

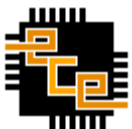


ECE364

Software Engineering Tools Lab

Lecture 1

Bash I



Outline

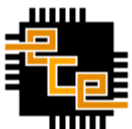
Programming the Bash Shell (Part 1)

- Preliminaries
- Variables
- Math
- Branching
- Script I/O
- Looping
- Simple Text Operations



The Shell

- The shell is a program that interfaces users with the operating system
- The Graphical Shell
 - Point and 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](http://www.gnu.org/software/bash/manual/bashref.html#Shell-Operation))*
 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



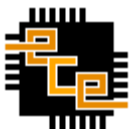
Return Codes

- When a command terminates it returns a numeric value called the **return code**
 - Recall returning an 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>`
 - **Always** end your script with an exit command!



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
```



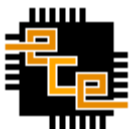
Commands in Bash (2)

- 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 #
- Bash will ignore comments when running commands in your script
 - The # ! line is the exception

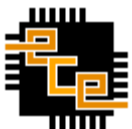


Debugging Bash Scripts

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

Option	Description
<code>-n</code>	Check the script for syntax errors but do not execute any commands
<code>-x</code>	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 than 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!



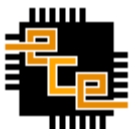
Variables in Bash (2)

- 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

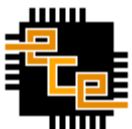
- Set by Bash automatically and can not be assigned directly

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

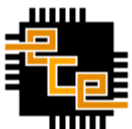


Demo: Basics



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 performed 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 $?
```

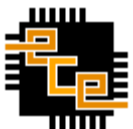
```
[[ -z "Nope" ]]        # is the string empty?
```

```
[[ "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 ]]
```

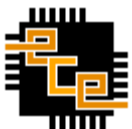


Arithmetic Test Expressions

Expression	Description
<code>x == y</code>	True if x is equal to y
<code>x != y</code>	True if x is not equal to y
<code>x < y</code>	True if x is less than y
<code>x > y</code>	True if x is greater than y
<code>x <= y</code>	True if x is less than or equal to y
<code>x >= y</code>	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
<code>-e <file></code>	True if <file> exists
<code>-f <file></code>	True if <file> is a regular file
<code>-d <file></code>	True if <file> is a directory
<code>-r <file></code>	True if <file> is readable
<code>-w <file></code>	True if <file> is writable
<code>-x <file></code>	True if <file> is executable
<code>-s <file></code>	True if <file> exists and is not empty

Example Usage:

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

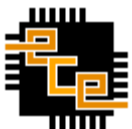


String Test Expressions

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

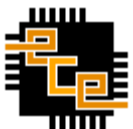
```
if <command/conditional test>  
then  
    <commands>  
elif <command/conditional test>  
then  
    <commands>  
else  
    <commands>  
fi
```



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
```



Demo: Conditional Testing



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

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



Globs (2)

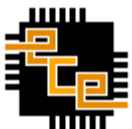
- Globbs are expanded into a list of file names that match the glob

- Examples:

```
ls *.c
```

```
cat ee364*.log
```

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



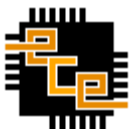
Globs (3)

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

- Example:

```
cat *junk_dsfsfsdfsfsf
```

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



Globs (4)

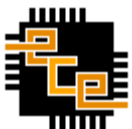
- Brace expansion can be combined with globs to form even more complex patterns

- 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
<code>-n</code>	Disable the automatic newline
<code>-e</code>	Treat <code>\</code> as an escaping character

```
> age=23
```

```
> echo "You are ${age} years old."
```

```
You are 23 years old
```

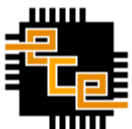
```
> echo "Hello \name"
```

```
Hello \name
```

```
> echo -e "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 than 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



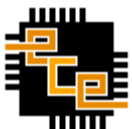
for Command

- 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
```



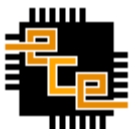
Loop Control Commands

`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
```



Demo: Loops

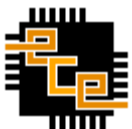


while Command

- 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
```

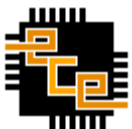


while Command (2)

- 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)

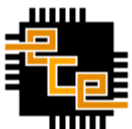
- Like the conditional tests `[[...]]` and `((...))` other command tests can be inverted with a `!` Operator

```
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

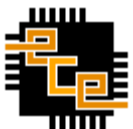


shift Command (2)

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



tail Command

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

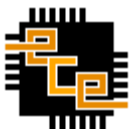
Option	Description
<code>-n <N></code>	Displays the last <code><N></code> lines
<code>-n +<N></code>	Displays all lines starting at line <code><N></code>
<code>-c <N></code>	Displays the last <code><N></code> characters
<code>-c +<N></code>	Displays all characters starting at the <code><N></code> th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



Demo: File Content Printing

