

Linux for Power Users

May 3, 2019

Presented by UWB ACM

uwbacm.com
bit.ly/acm-linux

Sign in at the bit.ly link, and join the Discord via our website!

```
hello world
```

Welcome!

Thank You For Being Here

Are you ready to become a power user?



Assumptions and Prerequisites

- You have attended ACM's Linux Crash Course before AND/OR you have used Linux before (at least a little bit)
- You have access to a Linux installation or a Linux bootable drive
- You have access to the internet on your Linux platform

These assumptions don't apply to you?

That's okay! All of the same commands can be performed on the Linux Lab machines, so please ssh into a machine of your choice if you are able to.

Can't ssh into another machine, or didn't bring a laptop? Feel free to follow along, take notes, and ask questions about the material we will cover today.

What Makes a Power User?

Making Linux Work For You

Don't be Scared

Linux commands are powerful!

Being comfortable with commands, man pages, and good old Google will be immensely helpful for:

- Advanced coursework
- Future employment
- Cloud development

And lots of other things!



Goals for Today's Session

- Get comfortable reading man pages
- Explore an unfamiliar repository in the command line
- Use command options to extract interesting and relevant information
- Use bash operators and simple commands to iteratively build complex commands
- Introduce logical constructs in bash
- Introduce scripting and discuss pros & cons of writing scripts vs using CLI commands

New Content At-A-Glance

We will be covering:

- `/`
- `~`
- `/etc`
- `tree`
- `Find`
- `*`
- `?`
- `>, >>, <`
- `|`
- `head and tail`
- `wc`
- `grep`
- `$?`
- `$HOME`
- `$PATH`
- `if`

A Quick Refresher

Welcome back to Linux!

Let's Boot It!

If you have Linux installed on the hard disk in your laptop, go ahead and boot your machine.

Otherwise, please ssh into one of the machines in the Linux Lab. E.g.:

```
ssh NETID@uw1-320-XX.uwb.edu
```

Use one of the following numbers in place of xx: 00-04, 06-15

Feel free to follow along and road-test the commands during the presentation.

Let's Review Basic Commands

Open the CLI (command line interface) terminal program of your choice, if you booted into a desktop.


All of the commands here should be familiar, but let's do a quick review.

- `pwd`
- `ls`
- `cd <dir>`
- `mkdir`
- `echo`
- `ssh, scp`
- `cat <file>`
- `man <command>`
- `javac, java, g++`
- `which`
- `apt, apt-get`

Basic Commands Defined

Command	What it Does	Example Usage/Output
<code>pwd</code>	“Print Working Directory”: Prints the absolute path of the current working directory you are in.	<code>/home/user</code>
<code>ls</code>	“List Subdirectories”: Lists the files and folders that are stored in your current working directory	<code>Code Downloads hi.cpp Media Schoolwork</code>
<code>cd <dir></code>	“Change Directory”: Changes the current working directory you are in.	<code>cd ..</code> -> move to parent folder <code>cd /</code> -> move to root directory <code>cd Code</code> -> moves to an existing immediate subdirectory named “Code”

Basic Commands Defined (Continued)

Command	What it Does	Example Usage/Output
<code>echo <text></code>	Prints standard input (argument[s]) to standard output (aka the console).	<pre>\$ echo hi hi \$ echo "have a great quarter!" have a great quarter!</pre>
<code>cat <file></code>	"Catalogue": Prints the contents of the specified file to the console.	<pre>\$ cat hello.txt Hello world! \$ cat driver.cpp int main() { return 0; }</pre>
<code>man <command></code>	"Manual Page": shows documentation for specified command. Usually equivalent to <code><command> --help</code>	Try It And See TM 

Basic Commands Defined (Continued)

Command	What it Does	Example Usage/Output
<code>mkdir <name></code>	Creates a new directory in the directory specified as an argument (or defaults to current directory)	<pre>\$ mkdir Code \$ ls Code Documents Pictures</pre>
<code>which <command></code>	Shows output for the executable file if it exists in your path. It is a handy way to check if you have software installed.	<pre>\$ which bash /usr/bin/bash \$ which fortune /usr/bin/fortune</pre>
<code>ssh <machine></code>	SSH (“Secure Shell”) allows you to remotely log into a machine and run commands, be productive, etc.	<pre>\$ ssh lizzy@test.tld Password: ...[output]... lizzy@test.tld:~ \$</pre>

Other Useful Basic Commands

Even if you don't use these commands regularly, they **will** prove very useful today.

- `git <command> <option[s]>`
 - `git` is a version control system, and the specific `git` command is provided in the second word (`<command>`)
 - Options are typically context-dependent based on the `git` command you are running. See the man pages for more information, or search the internet for specific functionality you want to use.
 - Examples:
 - `git add main.cpp` `git add .` `git add --all`
 - `git commit` `git commit -m "My Short Commit Message"`
 - `git clone https://www.github.com/UWB-ACM/Linux_MysteryBox`

Other Useful Basic Commands (Continued)

Even if you don't use these commands regularly, they **will** prove useful today.

Note: this only applies to participants who are NOT ssh'ed into a Linux Lab machine.

- `sudo apt update`
 - This updates the list of accessible software packages in the system.
 - If you need to install software, best practice is to run this before running `apt install`
- `sudo apt install <package_name>`
 - Request the package manager to install the software package specified

The *nix Filesystem

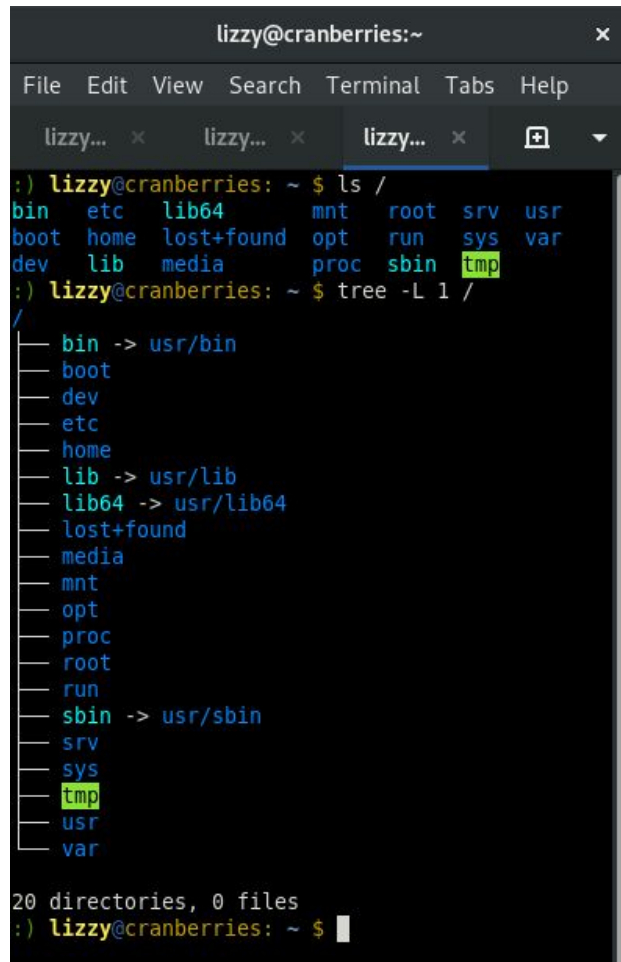
A Quick Tour in Three Parts

First Stop: /

The filesystem in Linux begins at the filesystem root. This is denoted by the forward-slash character `/`.

Run `ls /` in your terminal session to take a look at some of the immediate subfolders.

We will look at a few of the subfolders in greater depth in the next few slides.



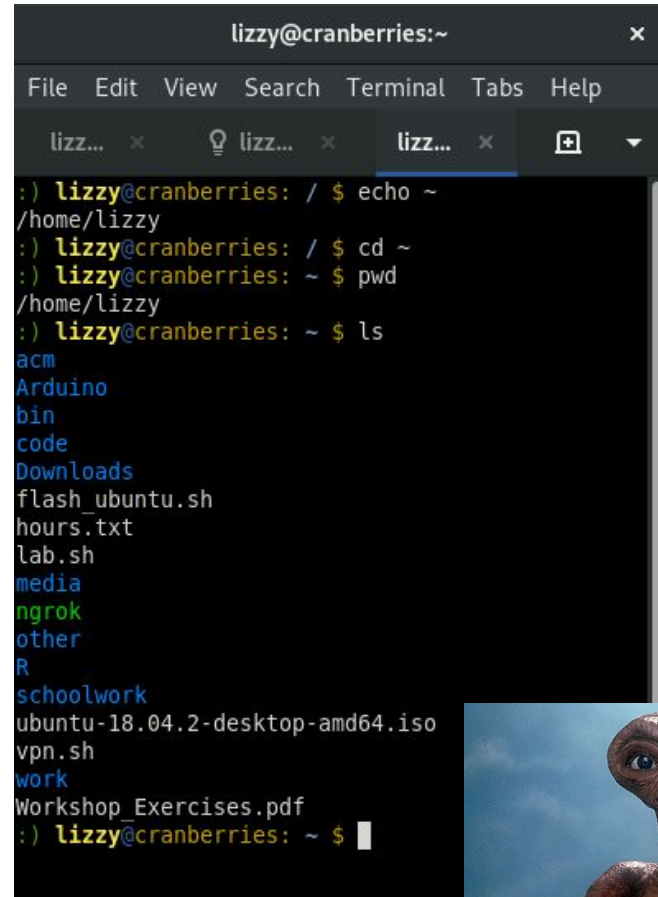
```
lizzy@cranberries:~  
File Edit View Search Terminal Tabs Help  
lizzy... x lizzy... x lizzy... x  
:) lizzy@cranberries: ~ $ ls /  
bin  etc  lib64  mnt  root  srv  usr  
boot home lost+found opt  run  sys  var  
dev  lib  media  proc sbin tmp  
:) lizzy@cranberries: ~ $ tree -L 1 /  
/  
├── bin -> usr/bin  
├── boot  
├── dev  
├── etc  
├── home  
├── lib -> usr/lib  
├── lib64 -> usr/lib64  
├── lost+found  
├── media  
├── mnt  
├── opt  
├── proc  
├── root  
├── run  
├── sbin -> usr/sbin  
├── srv  
├── sys  
├── tmp  
├── usr  
└── var  
  
20 directories, 0 files  
:) lizzy@cranberries: ~ $
```

Second Stop: ~

The tilde character ~ represents your user-space home directory. The path to a user's home directory looks slightly different depending on the machine configuration and user settings.

The home directory is a good place for personal file storage and productivity.

The screenshot on the right shows a few ways of manipulating the tilde shortcut.



```
lizzy@cranberries:~  
File Edit View Search Terminal Tabs Help  
lizz... x lizz... x lizz... x  
:) lizzy@cranberries: / $ echo ~  
/home/lizzy  
:) lizzy@cranberries: / $ cd ~  
:) lizzy@cranberries: ~ $ pwd  
/home/lizzy  
:) lizzy@cranberries: ~ $ ls  
acm  
Arduino  
bin  
code  
Downloads  
flash_ubuntu.sh  
hours.txt  
lab.sh  
media  
ngrok  
other  
R  
schoolwork  
ubuntu-18.04.2-desktop-amd64.iso  
vpn.sh  
work  
Workshop_Exercises.pdf  
:) lizzy@cranberries: ~ $
```



Third Stop: `/etc`

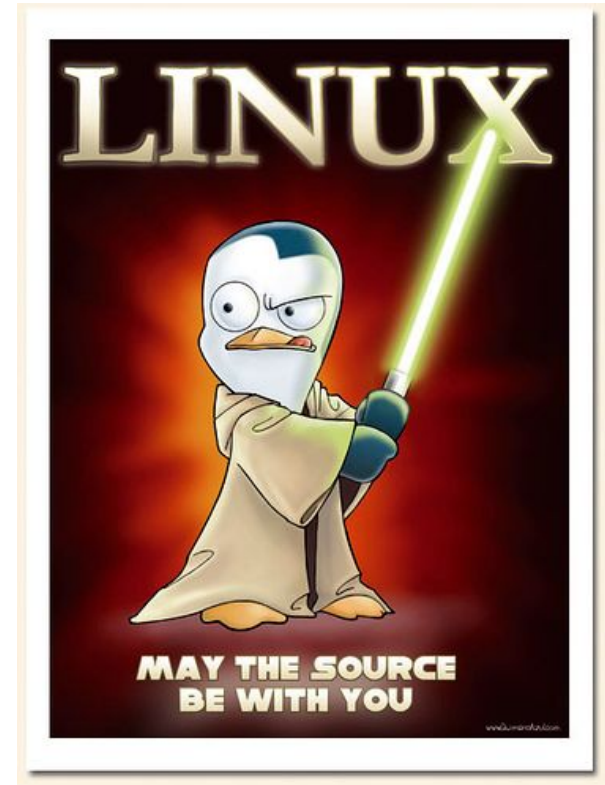
Usually “et cetera” implies insignificance.

In Unix, this folder is super important!

`/etc` can be thought of as “everything to configure”.

The files and folders tell Unix machines how to function, render information, serve content over the web, respect user- and system-level settings, and tons of other things.

Don't delete this folder :)



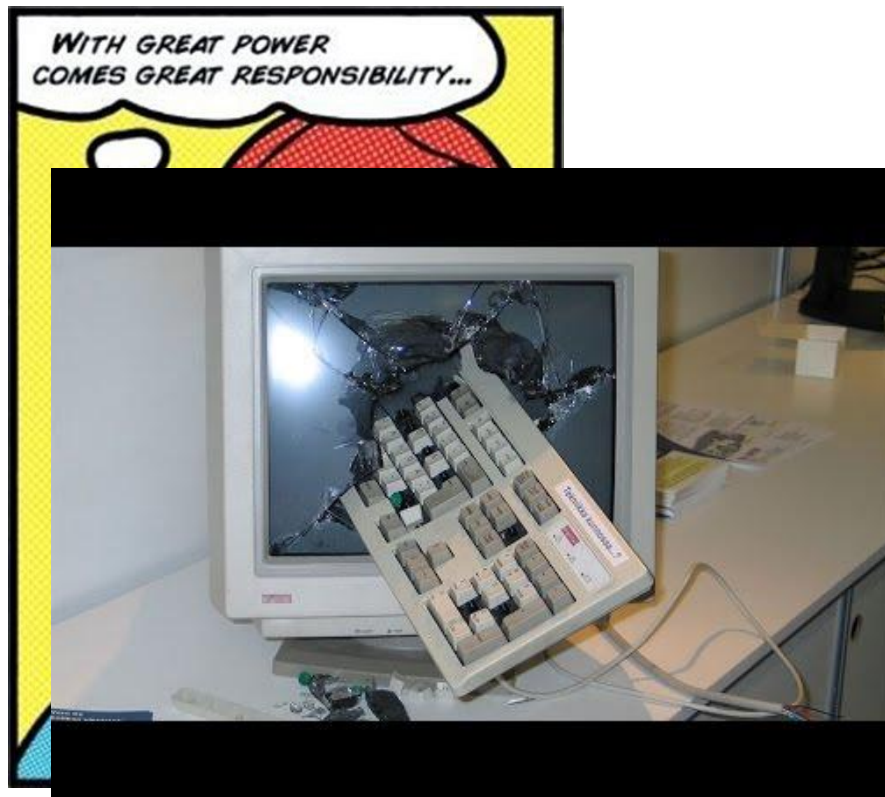
Go Forth And Explore

Feel free to explore your filesystem.

Use `cd`, `ls`, and new commands to learn more about your system.

Obligatory word of warning: be careful of modifying or removing files.

Remember that `rm` cannot be reversed (unless you're using `git`)!



New Commands!

Part I: Directory Exploration

tree

What does it do?

`tree` prints a nicely-formatted, alphabetically sorted recursive list of all files and folders in the current directory.

How to use it?

Command	What It Does
<code>tree</code>	Prints file structure for current directory
<code>tree <directory></code>	Prints file structure for the specified directory

find

What does it do?

`find` is a command that searches for a file or directory that matches specific criteria.

How to use it?

Command	What It Does
<code>find .</code>	Recursively prints all files and folders in the existing directory (similar to <code>tree</code> , but without nice formatting)

find (Continued): Options

It is usually very helpful to narrow your search for files based on criteria. `find` has tons of flags (options) to help you!

Option	What It Does	Example Usage
<code>-name <pattern></code> <code>-iname <pattern></code>	<code>-name</code> performs a case-sensitive search for items that match the specified pattern. <code>-iname</code> is case-insensitive.	<code>find . -name "*.cpp"</code>
<code>-type <type></code>	Searches for items that are of the specified type. <code>f</code> is a regular file, and <code>d</code> is directory. Other useful types include sockets, named pipes, and symlinks.	<code>find . -type f</code> <code>find /etc -type d</code>
<code>-mindepth <int></code> <code>-maxdepth <int></code>	Specifies how far into a subdirectory structure to search. <code>Maxdepth</code> searches <code><int></code> levels below starting directory before stopping; <code>mindepth</code> searches all levels below <code><int></code> directory levels.	<code>find . -maxdepth 2</code> <code>find / -mindepth 15</code>

New Commands!

Part II: File Exploration, Redirects, and Wildcards

Wildcards

What does it do?

Wildcards allow you to find text without knowing exactly what the text is, or to locate multiple strings that contain similar criteria.

How to use it?

Token	What It Does	Example Usage
*	“Shell Glob”: The asterisk symbol is colloquially known as a glob, and it matches 0 or more characters in any order.	<pre>\$ ls *.cpp driver.cpp fib.cpp test.cpp</pre>
?	Matches any single character in a sequence.	<pre>\$ ls test?.txt test1.txt test@.txt testN.txt</pre>

Redirects

What does it do?

Redirect operators write text to files and read text from files.

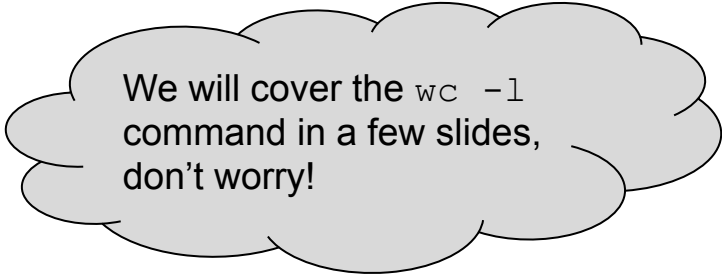
How to use it?

Operator	What It Does	Example Usage
>	Writes standard output from the LHS command to a file named on RHS. Overwrites RHS file if it exists.	<pre>\$ echo "Testing 123" > test.txt \$ cat test.txt Testing 123</pre>
>>	Same as >, but instead of overwriting file contents (if the file already exists), it appends the LHS string to the end of the file.	<pre>\$ echo "Again" >> test.txt \$ cat test.txt Testing 123 Again</pre>

Redirects (Continued)

What about reading text from files?

Operator	What It Does	Example Usage
<	Reads contents from RHS file and feeds it into the program called in the LHS argument.	<pre>\$ cat < test.txt Testing 123 Again \$ wc -l < test.txt 2</pre>



We will cover the `wc -l` command in a few slides, don't worry!

Redirects (Continued)

Some interesting tidbits:

- There are 3 (three) kinds of I/O streams in all Linux programs & commands:
 - Standard input
 - Standard output
 - Standard error
- Each stream has its own numerical code: 0, 1, and 2 respectively
- Redirect operators only stream standard output to a file by default.
- To redirect standard error streams to the file (or another stream), you need special syntax.
 - The syntax `<stream> &> test.txt` writes all streams to the file `test.txt`
 - To learn more about that, check out the resource links at the end of this presentation.

Piping

What does it do?

Piping (using the pipe character “|”) allows you to feed program output into another program *without writing it to a file*.

How to use it?

```
chris@ubercomputer ~ $ fortune | cowsay
/ If more of us valued food and cheer and \
| song above hoarded gold, it would be a |
| merrier world.                          |
\ -- J.R.R. Tolkien                       /
-----
      ^ ^
      (oo)\_____)
      ( )    /)\
      ||----w |
      ||     ||
```

Token	What It Does	Example Usage
	Redirects standard output from the LHS command to the RHS command/program.	<pre>\$ pwd find -name *.cpp ./code/main.cpp [...etc...] \$ cat test.txt wc -w 3</pre>

head and tail

What does it do?

`head` prints the first 10 lines of the specified file; `tail` prints the last 10 lines.

How to use it?

Command	What It Does	Example Usage
<code>head <filename></code>	Prints the first 10 lines of file.	<pre>\$ head test.txt Testing 123 Again</pre>
<code>tail <filename></code>	Prints the last 10 lines of file.	<pre>\$ tail test.txt Testing 123 Again</pre>

head and tail (Continued): Options

Another useful option for `head` and `tail` lets you view a specified number of lines you want to see.

Option	What It Does	Example Usage
<code>-n <int></code>	Prints the first/last <int> lines of file.	<pre>\$ head test.txt -n 1 Testing 123 \$ tail test.txt -n 1 Again</pre>
<code>-f</code>	Continues to monitor the file for changes and writes new content to the console	<pre>\$ tail test.txt -f Testing 123 Again Wow!!!</pre>

WC

What does it do?

`wc` provides word, line, and byte counts for text and files.

How to use it?

Command	What It Does	Example Usage
<code>wc <file></code>	Prints all 3 values for the text in the specified file.	<pre>\$ wc test.txt 2 3 18 test.txt</pre>
<code><stdin> wc</code>	Takes all piped content and runs <code>wc</code> on content	<pre>\$ cat test.txt wc 2 3 18</pre>

WC (Continued)

wc has useful options as well!

Option	What It Does	Example Usage
-l	Prints line count for text content	<pre>\$ wc -l test.txt 2 test.txt</pre>
-w	Prints word count for text content	<pre>\$ wc -w test.txt 3 test.txt</pre>
-L	Prints the maximum line length in the specified content	<pre>\$ wc -L test.txt 11 test.txt</pre>

grep

What does it do?

grep is a very powerful text analysis tool. Its most common use is to locate text in a file or set of files that matches specific criteria.

How to use it?

Command	What It Does	Example Usage
<code>grep <pattern> <file></code>	Searches for <pattern> in <file>	<pre>\$ grep main test.cpp int main() {</pre>
<code>grep <pattern></code>	Searches for <pattern> in standard input	<pre>\$ cat test.cpp grep main ./test.cpp:int main() { ./patterns.txt:main</pre>

grep (Continued): Options

grep is a richly featured command. (Note: knowledge of regex can be extremely useful for taking full advantage of grep)

Command/option	What It Does	Example Usage
<code>grep -R <pattern> <dir></code>	Recursively searches each file in <dir> for <pattern>	<pre>\$ grep -R hi . ./test.txt:hi ./folder/findme.txt:hi</pre>
<code>-f <file></code>	Instead of specifying a pattern in the terminal, read desired patterns to match from file (one per line)	<pre>\$ grep -R -f patterns.txt . ./test.cpp:int main() { ./patterns.txt:main</pre>
<code>-n</code>	Print line number of pattern matches	<pre>\$ grep -R -n main test.cpp:5:int main() { patterns.txt:1:main</pre>

grep

“grep is the most useful command you can learn in this workshop. It is also difficult to master usage of grep. Keep trying things, get familiar with basic regex, and make friends with the man page.”

-- Lizzy's opinion

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.

OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!



BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!

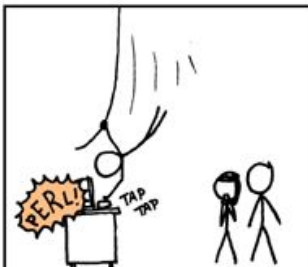


IT'S HOPELESS!

EVERYBODY STAND BACK.



I KNOW REGULAR EXPRESSIONS.



New Commands!

Part III: Environment Variables

\$: The Dollar Sign

Like most programming languages, the Bash shell has special symbols and keywords that represent:

- Information about the environment (system)
- The command-line session
- User-defined values (“variables”)

We will focus on **environment variables** today.

Environment variables can be accessed by prefixing the variable name with \$.

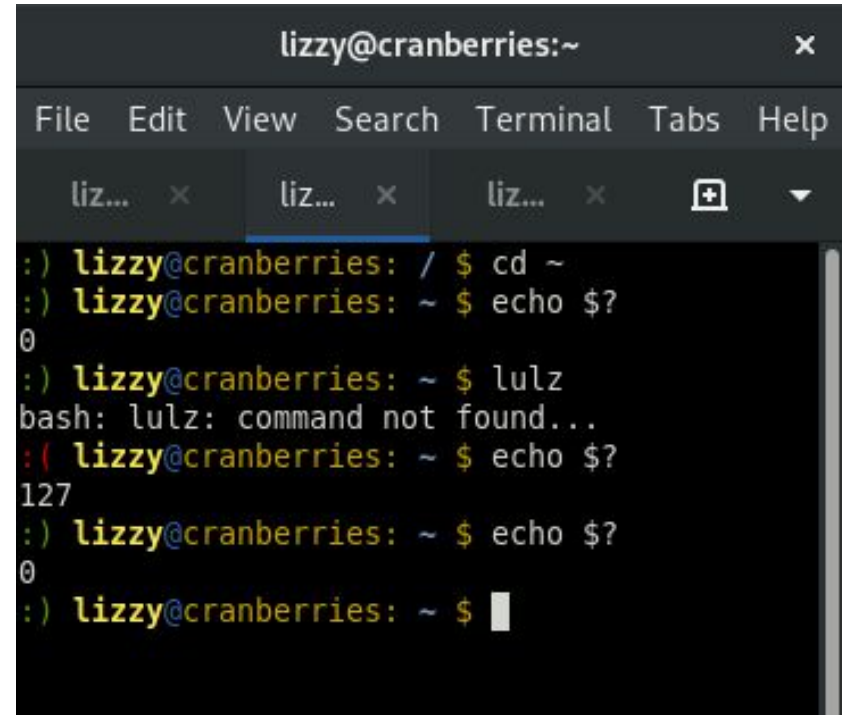


\$?

All bash commands return with an exit code, which tells us whether the command was successful.

An exit code of 0 indicates the command succeeded.

This environment variable `$?` tells us what the exit code for the last executed command is.

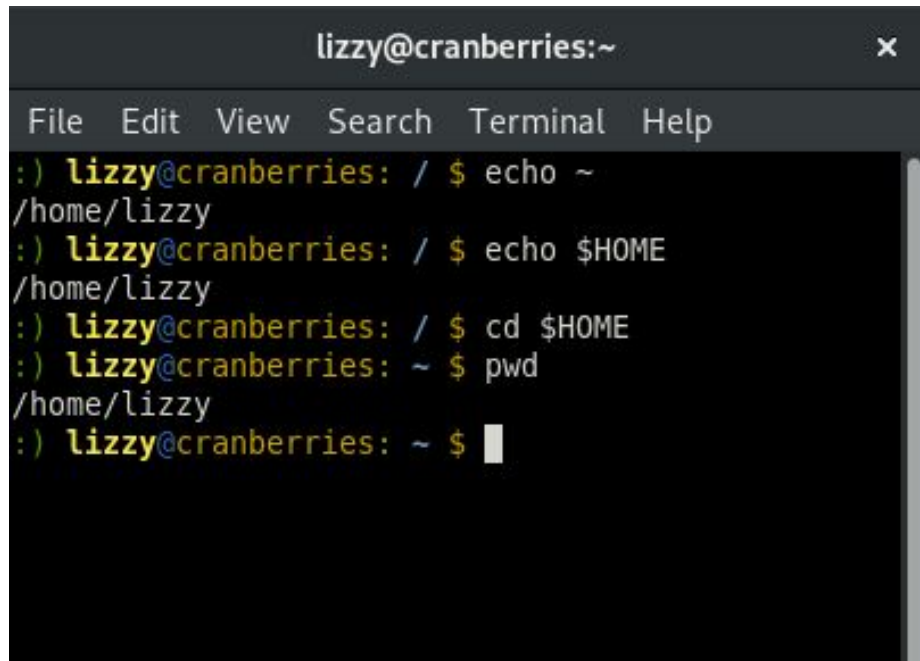


```
lizzy@cranberries:~  
File Edit View Search Terminal Tabs Help  
liz... x liz... x liz... x  
:) lizzy@cranberries: / $ cd ~  
:) lizzy@cranberries: ~ $ echo $?  
0  
:) lizzy@cranberries: ~ $ lulz  
bash: lulz: command not found...  
:( lizzy@cranberries: ~ $ echo $?  
127  
:) lizzy@cranberries: ~ $ echo $?  
0  
:) lizzy@cranberries: ~ $
```

\$HOME

This directory is the same as ~ that we saw before.

It is our home directory, which is intended for user-space productivity.



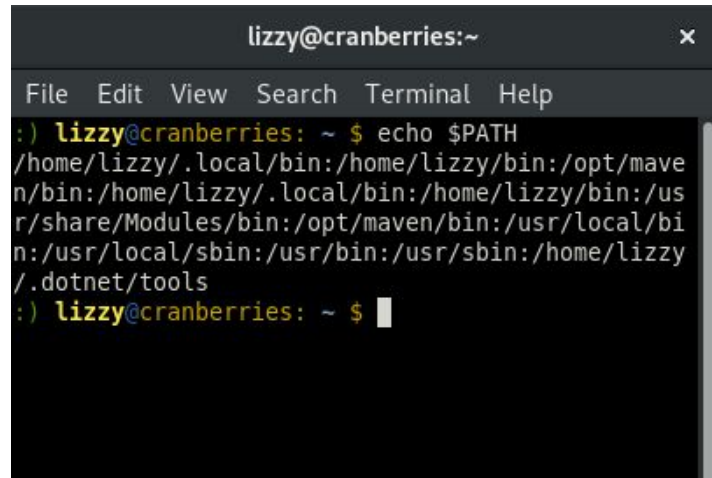
```
lizzy@cranberries:~  
File Edit View Search Terminal Help  
:) lizzy@cranberries: / $ echo ~  
/home/lizzy  
:) lizzy@cranberries: / $ echo $HOME  
/home/lizzy  
:) lizzy@cranberries: / $ cd $HOME  
:) lizzy@cranberries: ~ $ pwd  
/home/lizzy  
:) lizzy@cranberries: ~ $
```

\$PATH

All of the commands on our machine actually refer to an executable file somewhere in our filesystem.

So, how does bash know where to look for the executable file?

That responsibility belongs to the `$PATH` environment variable. It has a system filepath to all folders that contain commands we are looking for. Each separate path is delimited by colons.

A terminal window titled 'lizzy@cranberries:~' with a menu bar (File, Edit, View, Search, Terminal, Help). The terminal shows the command 'echo \$PATH' being executed, resulting in a long string of directory paths separated by colons. The paths include local user bins, system binaries like maven, and standard system paths like /usr/bin and /usr/sbin.

```
lizzy@cranberries: ~ $ echo $PATH
/home/lizzy/.local/bin:/home/lizzy/bin:/opt/maven/bin:/home/lizzy/.local/bin:/home/lizzy/bin:/usr/share/Modules/bin:/opt/maven/bin:/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin:/home/lizzy/.dotnet/tools
lizzy@cranberries: ~ $
```

New Commands!

Part IV: Bash Conditionals

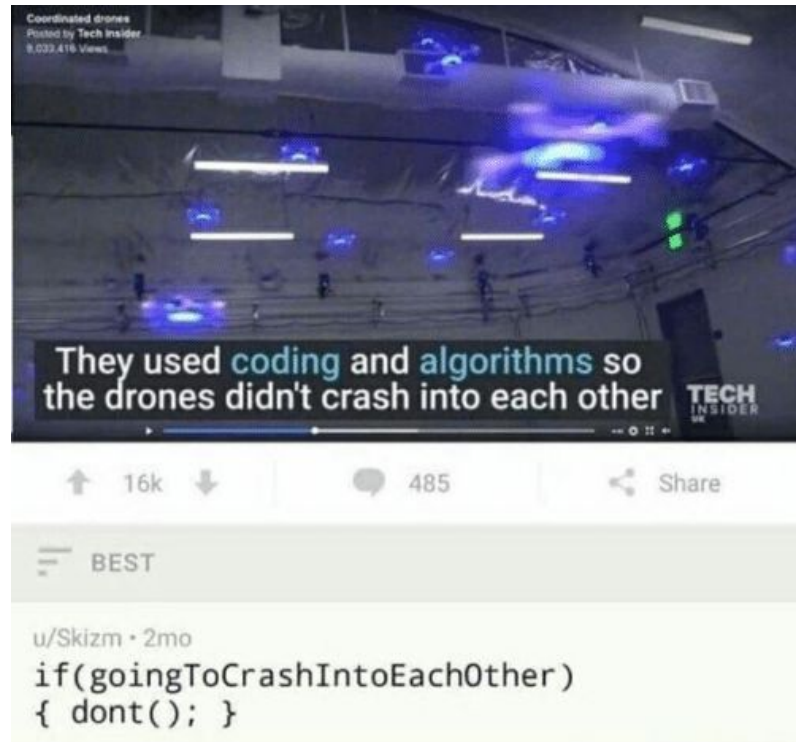
Bash Is Logical, Too

As programmers, we are used to using logic to conditionally run blocks of code.

We like `if` statements, `for` loops, `while` loops, `case` statements, and other types of logical constructs.

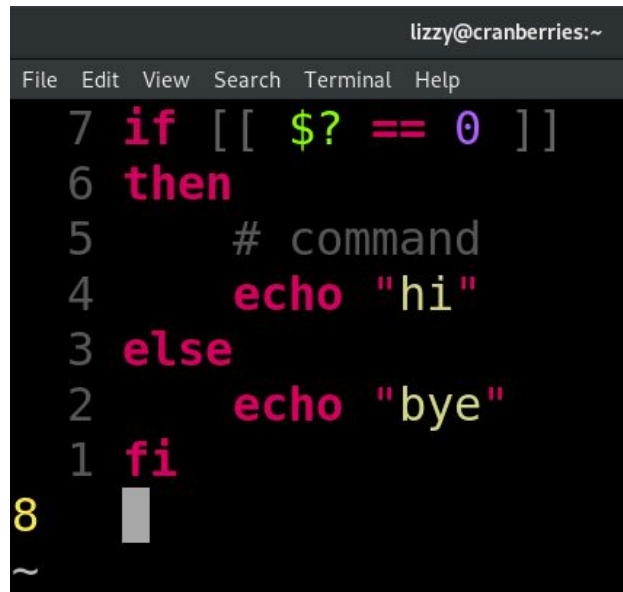
Bash has its own implementation of all of these statements!

We will only look at the `if` statement today.



Anatomy of Bash's `if`

Bash syntax	What it does	Java equivalent
<code>if</code>	Begins a logical if block in the script/command	<code>if</code>
<code>[[test]]</code>	The double square brackets <code>[[]]</code> denote a logical test to determine which subsequent command block to execute	<code>(test)</code>
<code>then</code>	Begin block of commands to execute if test is true	<code>{</code>
<code>else</code>	Begin block of commands to execute if test is false	<code>} else {</code>
<code>fi</code>	End logical block	<code>}</code>



The screenshot shows a terminal window with the title `lizzy@cranberries:~`. The menu bar includes `File`, `Edit`, `View`, `Search`, `Terminal`, and `Help`. The terminal content is a Bash script with line numbers 1 through 8 on the left. The script is as follows:

```
1  if [[ $? == 0 ]]
2  then
3      # command
4      echo "hi"
5  else
6      echo "bye"
7  fi
8  ~
```

Bash Conditional Tests

Now that we know a little bit about the `if` statement's syntax, we need syntax for running tests inside the double square brackets: `[[test]]`

Bash conditional test	What it does	Java equivalent	Bash example
<code>-e <file></code>	Tests if the specified filepath currently exists	<pre>import java.io.File; File f = new File(path); if(f.exists()) { }</pre>	<pre>if [[-e ~/.bashrc]] then echo "exists" fi</pre>
<code>-v <var></code>	Tests if the specified variable exists and has been assigned a value	<pre>// note: comparable // Java test doesn't // exist if (myVar != null) { }</pre>	<pre>if [[-v \$TODO]] then echo "agenda:" fi</pre>

Bash Conditional Tests: Continued

Bash conditional test	What it does	Java equivalent	Bash example
<code>s1 == s2</code> <code>s1 != s2</code> <code>s1 < s2</code> <code>s1 > s2</code>	Tests if strings <code>s1</code> and <code>s2</code> are lexicographically equal, unequal, greater than, less than, etc.	<code>if (s1.equals(s2)) { }</code> <code>// ...etc</code>	<code>if [[~ == \$HOME]]</code> <code>then</code> <code> echo "hi Toto"</code> <code>fi</code>
<code>-z <str></code>	Tests if the specified string has a length of 0	<code>if (s.length() == 0) { }</code>	<code>if [[-z \$PATH]]</code> <code>then</code> <code> echo "no path!"</code> <code>fi</code>
<code>-n <str></code>	Similar to <code>-z</code> , but tests if specified string's length is greater than 0	<code>If (s.length() > 0) { }</code>	<code>if [[-n \$PATH]]</code> <code>then echo \$PATH</code> <code>fi</code>

Other Neat Things To Look Into

It is recommended to experiment with other logical constructs and syntax variations. Here's a few ideas to get you started:

- Logical blocks can be concatenated into a single line. Each block segment (`if then-cmd1 cmd2 else-cmd3 end`) needs to be delimited with a semicolon `;`. For example, in the command line interface, we could say:

```
if [[ ~ == $HOME ]]; then cd ~; echo "hi Toto"; else echo  
"not in Kansas"; fi
```

- Look into the following logical constructs:

<code>do while</code>	<code>for do done</code>	<code>case esac</code>
-----------------------	--------------------------	------------------------

New Commands!

Part V: Scripting in Bash

What is a bash script?

A bash script is a file with a sequence of commands to run in a shell environment.

Instead of typing each command one-by-one each time you want to perform a set of tasks, you can create a bash script to run all of the commands in sequence.

Bash scripts are also great for automating tasks, like launching specific webpages, setting your terminal prompt, running specific programs on login or logout, and many more!



Example 1: Compile & Execute

Script	What is it doing?
<pre>#!/bin/bash # compile MyClass + peripherals javac *.java if [[\$? != 0]] then echo "compilation unsuccessful" else java MyClass fi</pre>	<p>Declares bash path for execution (shebang)</p> <p>Comment! Compile all .java files in directory</p> <p>Begin logical block; check exit code</p> <p>Exit was not 0; don't try to run program</p> <p>Run program! End logical block</p>

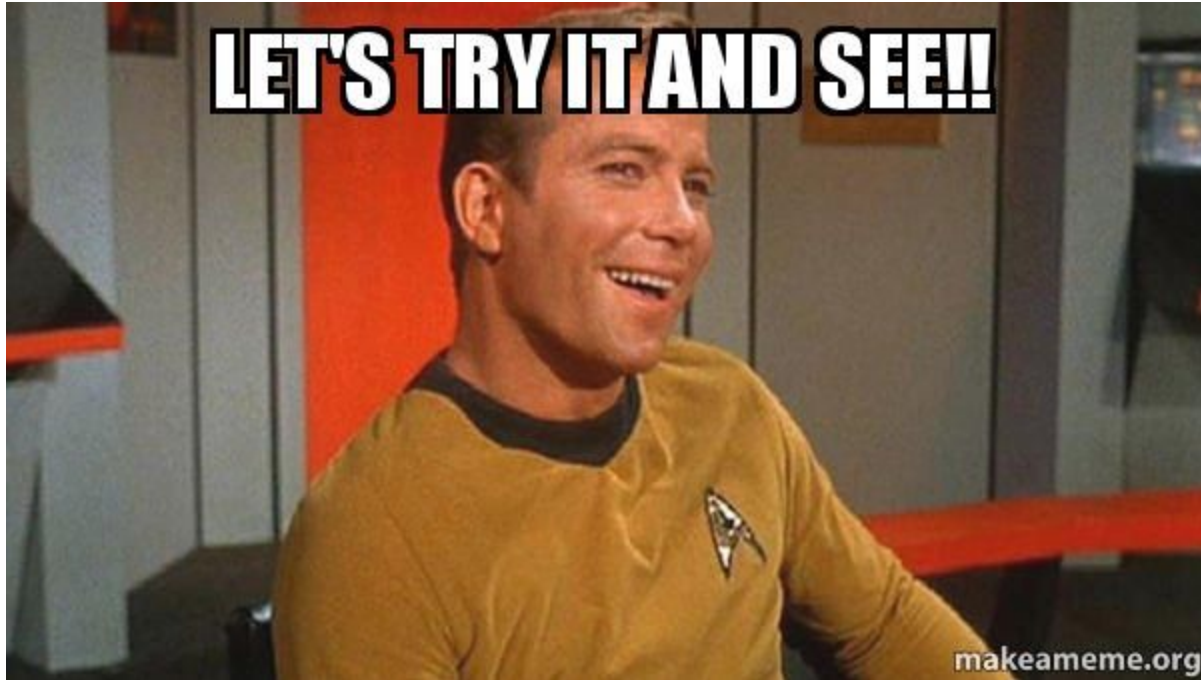
Example 2: Compile & Execute Again

Script	What is it doing?
<pre>#!/bin/bash # compile g++ -o test *.cpp if [[\$? != 0]]; then echo "compilation unsuccessful" exit 1 fi valgrind ./test &> out.txt cat out.txt grep "in use at exit" rm out.txt test</pre>	<p>Declares bash path for execution (shebang)</p> <p>Comment! Compile all .cpp files in directory</p> <p>Begin logical block; check exit code Exit was not 0; don't try to run program</p> <p>End logical block</p> <p>Run valgrind on your program, save to file Print the valgrind "in use at exit" line to standard output (hopefully 0 :)) Delete the files the script generated</p>

Practice Makes Perfect!

Now with demos!

The Best Way To Learn Is To Make Mistakes



Resources & Links

Resources For Today's Material

- This slide deck is available in https://github.com/UWB-ACM/Linux_MysteryBox and <https://github.com/UWB-ACM/Linux-Crash-Course>
- Man pages. Say it again. Man pages. (Stack Overflow is also a good resource)
- More information about piping and redirects, along with great examples: <https://ryanstutorials.net/linuxtutorial/piping.php>
- GNU has a great online manual discussing Bash logical constructs in depth: https://www.gnu.org/software/bash/manual/html_node/Bash-Conditional-Expressions.html
- Streams & file descriptors: <https://www.linux.com/blog/learn/2019/2/ampersands-and-file-descriptors-bash>

Resources For Today's Material (Continued)

- A brief tour of the Linux filesystem:
<https://www.linux.com/blog/learn/intro-to-linux/2018/4/linux-filesystem-explained>
- Environment variables: <https://linuxhint.com/bash-environment-variables/>
- Script shebang: [https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))
- A nice video about the history of grep and how it works:
<https://www.youtube.com/watch?v=NTfOnGZUZDk>
- For humor: XKCD
- More information about subshell processes and variable + env var scope between shells: [https://bash.cyberciti.biz/guide/What is a Subshell%3F](https://bash.cyberciti.biz/guide/What_is_a_Subshell%3F)

Recommended Next Steps

- Revel in your newfound powers! Be amazing!
- Learn git, and learn it well. Git gud! See <https://try.github.io/> and so many other websites & tutorials for resources
- Learn about Linux distributions and GUIs at <https://distrowatch.com/>
- Use `while read line` with piping/redirect operations for great justice (and line-by-line processing magic). Read more here:
<https://www.cyberciti.biz/faq/unix-howto-read-line-by-line-from-file/>
- Learn how to loop over a numerical range with `for` (see <https://ryantutorials.net/bash-scripting-tutorial/bash-loops.php> for this and other cool logical constructs)

Recommended Next Steps (Continued)

- Familiarize yourself with a few simple regex tricks to use with `grep` (such as matching several terms in the same command)
- Experiment with `sed` and `awk` (regex is very important here)
- Get familiar with basic usage in `vim` or `emacs`, and become your own superhero!
- Learn about permissions and access modification with `chown`, `chmod` et al
- Subshell processing with `$ (command [s])`
- Arithmetic operations
- Command substitution with backticks

...And so, so much more!~

Questions?

Thank you for listening!

Contact ACM if you want extra help!