ECE364
Software Engineering Tools Lab

Lecture 2
Bash
8/27/2018

Outline

- Variables
- Math
- Branching
- Script I/O
- Looping
- Simple Text Operations
- ~~Arrays~~ (online only)
- I/O Redirection
- Pipes
- Quotes
- Capturing Command Output
- Commands: cat, head, tail, cut, paste, wc

The Shell

- The shell is a program that interfaces users with the operating system
- The Graphical Shell
  - Point an click on things, sometimes you need to use the keyboard
- The Command Line Shell
  - Interactive mode: type commands at the prompt
  - Batch mode: run a multiple commands in a shell script

The Shell (2)

- What does the shell do? (Bash Manual 3.1.1)*

1. Read input from: a file, a string, or a user terminal
2. Break input into words and operators.
3. Parse tokens into simple & compound commands.
4. Perform shell expansions.
5. Execute commands.
6. Wait for commands to complete and collect exit codes.

*http://www.gnu.org/software/bash/manual/bashref.html#Shell-Operation

The Shell (3)

- A shell script should start with a special line:
  - `#! /absolute/path/to/the/shell`
  - This line indicates what shell program the OS should run
  - Allows a script to be run no matter what shell you are currently using
- Some common shells include:
  - `#! /bin/sh` Bourne Shell
  - `#! /bin/ksh` KornShell
  - `#! /bin/bash` Bourne-Again Shell (Bash)
  - `#! /bin/csh` C-Shell
  - `#! /bin/tcsh` TC-Shell

Return Codes

- When a command terminates it returns a numeric value called the return code
  - Recall returning and integer at the end of your main() function in C
  - A return value of zero indicates successful termination
- The `exit <n>` command terminates the current script with a return value `<n>`

Running Bash Scripts

- A bash shell script can be run by providing it to the bash executable program explicitly:
  - `bash MyScript.sh <arg1> <arg2> …`
- If the shell script contains a #! line then that shell will be invoked using the specified program
  - `./MyScript.sh <arg1> <arg2> …`
  - You will need execute permissions to run your script

Commands in Bash

- Commands must be separated from one another with a semicolon or newline

```
Cmd1; Cmd2; Cmd3
Cmd4
Cmd5
```

- If your line gets too long you can continue on the next line by adding a \ (backslash)

```
Cmd6 With a very long set of command arguments \
that span multiple \
lines
```

Commands in Bash (2)

- Some commands are built-in to the shell, others may be external shell scripts or programs
- When Bash executes your command it will first see if it is a built-in command
  - If not, it will attempt to execute your command as an external program
- Commands in bash can always be entered manually at the command prompt
  - Even complicated commands like `for` or `if`!

Commenting Bash Scripts

- Comments are denoted by a the pound sign `#`
  - Each line of a comment must start with `#`
- The `#!` line is called the shebang line.
  - Specifies the *interpreter* that should be used for the remainder of the file (e.g., bash, python, etc.).
  - The Wikipedia article about the shebang line is 8 pages!!!
    - https://en.wikipedia.org/wiki/Shebang_(Unix)

Debugging Bash Scripts

- Debugging features are controlled by providing additional arguments to the shell program
  - `bash [options] ./MyScript.sh`
  - `#! /bin/bash [options]`

Option	Description
`-n`	Check the script for syntax errors but do not execute any commands
`-x`	Prints out commands as they are executed

- The `-x` option is especially useful for observing what commands are executed as your script runs
  - Often easier then littering your code with print statements

Variables in Bash

- To assign and declare a variable:

```
VAR_NAME=value
```

- To access the variable:

`$VAR_NAME` or `${VAR_NAME}`

- Some Examples:

```
Var1=7
Var2=Hello!
Var3=$Var1      # Var3=7
Var4=${Var2}    # Var4=Hello!
```

NOTE: NO WHITESPACE BETWEEN THE EQUAL SIGN!

Variables in Bash (2)

- Before each command is executed, any variables are replaced with their current value, even in strings
  - Called variable substitution
  - Run your script with the –x shell debug option to see

```
> foovar=42

> echo foovar  # $ needed to substitute
foovar         # foovar looks like a string

> echo $foovar # becomes "echo 42"
42
```

Variables in Bash (3)

- Curly braces are optional when all you want is a simple variable substitution
- Sometimes you must use them to disambiguate between a variable name and adjacent characters in a string
- You should aim to improve readability.

```
> number=10
> echo "There are $numbers of people"
There are  of people
> echo "There are ${number}s of people"
There are 10s of people
```

Special Variables in Bash

- Set by Bash automatically and can not be assigned directly

`$#`	Number of command line arguments to the script
`$@`	All the command line arguments to the script
`$0`	The relative path to your script (includes its name)
`$$`	Current process ID number
`$?`	Return code from last executed command
`$1 to $N`	The Nth command line parameter
`$RANDOM`	A random integer value

Bash Math

- Bash supports basic math on integers
- Use `let` or `(( … ))` to isolate mathematical statements
  - You will get syntax errors if you forget this!
  - You can exclude $ from variable names in arithmetic evaluation
- Operators: `+, -, *, /, <<, >>, %`

Integer Math Example

- The let command indicates a mathematical expression

```
let a=66+11        # a is 77
let b=$a*2         # b is 154
let c=5/2          # c is 2
let d=(a-c)*6      # d is 450
```

- Alternatively you can enclose the mathematical expression in double parenthesis `((…))`

```
((a=66+11))        # a is 77
((b=$a*2))         # b is 154
((c=5/2))          # c is 2
((d=(a-c)*6))      # d is 450
```

Conditional Testing

- Bash has several flavors conditional tests
  - Arithmetic Tests – Compare numbers
  - File Tests – Check properties of files
  - String Tests – Compare string values
- Syntax for each test command varies depending on the flavor of test

Conditional Testing (2)

- When a command completes it will set the return value variable: `$?`
  - A return value of 0 indicates: success/true
  - A non-zero return value indicates: failure/false
  - This is opposite to programming languages!
- Any conditional test preformed in Bash is checking the return value of a command

Conditional Testing (3)

```
(( 5 == 5 ))          # is 5 == 5?
echo $?

[[ -e /etc/passwd ]]  # does the file exist?
echo $?

[[ -z "Nope" ]]       # is the string empty?
[[ "foo" != "bar" ]]  # this will overwrite $? !!!
echo $?
```

The above commands will produce the output:

```
0 0 0
```

Conditional Testing (4)

- Any conditional test may be inverted using the not operator (`!`) before the test expression

```
(( ! 5 == 5 ))
[[ ! -e /etc/passwd ]]
[[ ! -z "Nope" ]]
```

Conditional Testing (5)

- Multiple tests can be combined using AND (`&&`) and OR (`||`) operators

```
(( 5 == 5 && 6 == 6 ))
[[ ! -f /etc/passwd || -d /etc ]]
[[ ! -z $input && $input < "foo" ]]
```

- If you need to mix test flavors the operators can be placed outside of the parenthesis or brackets

```
(( 5 == 5 )) && [[ ! -d /etc ]]
```

Arithmetic Test Expressions

Expression	Description
`x == y`	True if x **is equal to** y
`x != y`	True if x **is not equal to** y
`x < y`	True if x **is less than** y
`x > y`	True if x **is greater than** y
`x <= y`	True if x **is less than or equal to** y
`x >= y`	True if x **is greater than or equal to** y

Example Usage:

```
(( $var_y <= $var_z ))    (( 5 != 9 ))
```

File Testing

- A significant part of many shell scripts is devoted to file tests
  - Does a file exist?
  - Is a file readable? writable? executable?
  - Is the file a directory?
  - Is the file empty?
- In practice you should always perform the appropriate file tests before operating on files

File Testing Expressions

Expression	Description
`-e <file>`	True if <file> **exists**
`-f <file>`	True if <file> **is a regular file**
`-d <file>`	True if <file> **is a directory**
`-r <file>`	True if <file> **is readable**
`-w <file>`	True if <file> **is writable**
`-x <file>`	True if <file> **is executable**
`-s <file>`	True if <file> **exists and is not empty**

Example Usage:

```
[[ -e $my_file ]]   [[ ! -x /bin/bash ]]
```

String Test Expressions

Expression	Description
`-z <str>`	True if <str> **is empty**
`-n <str>`	True if <str> **is not empty**
`<str1> = <str2>`	True if <str1> **is equal to** <str2>
`<str1> != <str2>`	True if <str1> **is not equal to** <str2>
`<str1> < <str2>`	True if <str1> **is lexicographically ordered before** <str2>
`<str1> > <str2>`	True if <str1> **is lexicographically ordered after** <str2>

Example Usage:

```
[[ -z $input ]][[ "foo" < "bar" ]]
```

`if` Command

```
if <command/conditional test>
then
     <commands>
elif <command/conditional test>
then
     <commands>
else
     <commands>
fi
```

`if` Command (2)

```
if gcc file.c
then
      echo "You code compiles!"
else
      echo "Try again…"
fi

if [[ -d $filename ]]
then
      echo "The file is a directory!"
elif [[ -f $filename ]]
      echo "The file is a regular file!"
fi
```

`if` Command (3)

```
read -p "Enter a number: " num
if (( $number < 10 ))
then
      printf "The number %d is too small!\n" $num
elif [[ -f /numbers/${num} ]]
then
      printf "The number %d already exists!\n" $num
else
      echo "$num" > /numbers/${num}
fi
```

`if` On A Single Line

```
[[ $DEBUG == "YES" ]] && save_output
```

Is equivalent to:

```
if [[ $DEBUG == "YES" ]]
then
    save_output
fi
```

```
[[ -e mydir ]] || mkdir mydir
```

Is equivalent to:

```
if [[ ! -e mydir ]]
then
    mkdir mydir
fi
```

Brace Expansion

- A sequence of elements like a1, a2, a3, … z1, z2, z3 can be generated using brace expansion
  - `{1..10}` `# 1 2 3… 10`
  - `{a..e}` `# a b c d e`
  - `{a..z}{1..3}` `# a1 a2 a3…z1 z2 z3`
  - `ee364{a..f}{1..9}` `# ee364a1 ee364a2…`
  - `a{b,C,5}f` `# abf aCf a5f`
  - `{1,2}x{a..b}` `# 1xa 1xb 2xa 2xb`

Globs (Pathname Expansion)

- A glob is a pattern that expands to match file names

`*`	Matches everything
`*.foo`	Matches strings ending in .foo
`ee364*`	Matches strings starting with ee364
`*bar*`	Matches string containing “bar”
`*.[ch]`	Matches strings ending in .c or .h
`?`	Matches any single character
`JK[0-9]???`	Matches JK followed by any 0-9 digit and three other characters. Ex: `JK4x2z` or `JK87bb`

Globs (2)

- Globs are expanded into a list of file names that match the glob
- Examples:

```
ls *.c

cat ee364*.log

rm –f accounts/ee364???/Lab*/*.bash
```

Globs (3)

- If no files match a glob the string will not be expanded!
- Example:

```
cat *junk_dsfsfsdfsf
cat: *junk_dsfsfsdfsf: No such file or directory
```

Globs (4)

- Brace expansion can be combined with globs to form even more complex patterns
- Example:

```
ls /pics/*.{jpg,png,gif}

# Same as above
ls /pics/*.jpg /pics/*.png /pics/*.gif
```

`echo` Command

- `echo [options] [string]`
  - Prints a string to standard output (the terminal)

Option	Meaning
`-n`	Disable the automatic newline
`-e`	Treat `\` as an escaping character

```
> age=23
> echo “You are ${age} years old.”
You are 23 years old

> echo “Hello \name”     > echo -e “Hello \name”
Hello \name                Hello
                               ame
```

`printf` Command

- Useful when you need to format output
  - Uses the same format string e.g. %s %d…
  - Arguments are separated by a space
  - No automatic newline

```
printf "Magic number is %d\n"  $RANDOM
printf "My name is %s\n" Goldfarb
printf "Pi = %1.2f e = %1.2f\n" 3.14159 2.71828
```

`read` Command

- `read [-p prompt] [variable]`
- Reads a single line from standard input into a variable

```
echo -n "Enter a line of text:"
read aLineOfText
echo "You entered: " $aLineOfText

read -p "How old are you? " age
echo "You are $age years old"

echo "Press [ENTER] to continue…"
read
```

`read` Command (2)

- The read command can populate more then one variable
  - e.g. `read First Second Third Rest`
  - Each variable will contain a word of text
  - The last variable will get remaining contents of the line

`for` Command

- The `for` command executes a loop over a set of elements in a list
  - **A list can be anything separated by whitespace**

```
for <var> in <list>
do
  <…commands…>
done
```

- A more C-like `for` command syntax is also allowed:

```
for ((<pre-cond>; <cond>; <iter-step>))
do
  <…commands…>
done
```

`for` Command (2)

```
for I in 1 2 3 4 5
do
  echo -n ${I}
done


for I in {1..5}
do
  echo -n ${I}
done


for (( I=1; I < 6; I++ ))
do
  echo -n ${I}
done
```

All three result in the same output:

```
12345
```

`for` Command (3)

- The list of a for loop can also be globs/brace expansions for iterating over files

```
# With globs
for File in *.c
do
  # Print all C source files
  lp -dSOME_PRINTER $File
done

# With brace expansion and globs
for File in /students/ee364{a..f}*.c
do
  # Compile all student files
  cc -Wall -lm -O3 -o${File}.o ${File}
done
```

Loop Control Commands

`continue`

- Used to skip to the next iteration of the inner-most loop

`break`

- Used to end the execution of the inner-most loop

Command Line Arguments

- What if we want to loop through the command line arguments?
  - Easy `$@` is a list of arguments

```
for arg in $@
do
  echo $arg
done
```

`while` Command

- Run a set of commands until a conditional test or command returns a non-zero value (false)

```
while gcc student_file.c
do
    echo "Student code still compiles!"
    inject_errors student_file.c
done

while (( $RANDOM % 10 != 0 ))
do
    echo "Still no luck!"
done
```

`while` Command (2)

- The read command is typically used with a while loop to process lines of text

```
while read line
do
  echo "$line"
done < $1 # Redirect into the loop!
```

- `$1` will be redirected into standard input of the while command which will execute read until all lines are read
- Note the placement of the redirect at the ***END*** of the while command

`while` Command (3)

- Like the conditional tests `[[…]]` and `((…))` other command tests can be inverted with a `!` Operator

```
while ! gcc student_file.c
do
    echo "Student still has bugs!"
    correct_errors student_file.c
done
```

`shift` Command

- Left shifts the parameters on the command line by `n` (default `n = 1`) places
- The `$0` parameter is NEVER shifted

`shift` Command (2)

```
echo '$0 -- ' $0
echo '$# -- ' $#
X=0
while (( $# != 0 ))
do
  ((X=X+1))
  echo "\"\$${X}\" was $1"
  shift
done
```

Example usage:

```
$ parameters q "1 2 3" xyz
$0 -- ./parameters
$# -- 3
"$1" was q
"$2" was 1 2 3
"$3" was xyz
```

`cat` Command

- `cat [option] [files]`
- Concatenates and prints the contents of each file
  - Standard input is used if no files are provided
  - A hyphen (`-`) may be used as one of the files to indicate standard in as an additional source of input

Option	Description
`-n`	Include line numbers for each line
`-s`	Remove extra empty lines so that there is at most one empty line between two non-empty lines
`-b`	Include line numbers of non-empty lines

`head` Command

- `head [option] [files]`
- Prints the beginning each file specified
  - Standard input is used if no files are provided

Option	Description
`-n <N>`	Displays the first `<N>` lines
`-n -<N>`	Displays all but the last `<N>` lines
`-c <N>`	Displays the first `<N>` characters/bytes

`tail` Command

- `tail [option] [files]`
- Prints the end (tail) each file specified
  - Standard input is used if no files are provided

Option	Description
`-n <N>`	Displays the last `<N>` lines
`-n +<N>`	Displays all lines starting at line `<N>`
`-c <N>`	Displays the last `<N>` characters
`-c +<N>`	Displays all characters starting at the `<N>`th character

`wc` Command

- `wc [options] [files]`
- Counts the number of lines in one or more files
  - Standard input is used if no files are provided

Option	Description
`-w`	Count the number of words in each file
`-l`	Count the number of lines in each file
`-c`	Count the number of characters in each file

`cut` Command

- `cut [options] [files]`
- Cuts out columns from one or more files
  - Standard input is used if no files are provided
  - Delimiters may only be single characters

Option	Description
`-d<D>`	Specifies the character `<D>` as the field delimiter. The default field delimiter is a `TAB` character
`-s`	Ignore lines that do not contain any delimiter characters
`-f<fields>`	Specifies a range or set of fields to include. A range can be a valid numeric range (e.g. 3-6) or a list of individual fields (e.g. 1,3,7)
`-c<chars>`	Specifies a range or set of character to include. A range can be a valid numeric range (e.g. 3-6) or a list of individual characters (e.g. 1,3,7) Note: No delimiter is set when cutting characters.

`cut` Command (2)

- Assume the file "tabdata" contains:

```
001    Mike Goldfarb       mgoldfar
002    Jacob Wyant         jwyant
003    Jung Yang           yang205
004    Aarthi Balachander  abalacha
```

- To print the record #s (first 3 characters):

```
$ cut -c 1-3 tabdata
001
002
003
004
```

`cut` Command (3)

- Assume the file "tabdata" contains:

```
001    Mike Goldfarb       mgoldfar
002    Jacob Wyant         jwyant
003    Jung Yang           yang205
004    Aarthi Balachander  abalacha
```

- To print the 2nd column (field):

```
$ cut -f2 tabdata
Mike Goldfarb
Jacob Wyant
Jung Yang
Aarthi Balachander
```

`cut` Command (4)

- Assume the file "tabdata" contains:

```
001    Mike Goldfarb       mgoldfar
002    Jacob Wyant         jwyant
003    Jung Yang           yang205
004    Aarthi Balachander  abalacha
```

- To print the 1st and 3rd column (field):

```
$ cut -f1,3 tabdata
001    mgoldfar
002    jwyant
003    yang205
004    abalacha
```

`paste` Command

- `paste [options] [files]`
- Joins lines together from one or more files
  - Opposite of the `cut` command
  - Delimiters may only be single characters

Option	Description
`-d<D>`	Specifies the character `<D>` as the field delimiter. The default field delimiter is a `TAB` character
`-s`	Paste files horizontally

`paste` Command (2)

- Assume the file "accounts" contains

```
ee364a01
ee364a02
```

- Assume the file "names" contains

```
Michael Goldfarb
Jung Yang
```

- To combine accounts and student names:

```
$ paste -d':' accounts names
ee364a01:Michael Goldfarb
ee364a02:Jung Yang
```

`paste` Command (3)

- Assume the file "accounts" contains

```
ee364a01
ee364a02
```

- Assume the file "names" contains

```
Michael Goldfarb
Jung Yang
```

- Using the `-s` option to paste horizontally:

```
$ paste -s -d',' accounts names
ee364a01,ee364a02
Michael Goldfarb,Jung Yang
```

Array Variables

- Declaring and initializing array: `A=(1 foo 2)`
- Accessing an array element: `${A[index]}`
  - Index may be a non-negative variable or number
- Getting all elements in an array: `${A[*]}`
  - `${A}` will only get the first element of the array!
- Assign an array element: `A[index]=<value>`
  - Can assign non-consecutive indices, arrays are sparse
  - Different from C, where an array elements are always contiguous
  - **Array indices start at zero!**

Arrays Variables (2)

- To get the size of an array: `${#Array[*]}`
- To get a list of indices: `${!Array[*]}`
- **Attempting to access an unset array index will simply return an empty string**
- When would a list of array indices be useful or necessary?

Array Variables (3)

```
A=(foo bar baz)
A[5]=cosby
A[10]=jello
```

```
for item in ${A[*]}
do
    echo $item
done
```

```
for item in ${!A[*]}
do
    echo ${A[$item]}
done
```

```
for ((I = 0; I < ${#A[*]}; I++))
do
    echo ${A[$I]}
done
```

What is the problem in the bottom for loop?

Reading Into an Array

```
while read -a Data  # Splits on whitespace
do
  echo Read ${#Data[*]} items.
  echo The third item is ${Data[2]}.
done < Some_Data_File
```

- Use the `-a` option of the read command to split each line read from `Some_Data_File` into an array
  - Note: read will still only read one line at a time

Converting to Arrays

- It is often helpful to convert scalar variables to arrays

```
values="1 2 3 4 5"
arrval=($values)
for i in ${arrval[*]}
do
  echo -n "$i "
done
```

Will print:

```
1 2 3 4 5
```

Converting to Arrays (2)

- The `set` command will convert a scalar into an array by setting each value to the command line parameter variables (`$n`):

```
values="a b c d e"
set $values
echo $1 $2 $3 $4 $5
```

Will print:

```
a b c d e
```

I/O Redirection

- Many commands and programs read and write to the standard file streams

```
$ ./setup.sh
What is your name?: Foo Bar
What is your age?: 31
```

- For example the above script prints some text to the screen and accepts input from the keyboard
  - Standard input and standard output

I/O Redirection (2)

- It is also possible to take input and output from non-standard sources using I/O redirection
- Input redirection takes input from a source such as a file or hardware device and directs it to standard input (`stdin`)
- Output redirection takes output from a program and directs it to standard output (`stdout`)

I/O Redirection (3)

- When the operating system reads and writes to a file is uses a special number called the file descriptor to identify the open file
  - Think of a file descriptor as the `FILE*` from C
- File descriptors allow you to precisely specify the file you want to read from or write to
  - By default it is assumed that you will read from standard input and write to standard output

I/O Redirection (4)

- The standard file descriptors are ALWAYS assigned the numbers:

Name	File Descriptor #
Standard Input (`stdin`)	0
Standard Output (`stdout`)	1
Standard Error (`stderr`)	2

- If you do not explicitly specify file descriptor numbers `stdin` or `stdout` are *usually* assumed

I/O Redirection (5)

- Redirect data into a command with `<`

  `<infile`

  `n<infile` `n` is the file descriptor number

```
# Redirect my_document into stdin
mail mgoldfar@purdue.edu < my_document

# Redirect work into file descriptor 4
grade_lab L1 4< work
```

I/O Redirection (5)

- Redirect data out from a command with `>`

`>file`	Redirect `stdout` into `file` and overwrite
`n>file`	Redirect output from file descriptor `n` into `file`
`>>file`	Append `stdout` to the contents of `file`
`n>>file`	Append output from file descriptor `n` into `file`

```
ls *.c  > source_files
ls *.h  >> source_files    # Append to source_files

# Redirect output from stderr (#2) to /dev/null
cc -Wall -O3 -oFile.o -cFile.c 2>/dev/null
```

Advanced I/O Redirection

- We can assign additional file descriptors if we need to read and write to multiple sources simultaneously
- A special `exec` command "opens" a new file descriptor that can be read to or written from

Statement	Description
`exec n<file`	Assigns file descriptor `n` to `file` for reading
`exec n>file`	Assigns file descriptor `n` to `file` for writing
`exec n>>file`	Assigns file descriptor `n` to `file` for appending

Advanced I/O Redirection (2)

- You can also redirect from one file descriptor to another

`<&n`	Redirects file descriptor `n` into `stdin`
`m<&n`	Redirects file descriptor `n` into file descriptor `m`
`>&n`	Redirects stdout to file descriptor `n` out to `stdout`
`m>&n`	Redirects file descriptor `m` out to file descriptor `n`

Advanced I/O Redirection (3)

- By default the read command reads input from `stdin` and echo writes output to stdout
  - This can be changed with I/O redirection
- `read [var1 var2 … varN] <&n`
  - Reads a line from file descriptor `n`
- `echo [options] [string] >&n`
  - Prints to file descriptor `n`

Advanced I/O Redirection (4)

```
# Open logfile.txt for writing
exec 4> logfile.txt

# Print a message to stdout
echo "Writing logfile…"

# Write to the logfile (notice the >&4)
echo "This will be written to logfile.txt" >&4
```

Advanced I/O Redirection (5)

```
# Open logfile.txt for reading
exec 4< logfile.txt

# Get the number of lines to read from stdin
read -p "how many lines? " nlines

# Print out each line by reading it
for (( i = 1; i <= $nlines; i++ ))
do
      # Read a line from logfile.txt
      read line <&4
      echo "Line $i: $line"
done
```

Advanced I/O Redirection (6)

- Why do we need to assign a file descriptor? Why not redirect directly from a file?

```
# Print out each line by reading it
for (( i = 1; i <= $nlines; i++ ))
do
  # BUG! Will always read the first line of logfile.txt
  # A descriptor will remember where to continue reading
  read line < logfile.txt
  echo "Line $i: $line"
done
```

Advanced I/O Redirection (7)

```
# This example shows how to read from multiple files
# Assume the input files have equal number of lines

exec 3< $1 # 1st argument is input file name
exec 4< $2 # 2nd argument is input file name
exec 5> $3 # 3rd argument is output file name

# Read from the first input file until the end
while read lineA <&3
do
  # Read one line from the second input file
  read lineB <&4

  # Write output to file descriptor 5
  echo "$lineA // $lineB" >&5
done
```

Special Files

- In Unix systems there are several special files that provide useful behaviours:
- `/dev/null`
  - A file that discards all data written to it
  - Reading always produces `<EOF>`
- `/dev/zero`
  - A file that discards all data written to it
  - Reading always produces a string of zeros
- `/dev/tty`
  - The current terminal (screen and keyboard) regardless of redirection

Pipes

- Pipes take output from one command and pass it as input to the next command

```
command_1 | command_2 | … | command_n
```

- `command_1` sends output to `command_2`
- `command_2` receives input from `command_1`
- `command_2` sends output to `command_3`…

- Example: Count the number of words in a file

```
$ cat TheWealthOfNations.txt | wc -w
  380599
```

`tee` Command

- `tee [-a] <file>`
- Sends all input from `stdin` to `stdout` and also to `<file>`

- Use the tee command when you need to save intermediate output of a command sequence

```
cmd1 | tee cmd1.out | cmd2
```

`tee` Command (2)

- The tee command overwrites the contents of its file
- Use the `-a` option to force tee to append to the file

```
cmd1 | tee -a cmd1.out | cmd2
```

Quotes

- There are various kinds of quotes, and each one can mean something different
  - `'` The single forward quote character
  - `"` The double quote character
  - `` ` `` The back quote character
  - `\` The backslash character
    (often used to begin an escape sequence)

Single Quotes

- Must appear in pairs
- Protects all characters between the pair of quotes
- Ignores all special characters
- Protects whitespace

Single Quotes (2)

```
$ Name='Ekim Brafdlog'

$ echo Welcome to ECE364 $Name
Welcome to ECE364 Ekim Brfdlog

$ echo 'Welcome to ECE364 $Name'
Welcome to ECE364 $Name

$ echo 'The book costs $2.00'
The book costs $2.00
```

Single Quotes (3)

- A star (*) character has some confusing behaviour:
  - Used within single quotes * is **NOT** expanded
  - **Except** when assigning it to a variable

```
$ echo *                $ files='*'
File1 File2 File3       $ echo $files
                        File1 File2 File3
$ echo '*'
*
```

Double Quotes

- Must come in pairs
- Protects whitespace
- Does **NOT** ignore the following characters
  - Dollar Sign $
  - Back Quote `
  - Backslash \

Double Quotes (2)

```
$ Path="/b/ee264"
$ echo "The path for ee364 is $Path"
The path for ee364 is /b/ee364

$ echo "The book costs \$2.00"
The book costs $2.00
```

Note: Since double quotes will treat $ as a variable it must be escaped with a backslash

$(command)

- Runs a command and captures its output
  - Capture program output into variables

```
$ echo "Directory is $(pwd)"
Directory is /home/min/a/mgoldfar

$ DIR=$(pwd)
  $ echo "Directory is ${DIR}"
  Directory is /home/min/a/mgoldfar
```

Historical note: Backquote

You will also see the backquote used in the same way, but this is deprecated.

```
echo "Directory is `pwd`"
```

$(command) (2)

- `$(...)` can be used to capture the output from a sequence of commands connected by pipes

```
$ now=$(date | cut -d' ' -f4)
$ printf "The current time is %s\n" $now
The current time is 14:56:02
```

Backquote: `command` (deprecated!)

- Historically, the backquote was used for this purpose, but that has been deprecated.
  - Use `$(...)` instead.

```
$ echo ”Directory is `pwd`”
Directory is /home/min/a/mgoldfar

$ DIR=`pwd`
  $ echo “Directory is ${DIR}”
  Directory is /home/min/a/mgoldfar
```

$((expression))

- Evaluates an arithmetic expression

```
$ echo 11 + 11
11+11

$ echo $(( 11 + 11 ))
22

$ k=99
echo $((k*66))
6534
```

The Backslash \

- Use to remove any special meaning that a symbol may have.
  - e.g `\$1.00` or `\$`
- Used to add special meaning to symbols like `\n` or `\b`
- If it is the last symbol on a line, it will act as a continuation indicator.

The Backslash \ (2)

```
$ echo "This item costs \$2.00”
This item costs $2.00

$ echo “Can you hear anything?\b”

$ echo ”My login ID is” \
      “\”$(whoami)\”” \
      “What is yours?”

My login ID is “mgoldfar” What is yours?
```

Combining head and tail

- Recall how `head` and `tail` works.
- Suppose you wanted to print lines 10 to 20
- Since `head` and `tail` read from `stdin` a pipe can be used to “connect” the commands

```
head -n 20 my_file | tail -n 10
```

- Many of the basic commands in this lecture can be piped together to perform complex operations

wc Command

- wc [options] [files]
- Counts the number of lines in one or more files
  - Standard input is used if no files are provided

Option	Description
-w	Count the number of words in each file
-l	Count the number of lines in each file
-c	Count the number of characters in each file

wc Command (2)

```
$ wc -w TheWealthOfNations.txt
 380599 TheWealthOfNations.txt

$ wc -wl TheWealthOfNations.txt
 35200  380599 TheWealthOfNations.txt

$ wc -c TheWealthOfNations.txt TheWealthOfNations.txt
 2256586 TheWealthOfNations.txt
 2256586 TheWealthOfNations.txt
 4513172 total

# Capturing the number of words:
# Note the conversion to an array:
$ words=($(wc -w *.txt | tail -n1))
echo "There are ${words[0]} in all files."
```

sort Command

- sort [options] [files]
- The sort command sorts data in a set of files
  - Standard input is used if no files are provided
  - Will merge multiple files to produce a single result

Option	Description
-f	Treat lowercase and uppercase letters the same
-k <Start>[,Stop]	Specifies the sort field in a line. If no stop position is specified the end of the line is used. Multiple -k options can be specified to indicate sorting behavior for ties
-n	Treat the field as a numeric value when sorting
-r	Sort in reverse order
-t <X>	Sets <X> as the field separator. TAB and SPACE are the default separators.

sort Command (2)

- Consider a file called "data" that contains:

```
555  Mike Goldfarb          mgoldfar
666  Jacob Wyant            jwyant
777  Jung Yang              yang205
444  Aarthi Balachander     abalacha
```

- To sort by TA name (2nd column):

```
$ sort -k2 data
  444    Aarthi Balachander    abalacha
666  Jacob Wyant            jwyant
777  Jung Yang              yang205
555  Mike Goldfarb          mgoldfar
```

sort Command (3)

- Consider a file called "data2" that contains:

```
ece 201 fff
aaa 100 fff
bbb 199 ggg
ccc 302 fff
```

- To sort on column 3 first and then on column 2:

```
$ sort -k3 -k2 data2
  aaa 100 fff
ece 201 fff
ccc 302 fff
bbb 199 ggg
```

diff Command

- The diff command compares files line by line
- diff <file1> <file2>
  - Will compare file1 with file2 and print a list of differences
- diff --brief <file1> <file2>
  - Will print a short message if file1 differs from file2
- diff will produce a return code of 0 if the files do not differ and 1 otherwise

`diff` Command (2)

data1
```
1 2 3 4
1 2 3 4
1 2 3 4
```

data2
```
1 2 3 4
5 6 7 8
1 2 3 4
```

```
$ diff data1 data2
2c2
< 1 2 3 4
---
> 5 6 7 8
```

Line 2 of data1 was changed to line 2 in data2

`diff` Command (3)

data1
```
1 2 3 4
1 2 3 4
1 2 3 4
```

data2
```
1 2 3 4
1 2 3 4
1 2 3 4
5 6 7 8
```

```
$ diff data1 data2
3a4
> 5 6 7 8
```

Line 4 of data2 was added after line 3 in data1

`diff` Command (3)

data1
```
1 2 3 4
1 2 3 4
1 2 3 4
1 2 3 4
```

data2
```
1 2 3 4
1 2 3 4
1 2 3 4
```

```
$ diff data1 data2
4d3
< 1 2 3 4
```

Line 4 of data1 was removed after line 3 in data2