2 - pipe output of cmd1 into input of cmd2
  - Cmd1; cmd2 - do one after another
  - Cmd1 `cmd2` - use output of cmd2 as argument to cmd1
- Can use cmd logic
  - Cmd1 || cmd2 - do cmd2 if cmd1 fails
  - Cmd1 && cmd2 - do cmd 2 if cmd1 succeeds

## Some Bash redirection syntax

redirect stdout to a file →	<code>command &gt; output</code>
redirect stderr to a file	<code>command 2&gt; output</code>
redirect stdout to stderr	<code>command 1&gt;&amp;2 output</code>
redirect stderr to stdout	<code>command 2&gt;&amp;1 output</code>
redirect stderr and stdout to a file	<code>command &amp;&gt; output</code>

Reading: [Bash Redirections](#) (spec), [bash hackers redirections](#) (examples)



### 3.5.4 Command Substitution

Command substitution allows the output of a command to replace the command itself. Command substitution occurs when a command is enclosed as follows:

```
$(command)
```

or

```
`command`
```

by executing *command* in a subshell environment and replacing the command substitution with the output, with any trailing newlines deleted. Embedded newlines are not deleted, but they may be removed. Command substitution `$(cat file)` can be replaced by the equivalent but faster `<(cat file)`.

When the backquoted form of substitution is used, backslash retains its literal meaning except when followed by `'`, `"`, or `\`. A backslash followed by a backslash terminates the command substitution. When using the `$(command)` form, all characters are treated literally; none are treated specially.

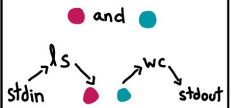
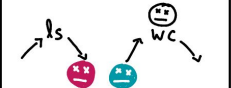
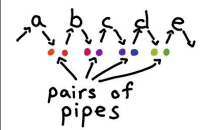
To nest when using the backquoted form, escape the inner backquotes with backslashes.

When using the `$(command)` form, word splitting and filename expansion are not performed on the results.

# pipes

JULIA EVANS  
@b0rk

drawings.jvns.ca

<p>Sometimes you want to send the <u>output</u> of one process to the <u>input</u> of another</p> <pre>\$ ls   wc -l</pre> <p>53 ← 53 files!</p>	<p>a pipe is a pair of 2 magical file descriptors</p> <p>● and ●</p> 	<p>When <code>ls</code> does <code>write(●, "hi")</code> <code>wc</code> can read it! <code>read(●)</code> → "hi"</p>
<p>pipe buffers</p> <p><code>ls</code> "I'm gonna write a bajillion bytes to ●"</p> <p>"uh no if my buffer is full you have to wait" ●</p>	<p>what if your target process dies?</p>  <p><code>ls</code> gets sent <code>SIGPIPE</code> if ● gets closed (<code>ls</code> usually dies)</p>	<p>you can pipe SO MANY things together</p> <pre>\$ a   b   c   d   e</pre>  <p>pairs of pipes</p>

# Bash (shell) Language

- Bash acts as a language interpreter
  - Commands are subroutines with arguments
  - Bash interprets the arguments & calls subroutine
  - **Bash also has its own variables and logic**

# Towards Scripts

- Shell has a state (working directory, user, aliases, history, streams)
- Can expand state with variables
- ‘Source’ runs a file and changes state

```
Printenv  
echo $PS1  
echo $PWD  
echo $PATH
```

# Special Variables

Common variables which set shell state:

\$HOME - sets home directory. \$HOME=~ /CSE374 would reset your home directory to always be CSE374

\$PS1 - sets prompt

\$PATH - tells shell where to look for things. Often extended:

\$PATH=\$PATH:~/CSE374

Show current state: `printenv`

# Variables & Alias

Define variable

```
i=15
```

Access variable

```
ði
```

Undefined variable is empty

string

```
i=5
echo ði
echo hip hip
alias cheer="echo yay"
cheer
```

```
Alias cheer="echo yahoo\!"
```

Defines a shortcut, or 'alias' to a command

Essentially a super simple script.

```
.bashrc
```

# Alias

Defines a shortcut or 'alias' to a command.

*(Essentially a really easy script)*

Also, 'alias'

.bash\_profile

.bashrc

.bashrc

- Executed for login shells
- Use for commands run once
  - Changing \$PATH

- Executed for non-login shells
- Use for commands that are re-run
  - Aliases & functions

```
echo 'alias greet="echo hello $USER"' >> ~/.bashrc
echo mntclassdir cse374 >> ~/.bashrc
greet
source ~/.bashrc
greet
```

# Towards Scripts

- Shell has a state (working directory, user, aliases, history, streams)
- Can expand state with variables
- ‘Source’ runs a file and changes state
- Can run a file without changing state by running script in new shell.
- Allows for repeatable processes and actions

# Variables useful in a script

`$#` stores number of parameters (strings) entered

`$0` first string entered - the command name

`$N` returns the Nth argument

`$?` Returns state of last exit

`$*` returns all the arguments

`$@` returns a space separated string with each argument

(\* returns one word with spaces, @ returns a list of words)



# Variables

Shell has a state, which includes shell variables

All variables are strings (but can do math, later)

White space matters - not spaces around the '='

Create: `myVar=` or `myVar=value`

Set: `myVar=value`

Use: `$myVar`

Remove: `unset $myVar`

List variables (use `'set'`)

# Export Variables

Use: `export myVar`

To make variable available in the initial shell environment.

If a program changes the value of an exported variable it does not change the value outside of the program

`: export -n` remove export property

Variables act as though passed by value

# Okay, lets make a script!

1. First line of file is `#!/bin/bash` (specifies which interpreter to execute)
2. Make file executable (`chmod u+x`)
3. Run a file `./myNewScript`
4. Shell sees the shell program (`/bin/bash`) and launches it to run the script
5. Can include
  - a. String tests (string returns true if non-zero length, `string < string`, etc.)
  - b. Logic (`&&`, `||`, `!`) - use double brackets
  - c. File tests (`-d` : is directory, `-f`: is file, `-w`: file has write permission etc.)
  - d. Math - use double parens

# Script Arguments & Errors

Script refers to  $i^{\text{th}}$  argument at  $\$i$  ;  $\$0$  is the program

Use 'shift' to move arguments towards left ( $\$i$  become  $\$i-n$ )

Exit your shell with 0 (normal) or 1 (error)

# Exit Codes

Command 'exit' exits a shell, and ends a shell-script program.

Exit with no error:

```
Use exit or exit 0
```

Exit with error:

```
User exit 1 or.. {1-255}
```

---

# Quoting Variables

**In order to retain the literal value of something use ‘single quotes’**

**In order to retain all but \$, ` , \ use “double quotes”**

**Put \$\* and @\$ in quotes to correctly interpret strings with spaces in them.**

# Arithmetic

**Variables hold strings, so we need a way to tell the shell to evaluate them numerically:**

**`k=$i+$j` does not add the numbers**

**Use the shell function `((`**

**`k=$(( $i+$j ))`**

**Or `let k="$i+$j"`**

**The shell will automatically convert the strings to the numbers**

# Functions and local variables

**Yes, possible**

**Generally, a script's variables are global**

```
name () compound-command [ redirections ]
```

or

```
function name [( )] compound-command [ redirections ]
```

Ex:

```
func1()  
{  
    local var='func1 local'  
    func2  
}
```



# Stuff to watch out for

White space: spacing of words and symbols matters

Assign WITHOUT spaces around the equal, brackets are WITH SPACES

Typo on left creates new variable, typo on right returns empty string.

Reusing variable name replaces the old value

Must put quotes around values with spaces in them

Non number converted to number produces '0'

# Conditionals

Binary operators: `-eq -ne -lt -le -gt -ge`

Can use the `[[` shell command to use `<`, `>`, `==`

Syntax is a little different, but commands works as expected

```
if test; then
    commands
fi
```

```
while test; do
    commands
done
```

```
for variable in words; do
    commands
done
```

# Flow control

```
test expression or [ expression ]
```

```
if [ -f .bash_profile ]; then
    echo "You have a .bash_profile.
Things are fine."
else
    echo "Yikes! You have no
.bash_profile!"
fi
```

[http://linuxcommand.org/lc3\\_man\\_pages/testh.html](http://linuxcommand.org/lc3_man_pages/testh.html)

# Shell-scripting Notes

Bash Scripting

Interpreted

Esoteric variable access

Everything is a string

Easy access to files and program

Good for quick & interactive programs

Java Programming

Compiled

Highly structured, Strongly typed

Strings have library processing

Data structures and libraries

Good for large complex programs

# Scripting Style Guide

Scripts should generally be <200 lines

*Do one thing and do it well.*

Always use spaces, not tabs (indent line with two spaces)

Comment code with ‘#’

<https://google.github.io/styleguide/shell.xml>

# Emacs (text editor)

C-x C-s #save

C-x C-c # quit

C-e # go to end of line

C-a # go to beginning of line

C-x C-f # find a file

C-g #exit menu

C-x C-k # kill a buffer

You can use any text editor you like. Emacs is amazingly powerful, and highly customizable with lisp scripts. It is probably worth learning.