



# **ECE 364**

## **Software Engineering Tools Lab**

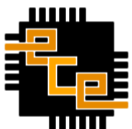
Lecture 2

Bash II



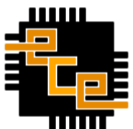
# Lecture 2 Summary

- Programming the Bash Shell (Part 2)
- I/O Redirection and Pipes
- Quotes
- Capturing Command Output
- Commands: cat, head, tail, cut, paste, wc



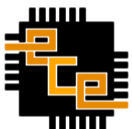
# Array Variables

- 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
```

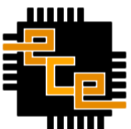
```
A[10]=jello
```

```
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
```

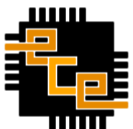
What is the problem in the bottom for loop?



# 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



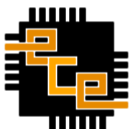
# Converting to Arrays

- 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
```



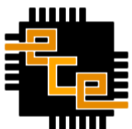
## Converting to Arrays (2)

- 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
```





# Demo: Arrays



# I/O Redirection

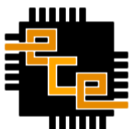
- Many commands and programs read and write to the standard file streams

```
$ ./setup.sh
```

```
What is your name?: Foo Bar
```

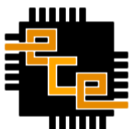
```
What is your age?: 31
```

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



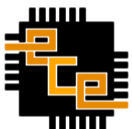
# I/O Redirection (2)

- 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 it 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

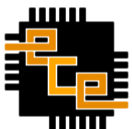


# I/O Redirection (4)

- The standard file descriptors are **ALWAYS** assigned the numbers:

Name	File Descriptor #
Standard Input ( <code>stdin</code> )	0
Standard Output ( <code>stdout</code> )	1
Standard Error ( <code>stderr</code> )	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

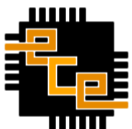
n<infile n is the file descriptor number

# 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 `>`

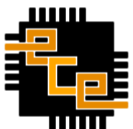
<code>&gt;file</code>	Redirect <code>stdout</code> into <code>file</code> and overwrite
<code>n&gt;file</code>	Redirect output from file descriptor <code>n</code> into <code>file</code>
<code>&gt;&gt;file</code>	Append <code>stdout</code> to the contents of <code>file</code>
<code>n&gt;&gt;file</code>	Append output from file descriptor <code>n</code> into <code>file</code>

```
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`

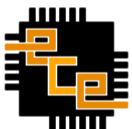
Assigns file descriptor `n` to `file` for reading

`exec n>file`

Assigns file descriptor `n` to `file` for writing

`exec n>>file`

Assigns file descriptor `n` to `file` for appending

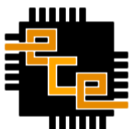




# Advanced I/O Redirection (2)

- You can also redirect from one file descriptor to another

<code>&lt;&amp;n</code>	Redirects file descriptor <code>n</code> into <code>stdin</code>
<code>m&lt;&amp;n</code>	Redirects file descriptor <code>n</code> into file descriptor <code>m</code>
<code>&gt;&amp;n</code>	Redirects <code>stdout</code> to file descriptor <code>n</code> out to <code>stdout</code>
<code>m&gt;&amp;n</code>	Redirects file descriptor <code>m</code> out to file descriptor <code>n</code>

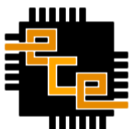


# Demo: I/O Redirection



# 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`

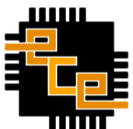


# Advanced I/O Redirection (4)

```
# 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
```

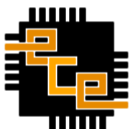


# Advanced I/O Redirection (5)

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

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

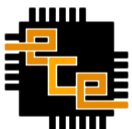
# Print out each line by reading it
for (( i = 1; i <= $nlines; i++ ))
do
    # Read a line from logfile.txt
    read line <&4
    echo "Line $i: $line"
done
```



# 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
```



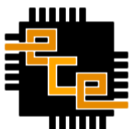
# Advanced I/O Redirection (7)

```
# 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
```

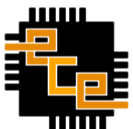
```
# Read from the first input file until the end
while read lineA <&3
do
    # Read one line from the second input file
    read lineB <&4

    # 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

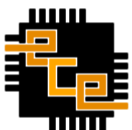
```
command_1 | command_2 | ... | command_n
```

- `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
```

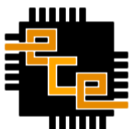
```
380599
```



# 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
```

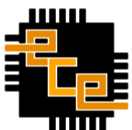


## Demo: Pipes & tee



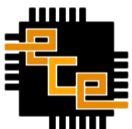
# Quotes

- 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
```



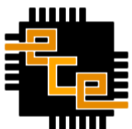
# Single Quotes (3)

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

```
$ echo *  
File1 File2 File3
```

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

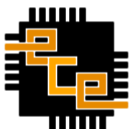
```
$ echo `*`  
*
```





# 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"
```

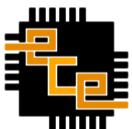
```
$ echo "The path for ee364 is $Path"
```

```
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



# The Back Quote `

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

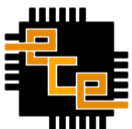
```
$ echo "Directory is `pwd`"
```

```
Directory is /home/min/a/mgoldfar
```

```
$ DIR=`pwd`
```

```
$ echo "Directory is ${DIR}"
```

```
Directory is /home/min/a/mgoldfar
```



## \$ (command)

- Another way to capture command output
  - Prefer using this over the back quote

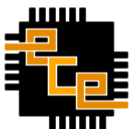
```
$ echo "Directory is $(pwd)"
```

```
Directory is /home/min/a/mgoldfar
```

```
$ DIR=$(pwd)
```

```
$ echo "Directory is ${DIR}"
```

```
Directory is /home/min/a/mgoldfar
```



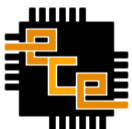
## \$ (command) (2)

- \$ (...) 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
```



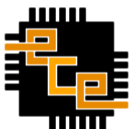
# `$ ( ( expression ) )`

- Evaluates an arithmetic expression

```
$ echo 11 + 11  
11+11
```

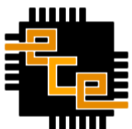
```
$ echo $ ( ( 11 + 11 ) )  
22
```

```
$ 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?
```



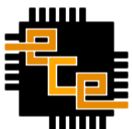


# Combining `head` and `tail`

- 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
```

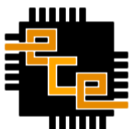
- Many of the basic commands in this lecture can be piped together to perform complex operations



# 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
<code>-w</code>	Count the number of words in each file
<code>-l</code>	Count the number of lines in each file
<code>-c</code>	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
```

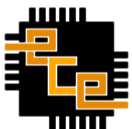
```
$ wc -c TheWealthOfNations.txt TheWealthOfNations.txt
2256586 TheWealthOfNations.txt
2256586 TheWealthOfNations.txt
4513172 total
```

```
# Capturing the number of words:
```

```
# Note the conversion to an array:
```

```
$ words=$(wc -w *.txt | tail -n1)
```

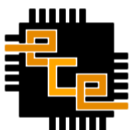
```
echo "There are ${words[0]} in all files."
```



# 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
<code>-d&lt;D&gt;</code>	Specifies the character <code>&lt;D&gt;</code> as the field delimiter. The default field delimiter is a <code>TAB</code> character
<code>-s</code>	Ignore lines that do not contain any delimiter characters
<code>-f&lt;fields&gt;</code>	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)
<code>-c&lt;chars&gt;</code>	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):

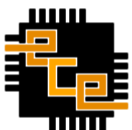
```
$ cut -c 1-3 tabdata
```

```
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):

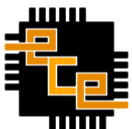
```
$ cut -f2 tabdata
```

```
Mike Goldfarb
```

```
Jacob Wyant
```

```
Jung Yang
```

```
Aarthi Balachander
```



# cut Command (4)

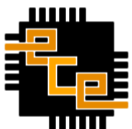
- Assume the file “**tabdata**” contains:

```
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
```

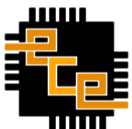
```
001      mgoldfar
002      jwyant
003      yang205
004      abalacha
```



# 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
<code>-d&lt;D&gt;</code>	Specifies the character <code>&lt;D&gt;</code> as the field delimiter. The default field delimiter is a <code>TAB</code> character
<code>-s</code>	Paste files horizontally





## **paste** Command (2)

- Assume the file “accounts” contains

```
ee364a01
```

```
ee364a02
```

- Assume the file “names” contains

```
Michael Goldfarb
```

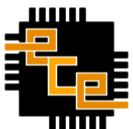
```
Jung Yang
```

- 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
```

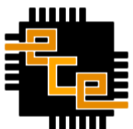
```
Jung Yang
```

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

```
$ paste -s -d',' accounts names
```

```
ee364a01,ee364a02
```

```
Michael Goldfarb,Jung Yang
```



# 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

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



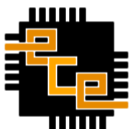
## sort Command (2)

- Consider a file called “data” that contains:

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

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

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



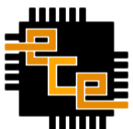
## sort Command (3)

- Consider a file called “data2” that contains:

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

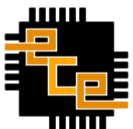
- To sort on column 3 first and then on column 2:

```
$ 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
```

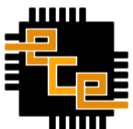
```
2c2
```

```
< 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

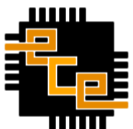
5 6 7 8

```
$ 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**

