CSE 374 Lecture 5

Scripting Continued

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)

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 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 string with spaces, @ returns an array of words)

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.

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

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

Flow control

if *test*; then
    commands
fi

while *test*; do
    commands
done

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

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

Shell-scripting Notes

Bash Scripting	Java Programming
Interpreted	Compiled
Esoteric variable access	Highly structured, Strongly typed
Everything is a string	Strings have library processing
Easy access to files and program	Data structures and libraries
Good for quick & interactive programs	Good for large complex programs

Scripting Style Guide

Do one thing and do it well.

Scripts should generally be <200 lines

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

Comment code with ‘#’

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

Some useful utilities

Use `man -k` (or `apropos`): find commands with subject search

Use `find`: location a file on a computer (`locate`: locate a file in the directory database)

Use `!?phrase`: execute the last command containing phrase

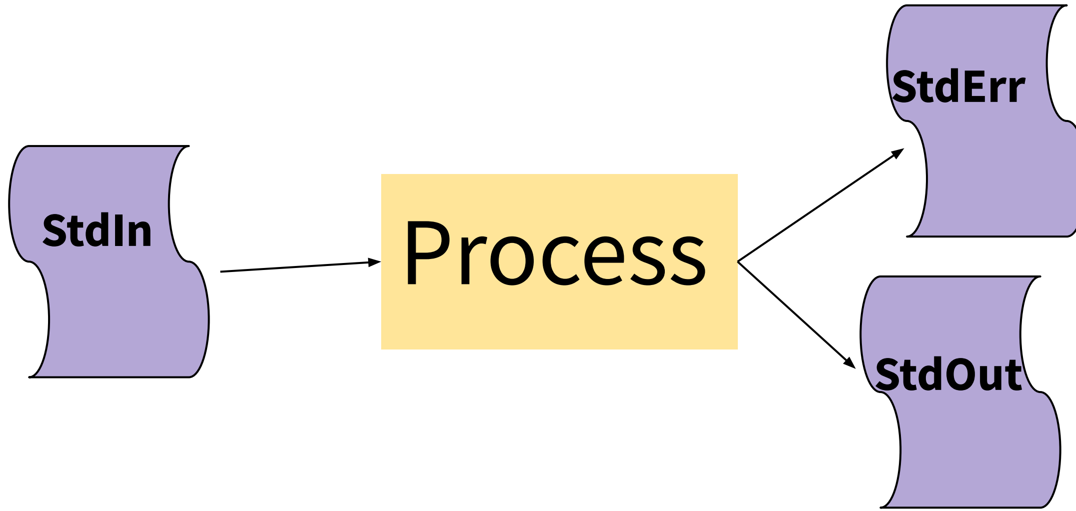
Use `^typo^correct`: correct a typo in the last command

Use `diff f1 f2`: find lines that are different in f2 than in f1 (or `sdiff`)

Up next: Regular expressions

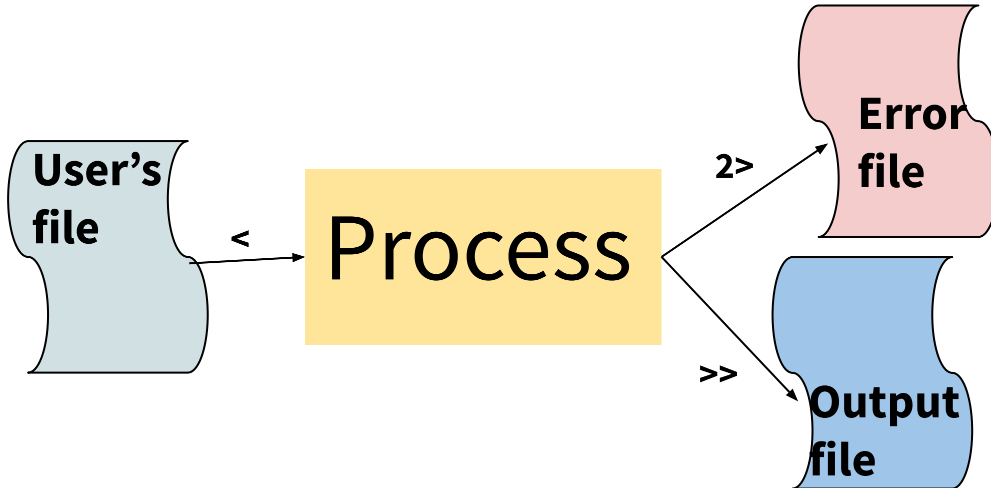
Regular expressions: string of symbols and characters used for pattern matching

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.