Introduction to Operating Systems I

Benjamin Brewster

Tools versus Theory

- C++? Java? *nix? Apple? You're CS majors, not *nix majors!

- After this class, you should be able to hold an intelligent conversation about any operating system by studying a model OS like UNIX

*NIX

- Why *nix?
  - Stable: good luck crashing it
  - Powerful: dense commands
  - Standard: used everywhere

- Worldwide Device Shipments in 2015 (smartphones, tablets, laptops and PCs)
  - Android (Linux): 1.3 billion
  - Windows: 283 million
  - iOS (UNIX): 276 million
  - OSX (UNIX): 21 million
  - Others (Some Linux): 550 million

NIX is 82.3% of non-Other OSs shipped

Source: Gartner, 4/2016

What is an Operating System?

- A software program that sits between software applications and the computational hardware


```
public void processData()
{
  do
  {
    int data = getData();
    if(data < 0)
      performOperation1(data);
    else
      performOperation2(data);
  }
  while(hasMoreData());
}
```

Why are OSs Important?

- Most applications interact with the OS in some fashion

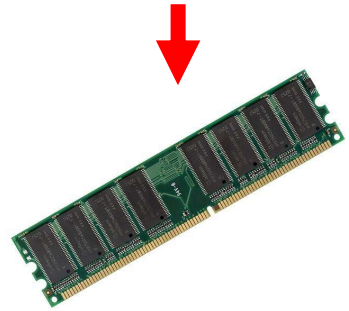
- As a programmer, you will need to:
  - Use the capabilities of the OS to do most anything
  - Be aware of the policies and limitations of the OS

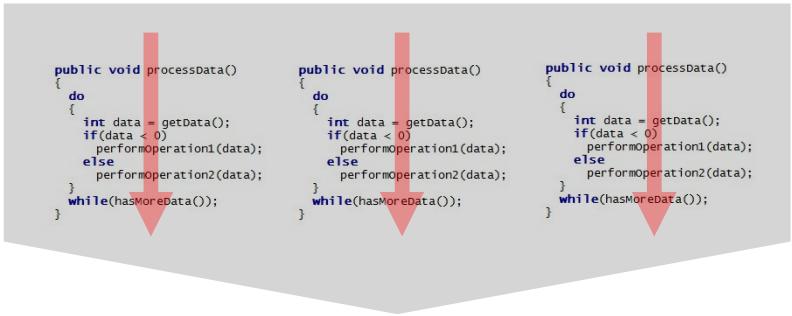
Goals of an Operating System

- Universal Goals
  - Provide convenient software interface to hardware resources
  - Maximize utilization of hardware
  - Solve contention
  - Provide services
- Common Goals
  - Provide security
    - Protect against other buggy applications/crashes
    - Control access to your data by others
  - Support software development
  - Provide standardized software libraries
    - Including a standardized user interface

Definitions

- Program
  - A stored algorithm or plan of execution
- Process
  - A program that has been loaded into memory and is executing
- Thread
  - A line of execution in a process





Standard OS Services

1. Process and thread management
   - Starting a new program (becomes a process & thread)
   - Ending a process/thread
   - Debugging programs/processes

Standard OS Services

1. Process and thread management
   - Starting a new program (becomes a process & thread)
   - Ending a process/thread
   - Debugging programs/processes

2. File and input/output management
   - Organizing bits into meaningful structures: Files
   - Providing interfaces for reading and writing to files
   - Communicating with external devices
   - Organizing files: Directories

Standard OS Services

3. Interprocess communication (IPC)
   - Signals, pipes, network sockets (TCP/IP)
   - Including between two different computers

Standard OS Services

3. Interprocess communication (IPC)
   - Signals, pipes, network sockets (TCP/IP)
   - Including between two different computers

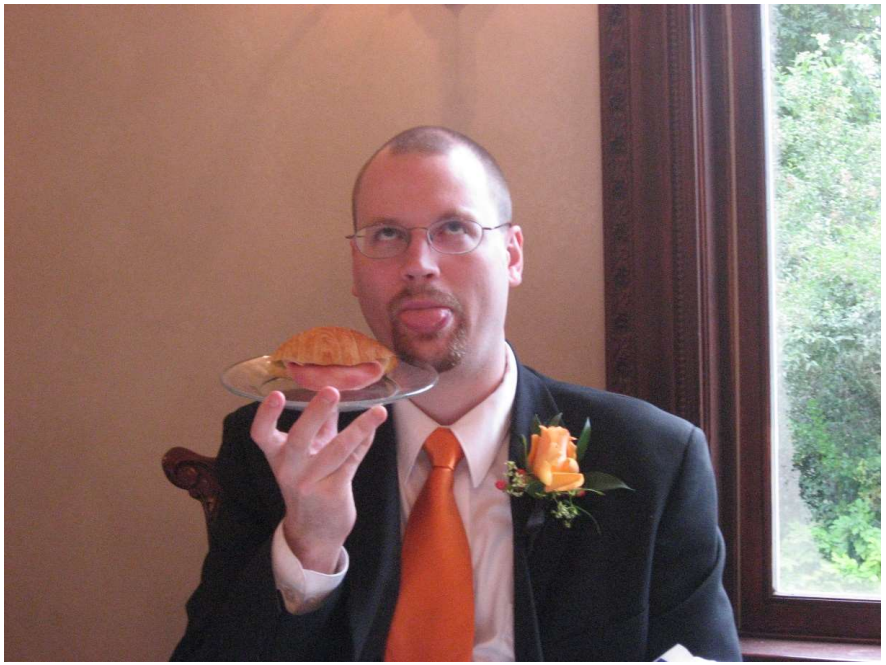
4. Process coordination
   - Contention management leads to shared access

Interacting With the OS

- Users
  - via Graphical User Interface (GUI)
  - via Command Line Shell (|-|4><0|2$)
- Programs
  - via Functions
    - System calls
    - Application Programming Interface (API) - Functions
  - via Network communication
    - Message-based
    - Connection-based

Enjoy!

- If you're not having fun, you're (probably) doing it wrong.



Introduction to UNIX

Benjamin Brewster

UNIX - 1965

- AT&T Bell Laboratories, MIT and General Electric develop a new time-sharing OS called Multics (MULTiplexed Information and Computer Services), which was the first OS to provide a hierarchical file system.

MIT development led by Fernando J. Corbató

UNIX - 1969

Image by user Toresbe, CC SA 1.0, via Wikimedia Commons

PDP-7 under restoration

- Bell Labs management pulls out of the Multics project due to delays, but Bell Labs researchers continue to experiment on their own.
- Bell Labs' Ken Thompson, Dennis Ritchie, Rudd Canaday, and Doug McIlroy create UNIX on a PDP-7.
- While Multics was a large project; UNIX was just a few researchers (carrying on UNiplexed Information and Computer Services).

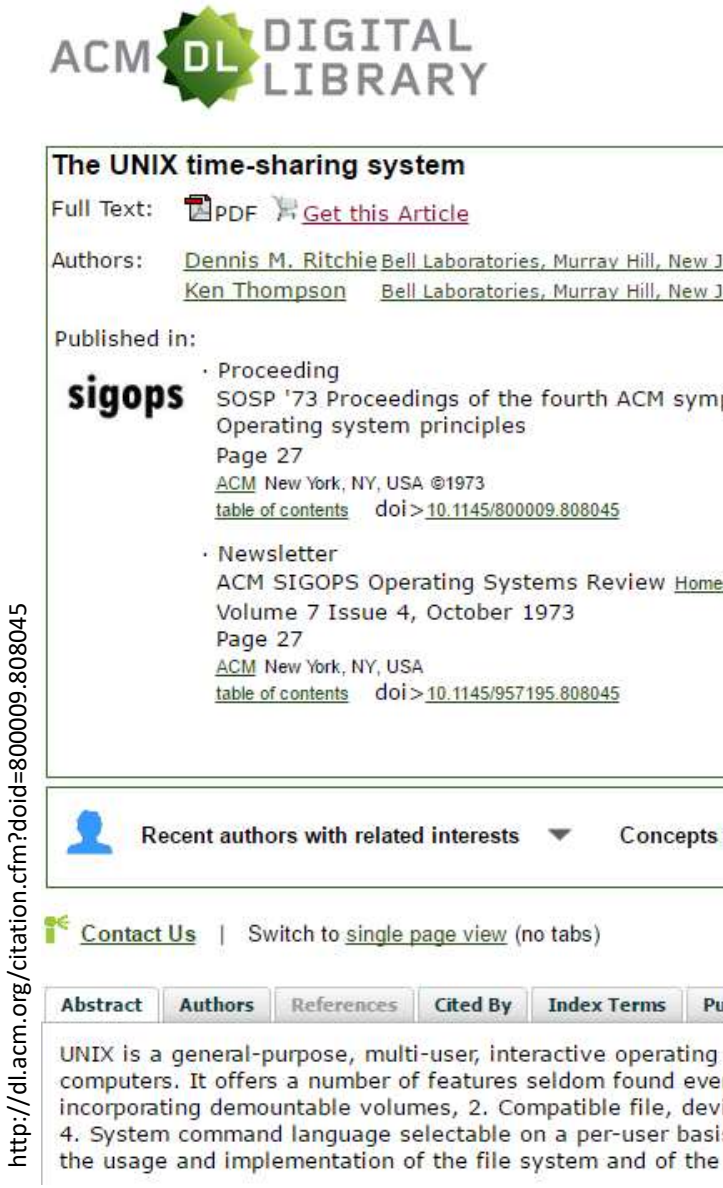
UNIX - 1970s

- January 1, 1970 becomes “time zero” for UNIX.
- Text processing abilities added, allowing UNIX to start being used for actual work by another department at Bell Labs.
- UNIX now runs on a PDP-11/20.

Ken Thompson (sitting) & Dennis Ritchie at PDP-11

UNIX - 1970s

- First manual produced as online “man pages” On November 3, 1971.
- UNIX re-written in C (created by Dennis Ritchie) in 1972, spurring further development in the C language itself.
- Presented to the outside world at the 1973 Symposium on Operating Systems Principles, but could not be turned into a product because of anti-trust regulations.



Linux

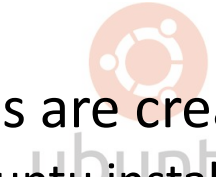
- Initially developed by Linus Torvalds to be a UNIX-like system, but not beholden to expensive license agreements with AT&T.
- Open source, protected by the GNU Public License (GPL)
- Supports the POSIX UNIX specification
  - Code written for another POSIX-based UNIX (ie Solaris, HPUX, AIX, etc) shouldn't need many changes to run on Linux.
  - Linux knowledge will apply to most UNIX systems
- Stable, robust, free
- First version released March 14, 1994.

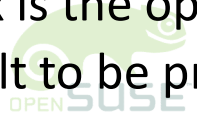
Linus Torvalds speaking at LinuxCon Europe 2014

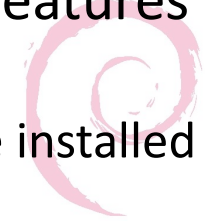
You Can Run Linux at Home

- There are many Linux distributions
  - Ubuntu, openSUSE, Red Hat, Debian, Arch Linux, Chrome OS, Mint, etc.
  - All can be downloaded; some can be purchased
  - EECS at OSU has several Linux machines you can connect to

- Distros are created with different purposes and features in mind
  - Ubuntu installs everything easily
  - Arch Linux is the opposite, requiring everything to be installed manually
  - Tails is built to be private from the ground up







What Is The Shell?

- The user interface to the operating system - a text-based, command line interpreter

  ```
  $ □
  ```

- Provides access to all UNIX user-level programs
  - Start programs
  - Manage processes (job control)
  - Connect together processes (pipes)
  - Manage I/O to and from processes (redirection)
  - Kill programs

Shell Prompts

- Traditional prompt and cursor – your commands go *after* the prompt:

```
$ □
```

- Can be customized:

```
[1459][brewsteb@flip1:~]□
brewsteb@flip1@12:42:17$ □
~/CS344/prog1$ □
```

- A great place to create prompt code: http://ezprompt.net/
- For example:

```
export PS1="\u@\h@\T\\$ "
```

- Place custom prompts with this line in ~/.bashrc

Shell Examples

- Type in a command after the prompt ($), and bash executes it with the output:


```
$ ls
CS344   CS419   CS372
```


```
$ ls -a
.    ..    .bashrc    CS344    CS419    CS372
```

Different UNIX Shells

- Basic Shells
  - Bourne Shell (/bin/sh)
  - C-Shell (/bin/csh)

- Enhanced Shells
  - BASH “Bourne-again shell” (/usr/local/bin/bash)
  - TCSH enhanced C-Shell (/usr/local/bin/tcsh)
  - Korn Shell (/bin/ksh)

bash *was written by Brian Fox as an upgrade to Stephen Bourne’s shell “*sh*”, which itself was a replacement for Ken Thompson’s “*sh*” shell, which was based on the “*sh*” shell from Multics.*

Most Common UNIX Commands

- Directory/file management
  - cd, pwd, ls, mkdir, rmdir, mv, cp, rm, ln, chmod
- File viewing and selecting
  - cat, more/less, head, tail, grep, cut
- Editors
  - vi, (x)emacs, pico, textedit
- Other useful commands
  - script, find, telnet, ssh, and many more!

Common UNIX Commands - Directories & Files

- **pwd** - Print working directory. The “where am I” command; similar functionality can be integrated into the shell prompt.
- **cd** - Change directory. Moves your current working directory to a different one; accepts relative and absolute arguments.

```
$ cd ~/CS344/prog1
$ pwd
/nfs/stak/faculty/b/brewsteb/CS344/prog1
$ cd ..
$ pwd
/nfs/stak/faculty/b/brewsteb/CS344
$ cd ~
$ pwd
/nfs/stak/faculty/b/brewsteb
```

~ is a shortcut to your home directory

/ is the root of the entire directory structure

Common UNIX Commands - Directories & Files

- **`ls`** - Displays the files in a given directory. Accepts relative and absolute arguments.

```
$ ls
CS344    CS464    pidtest
$ ls -pla
drwx--x--x. 1 brewsteb upg57541   896 Jul 14 10:35 ./
drwxr-xr-x. 1 root     root      1100 Jun 22 21:38 ../
-rw-r--r--. 1 brewsteb upg57541  2431 Aug 18  2015 .bashrc
drwx------. 1 brewsteb upg57541  4153 Aug 13  2015 CS344/
drwx------. 1 brewsteb upg57541  9472 Jul 12 18:49 CS464/
-rwxrwx---. 1 brewsteb upg57541  6561 Mar  2 09:43 pidtest
$ ls -pla --color=auto
drwx--x--x. 1 brewsteb upg57541   896 Jul 14 10:35 ./
drwxr-xr-x. 1 root     root      1100 Jun 22 21:38 ../
-rw-r--r--. 1 brewsteb upg57541  2431 Aug 18  2015 .bashrc
drwx------. 1 brewsteb upg57541  4153 Aug 13  2015 CS344/
drwx------. 1 brewsteb upg57541  9472 Jul 12 18:49 CS464/
-rwxrwx---. 1 brewsteb upg57541  6561 Mar  2 09:43 pidtest
```

Common UNIX Commands - Directories & Files

- **`alias`** - Create a shortcut for running a program.

```
$ alias l="ls -pla --color=auto"
$ l
drwx--x--x. 1 brewsteb upg57541   896 Jul 14 10:35 ./
drwxr-xr-x. 1 root     root      1100 Jun 22 21:38 ../
-rw-r--r--. 1 brewsteb upg57541  2431 Aug 18  2015 .bashrc
drwx------. 1 brewsteb upg57541  4153 Aug 13  2015 CS344/
drwx------. 1 brewsteb upg57541  9472 Jul 12 18:49 CS464/
-rwxrwx---. 1 brewsteb upg57541  6561 Mar  2 09:43 pidtest
```

Common UNIX Commands - Directories & Files

- **`mkdir`** - Create directories
- **`rmdir`** - Delete directories
- **`rm`** - Delete files (and directories if used recursively)
- **`mv`** - Move or rename files and directories
- **`cp`** - Copy files and directories

- Example on next page…

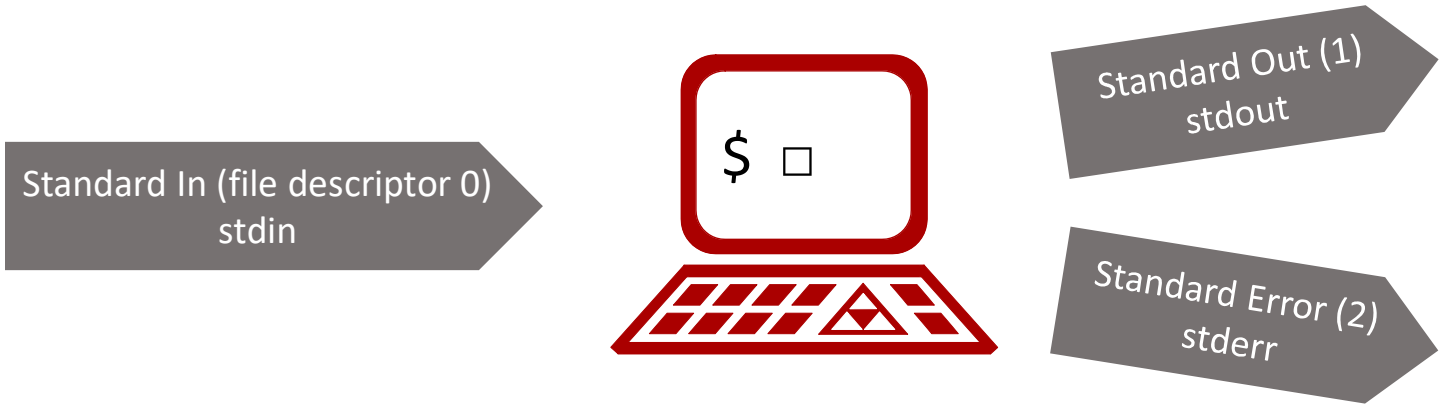
Common UNIX Commands - Directories & Files

```
$ mkdir tempdir
$ touch myfile
$ mv myfile tempdir
$ cd tempdir
$ alias l="ls -pla --color=auto"
$ l
drwxrwx---. 1 brewsteb upg57541  80 Jul 14 11:29 ./
drwxrwx---. 1 brewsteb upg57541 104 Jul 14 11:29 ../
-rw-rw----. 1 brewsteb upg57541   0 Jul 14 11:29 myfile
$ cp myfile myfile_backup
$ rm myfile
rm: remove regular empty file `myfile'? y
$ ls
myfile_backup
$ mv myfile_backup myfile
$ ls
myfile
$ cp myfile ../newfile
$ cd ..
$ ls
newfile tempdir
$ rmdir tempdir
rmdir: failed to remove `tempdir': Directory not empty
$ rm -rf tempdir
```

The `alias` command can be placed into ~/.bashrc where it will be run each time you log in to this computer. Handy!

Aside: Standard files and the Terminal

- The shell automatically opens the terminal for reading-from, with one file, and for writing-to, with two files:



- High level: If you don't specify otherwise, program input and output goes to and from your shell prompt by default on every line
- We view and manipulate this input and output with the new few programs/commands

Common UNIX Commands - Getting Data from Files

- **`echo`** - send character data to standard out

```
$ echo text
text
$ echo test text
test text
$ echo -e "test text\nnext line"
test text
next line
$
```

- `echo` sends its data to stdout, which has not been explicitly changed away from the terminal.
- The terminal dutifully displays data sent to it on your screen (or on another network-attached screen, but that's beyond our scope).

Common UNIX Commands - Getting Data from Files

- **`cat`** - concatenate character data stored in a file with character data from other files.
- Used primarily to dump data to the terminal.

```
$ cat file1
file1contents
$ cat file1 file2
file1contents
file2contents
```

- `cat` sends its data to stdout, which wasn't explicitly changed from the terminal, so out to our screen it comes.

Redirecting stdout

- The > operator tells stdout to open a different file for writing to (affects entire line).

```
$ ls
$ echo -e "cookie\nbeefcake\napple" > foodlist
$ ls
foodlist
$ cat foodlist
cookie
beefcake
apple
```

stdout redirected to open a file named foodlist for writing instead of the terminal

No redirection here, so cat's stdout has the terminal open for writing

Common UNIX Commands - Getting Data from Files

- **`sort`** - Takes data from stdin OR a file and sends the data, alphabetically sorted by line, to stdout.

```
$ echo -e "cookie\nbeefcake\napple" > foodlist
$ cat foodlist
cookie
beefcake
apple
$ sort foodlist
apple
beefcake
cookie
```

Redirecting stdin

- The < operator tells stdin to open a different file for reading from (affects entire line).

```
$ sort foodlist
apple
beefcake
cookie

$ sort < foodlist
apple
beefcake
cookie
```

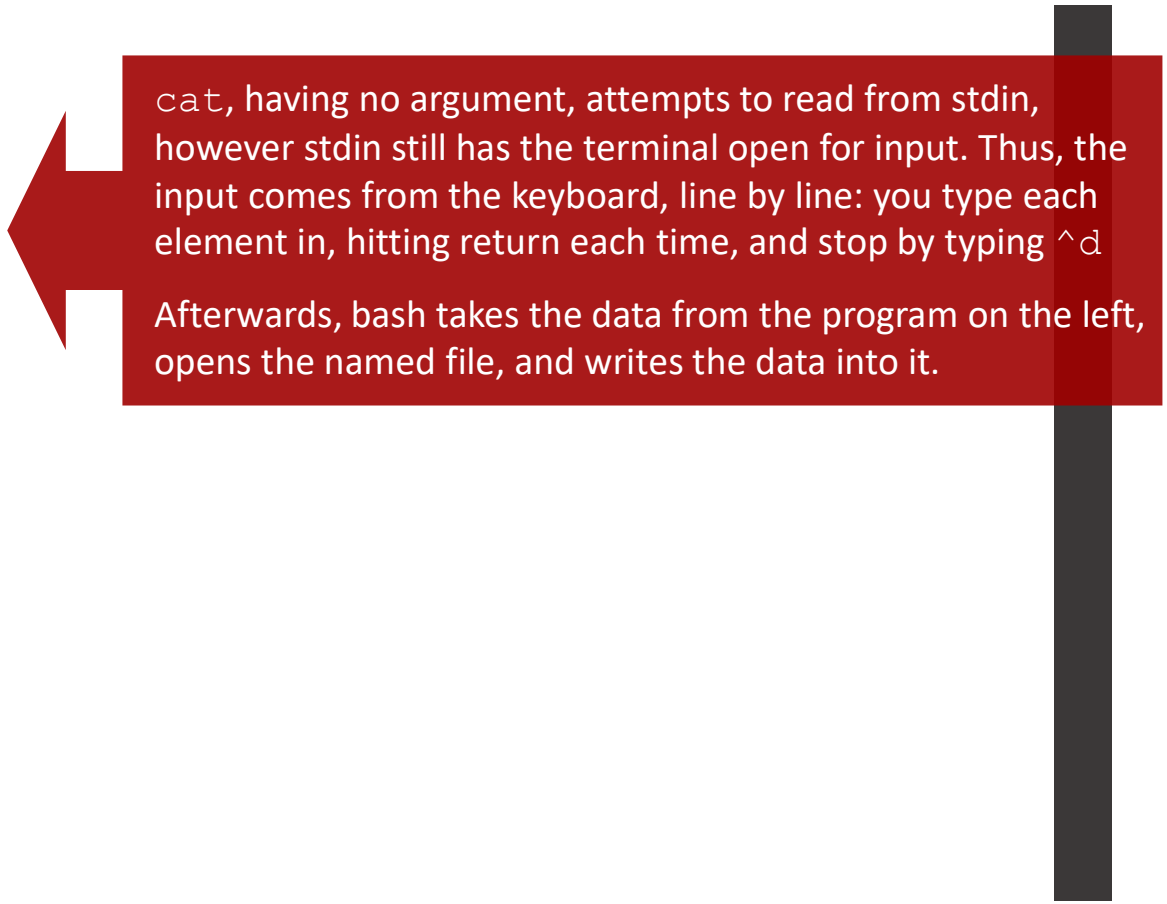
Here, `sort` opens up the file with filename given in the first argument, reads the data in, sorts it, and sends it to stdout

Here, bash opens up the named file, and sends the data on as input to the program on the left.

`sort`, meanwhile, has no first argument, so it reads data from stdin, and does not know where the data comes from! It gets sorted and sent to stdout.

stdin with no Redirection

```
$ cat > list
cookie
beefcake
apple
$ cat list
cookie
beefcake
apple
```



Redirecting both stdin and stdout

```
$ echo -e "cookie\nbeefcake\napple" > foodlist
$ sort < foodlist
apple
beefcake
cookie
$ sort < foodlist > sortedList
$ cat sortedList
apple
beefcake
cookie
```

1. `sort` has no file given to it as an argument, so it tries to read from stdin
2. stdin for this line gets redirected from the file foodlist
3. The file contents get fed into `sort`, which sorts them.
4. stdout is redirected to the file sortedList,
5. `sort`'s output is written to stdout, which goes to sortedList

```
$ sort >sortedList <foodlist
$ cat sortedList
apple
beefcake
cookie
```

The order of the stdin and stdout redirections doesn't matter. You can leave out the space, too, if that makes thing more clear

Shell Filename Expansion

- Certain metacharacters are expanded and replaced with all files with matching names
- * - matches anything
- ? – matches any one character

- Note that this isn't a regular expressions encoding - that's performed with different programs, most notably `grep`

Shell Filename Expansion Examples

- List all files in the current directory whose names start with “CS344”:

```
$ ls CS344*
```

- List all files in the current directory that have “CS344” somewhere in their name:

```
$ ls *CS344*
```

- In all subdirectories that contain “CS344“ followed by any character, list all files:

```
$ mkdir CS344; touch CS344/fileInCS344
$ mkdir CS3440; touch CS3440/fileInCS3440
$ mkdir CS34401; touch CS34401/fileInCS34401
$ ls */*CS344?
CS3440/fileInCS3440
```

Pipes

- Provide a way to communicate between commands without using temporary files to hold the data

```
$ echo -e "cookie\nbeefcake\napple" | sort
apple
beefcake
cookie
$ echo -e "cookie\nbeefcake\napple" > foodlist
$ cat foodlist | sort
apple
beefcake
cookie
$ cat foodlist | sort > sortedList
$ cat sortedList
apple
beefcake
cookie
```

A pipe takes the stdout of one command and connects it to the stdin of the next

`more`, append, and pipes

- **`more`** - Takes character data and displays only one screen-full at a time; navigate with up, down, & spacebar; quit with q.

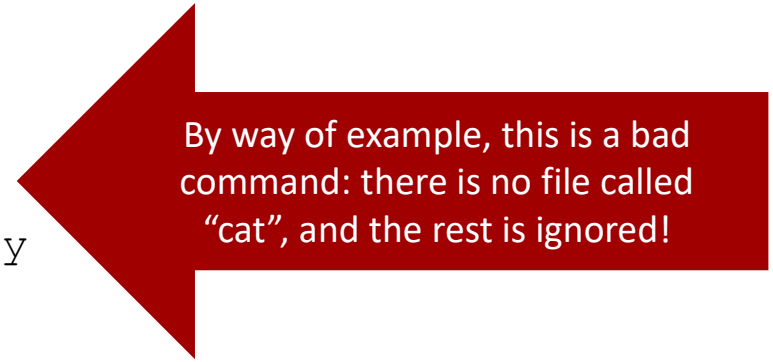
- >> operator *appends* character data to the end of a file, as opposed to *replacing* the contents of an existing file as > does.

- Shell demo script:

```
$ echo -e "1\n2\n3\n4\n5" > rowfile
$ echo -e "6\n7\n8\n9\n10\n11\n12\n13\n14\n15" >> rowfile
$ echo -e "16\n17\n18\n19\n20\n21\n22\n23\n24\n25" >> rowfile
$ cat rowfile | sort -nr >> rowfile
$ cat rowfile
$ cat rowfile | more
```

Live Shell Demo Script

```
$ bash
$ echo -e "cookie\nbeefcake\napple" > foodlist
$ echo -e "dogfish\neel" > fishlist
$ ls; ls
$ cat foodlist fishlist
$ sort < cat foodlist fishlist
-bash: cat: No such file or directory
$ cat foodlist fishlist > biglist
$ cat biglist
$ sort < biglist > sortedBiglist
$ cat sortedBiglist
$ sort -r < biglist > reverseSortedBiglist
$ cat reverseSortedBiglist
$ cat *list *list | sort | more
```



Bash Shell Scripting

Benjamin Brewster

Shell Scripting

- All of the commands that are accessible from the shell can be placed into a shell “script”

- Shell scripts are executed line by line, as if they were being typed in line by line

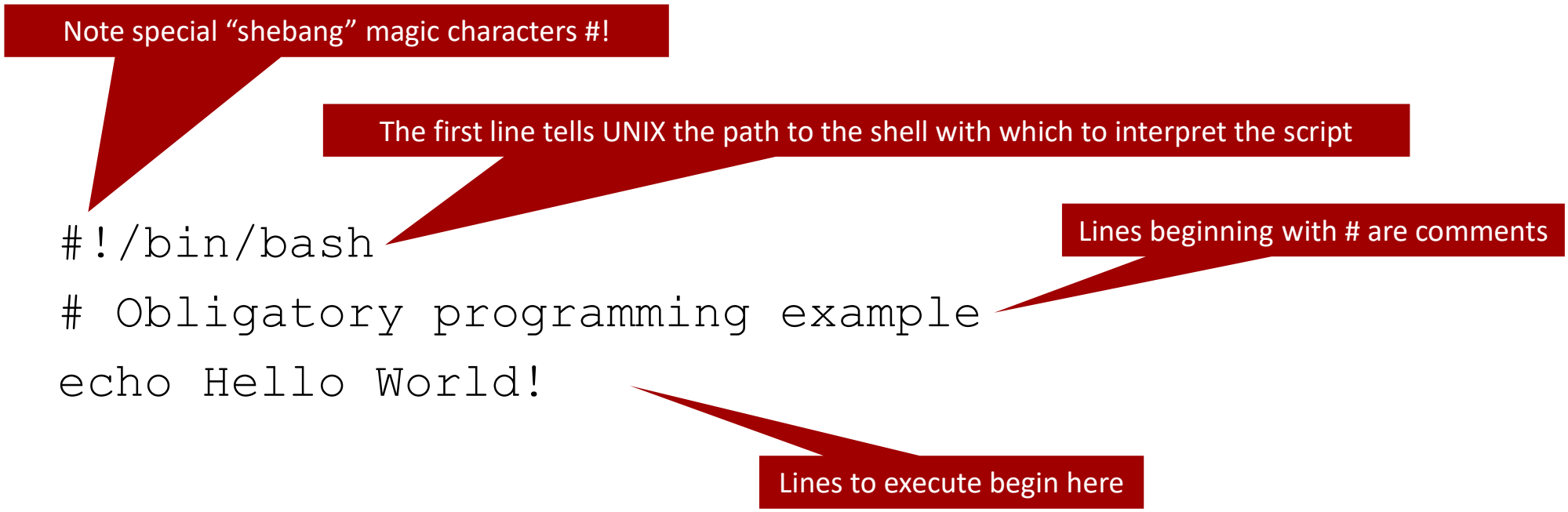
Shell Script – High Level

- High level programming language
  - Variables, conditionals, loops, etc.
- Interpreted
  - No compiling, no variable declaration, no memory management
- Powerful/efficient
  - Do lots with little code
  - Can interface with any UNIX program that uses stdin, stdout, stderr
- String and file oriented

When to Use Shell Scripts?

- Automating frequent UNIX tasks
- Simplify complex commands
- For small programs that:
  - … you need to create quickly
  - … will change frequently
  - … need to be very portable
  - … you want others to see and understand easily
- As a glue to connect together other programs

Hello World!


Note special “shebang” magic characters #!
The first line tells UNIX the path to the shell with which to interpret the script
Lines beginning with # are comments
#!/bin/bash
Obligatory programming example
echo Hello World!
Lines to execute begin here

Variables

- Create a variable by simply assigning to it:

```
myint=1
mystr=Ben
mystr=123
```

- Get the value back out by using the $ operator in front of the variable:

```
myint=1
echo $myint
```

Quote Marks – Protecting Your Text

- Quotation marks allow you to control the expansion of variables within strings of text, and group text into a single argument.

- Single Quotes (' '): No variable expansion

```
$ printf 'all your base $abtu\n'
all your base $abtu
```

- Double Quotes (" "): Variables are expanded

```
$ printf "all your base $abtu\n"
all your base are belong to us
```

Note how these quoted strings are interpreted as one argument

- Backslash character (\) means evaluate literally, instead of interpret:

```
$ nt="NOTTACHANCE"
$ printf "\$nt"; printf "$nt\n"
$ntNOTTACHANCE
```

Printing Example - Demonstration

```
$ echo -e "cat\ndoge\nkat\ndoug" | sort
cat
doge
doug
kat
```

Somewhat complicated

Versus:

```
$ cAd
cat
doge
doug
kat
```

Easier to type! How do we built this script?

Printing Example - Built From Scratch

```
$ echo "#!/bin/bash"
-bash: !/bin/bash": event not found
```

This occurs because the history feature engages when ! Is the first character

```
$ echo "#\x21/bin/bash"
#\x21/bin/bash
```

Without the -e argument, the hexadecimal ASCII code (21) for ! won't be expanded

```
$ echo -e "#\x21/bin/bash" > cAd
$ echo -e "echo \"cat\"\necho \"doge\"\necho \"kat\"\necho \"doug\"" | sort >> cAd
$ chmod +x cAd
$ cAd
cat
doge
doug
kat
```

Make this file be executable - more on `chmod` later

Editors and Word Processors

- Instead of using “echo >>” to build up a file, we typically use word processing programs like:
  - emacs
  - vi
  - pico

Upcoming lecture on vi

- You can write UNIX files on other operating systems, but the differences in line endings in file formats is a challenge
  - `dos2unix` converts windows line endings to those used by UNIX

Shell Keywords

- Keywords are commands in bash that are interpreted by the shell as delimiters, branching constructs, and loops, among others.

- `compgen` generates possible completion matches for various categories of keywords.
  - -k is all reserved keywords

- Can’t use keywords as variable names

```
$ compgen -k
if
then
else
elif
fi
case
esac
for
select
while
until
do
done
in
function
time
{
}
!
[[
]]
coproc
```

Environment Variables

- A set of variables that are always available in your shell
- These environment variables control options that change the operation of the shell
- Here are a few common ones:
  - PATH - the set of directories bash will search through to find a command
  - HOME - a shortcut back to your home directory (equivalent to ~)
  - SHELL - the full path to the default shell
  - HOSTNAME - the name of the computer you're currently using

There's about a billion more of these: you can get a complete list of all of them plus their current contents like so:

```
$ printenv
```

Environment Variables - PATH and HOME

```
$ echo $PATH
/bin:/sbin:/usr/local/bin:/usr/bin:/usr/local/apps/bin:/usr/bin/X11
:/nfs/stak/faculty/b/brewsteb/bin:.

$ echo HOME
HOME

$ echo $HOME
/nfs/stak/faculty/b/brewsteb

$ echo ~
/nfs/stak/faculty/b/brewsteb

$ echo $~
$~
```

Oops

~ is not a variable, it controls the bash feature called *tilde expansion*

Special Parameters

- A set of variables that are always available in a script
- Here are a few common ones:
  - $ - the process ID of the script itself (every running process has a unique PID)
  - ? - the return value of the previously terminated command or script
  - # - the number of arguments (positional parameters) given when a script is executed
  - 1 - the first argument (positional parameter) as of when the script was ran
  - 2 - the second argument (positional parameter) as of when the script was ran
  - 3, 4, etc.

These and more are all explained on the page: Built-In Shell Variables

Return Values Examined

- The `exit` bash shell command and C `exit()` function return their results to the `?` variable.
- 0 is interpreted to mean no errors (normal execution), anything else is a result with specific meaning (usually an error)

```
$ cat exittest
#!/bin/bash
ls
printf "result of ls: $?\n"
exit 5
$ exittest
exittest      testloop       prog1
result of ls: 0
$ echo $?
5
```

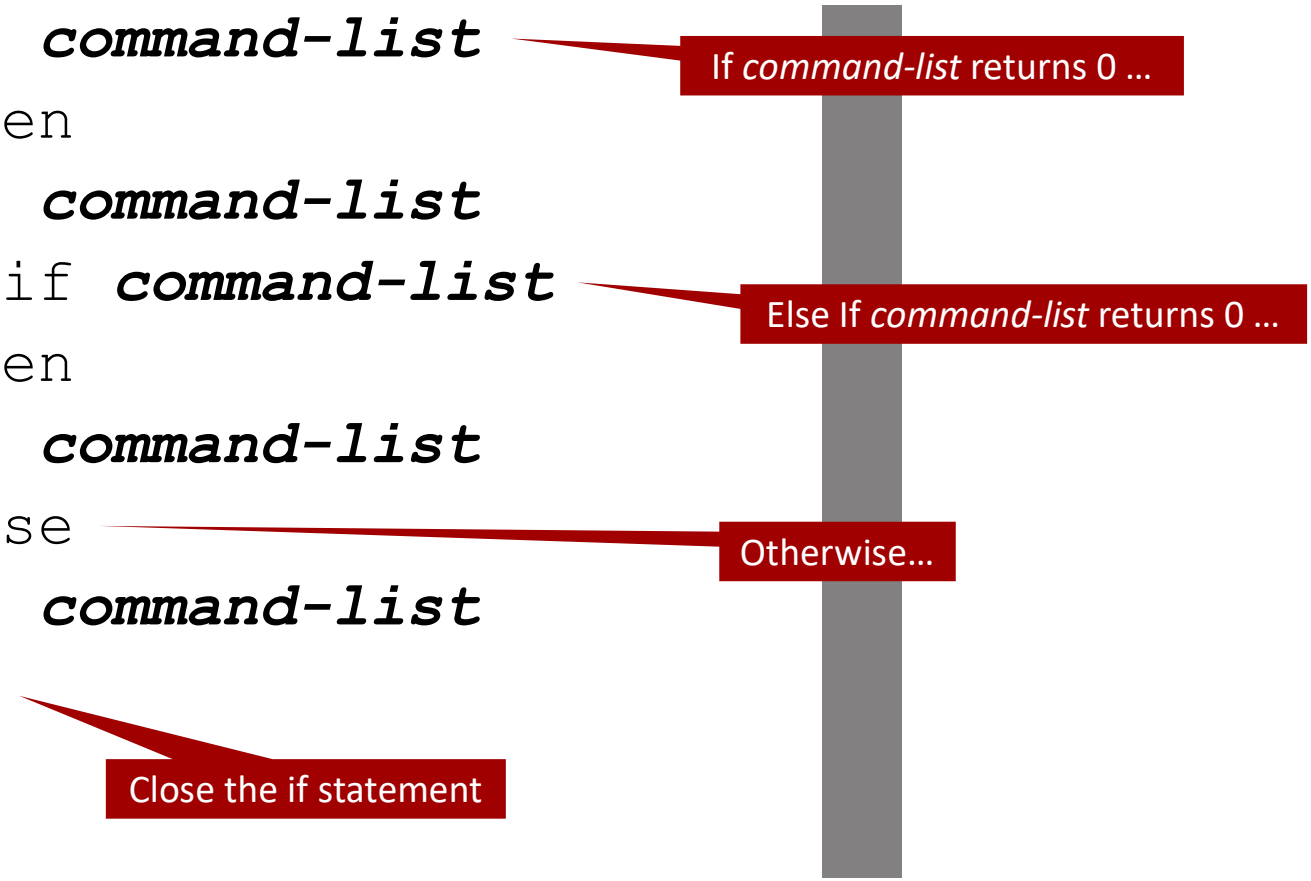
Return Values Examined

- The `exit` bash shell command and C `exit()` function return their results to the `?` variable.
- 0 means no errors (normal execution), anything else is a result with specific meaning (usually an error)

```
$ echo hello world
hello world
$ echo $?
0
$ cd %=^2%21
=^2%21: No such file or directory.
$ echo $?
1
```

The bash `if` Statement - Return Value Enabled

```
if command-list
then
    command-list
elif command-list
then
    command-list
else
    command-list
fi
```



Error Handling

- The shell will keep executing even if commands have the wrong syntax, delete files, break things, and in general cause havoc

- If you want the shell to exit if any commands have a problem (i.e. return a non-zero value after executing, with some exceptions):
  - Use -e with /bin/bash

- Most *signals* will kill script immediately
  - E.g.: CTRL-C sends SIGINT

Error Handling - Cry Havoc!

```
$ cat havoc
#!/bin/bash
cd qwcein
ceiqim
eqo eo
$ chmod +x havoc
$ havoc
./havoc: line 2: cd: qwcein: No such file or directory
./havoc: line 3: ceiqim: command not found
./havoc: line 4: eqo: command not found
```

Protecting Your Thesis - Disastrous results

```
#!/bin/bash
cp thesis.docx thesis_current.docx
rm -f thesis.docx
```

Protecting Your Thesis - One Fix

```
#!/bin/bash -e
cp thesis.docx thesis_current.docx
rm -f thesis.doc
```

Will only be executed if the result of the previous command was 0

Protecting Your Thesis - Another Fix

```
#!/bin/bash
if cp thesis.docx thesis_current.docx
then
   rm –f thesis.docx
else
   echo "copy failed” 1>&2
   exit 1
fi
```

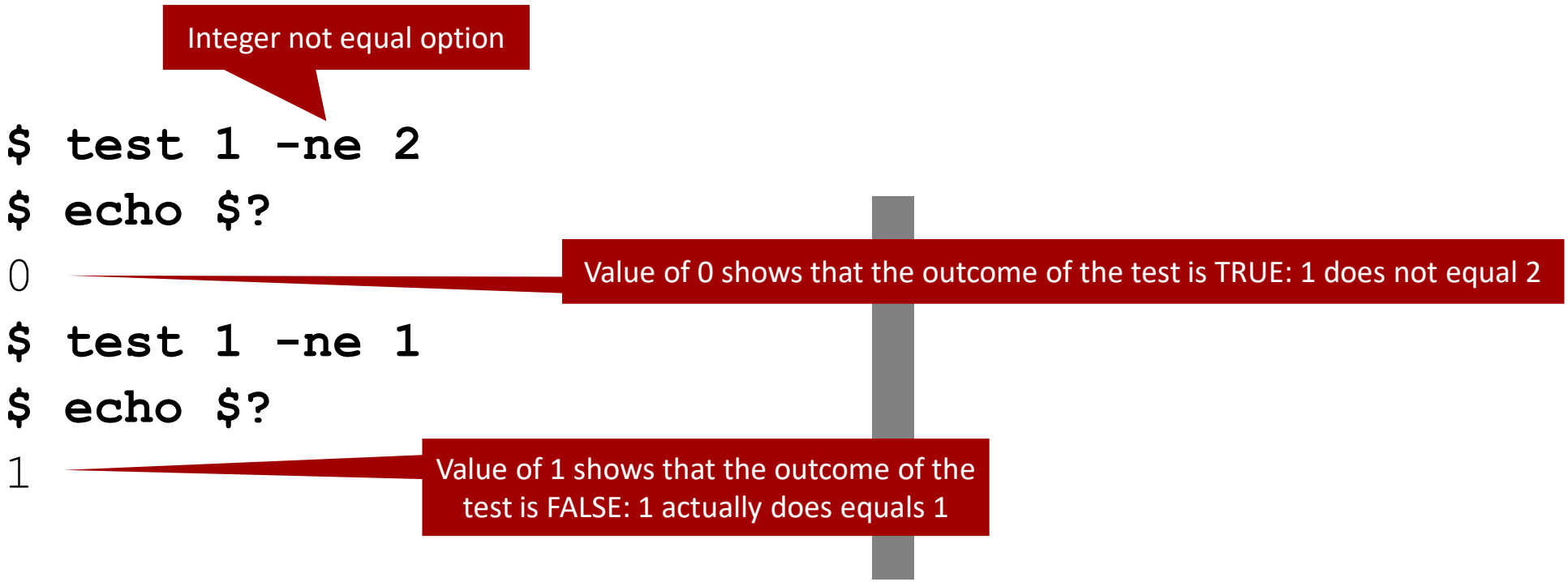
Protecting Your Thesis - Another Fix

```
#!/bin/bash
if cp thesis.docx thesis_current.docx
then
   rm –f thesis.docx
else
   echo "copy failed” 1>&2
   exit 1
fi
```

Bash command meaning: Take anything going to file descriptor 1 (stdout) and redirect it to file descriptor 2 (stderr) for the duration of this command

Protecting Your Thesis - Yet Another Fix with `test`

- You can test for file existence, equality of strings, length, permissions, number equality, etc.



Protecting Your Thesis - Yet Another Fix with `test`

```
#!/bin/bash
cp thesis.docx thesis_current.docx
if test $? -ne 0
then
   echo "copy failed" 1>&2
   exit 1
fi
rm -f thesis.docx
```

Again: bash command meaning: Take anything going to file descriptor 1 (stdout) and redirect it to file descriptor 2 (stderr) for the duration of this command

`for` Loop Syntax

```
$ cat forloop
#!/bin/bash
for i in a b c d
do
    printf "<%s>" $i
done
printf "\n"
$ forloop
<a><b><c><d>
$
```

`while` Loop Syntax

Test can also be written as square brackets:
`while [ $i -ne 2 ]`

```
$ cat whileloop
#!/bin/bash
i=0
while test $i -ne 2
do
     printf "i = $i, not stopping yet\n"
     i=$(expr $i + 1)
done
printf "Stopping, i = $i\n"
$ whileloop
i = 0, not stopping yet
i = 1, not stopping yet
Stopping, i = 2
$
```

Huh?

Subshells

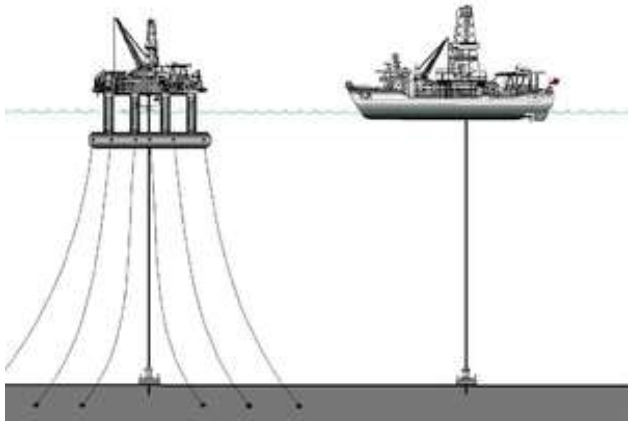
- Some operators (like =) and commands require strings or numbers to operate on, not other commands:

```
$ cat sumtest
#!/bin/bash
num1=9
num2=3 + $num1
echo "num2: $num2"
$ sumtest
./sumtest: line 3: +: command not found
num2:
$
```

Subshells

- If we want to increment, or set to an arbitrary value, we first have to calculate that value and return it *as text* to the script, then the script can continue.

- These are executed as entirely separate shells (called subshells) in their own processes that run and return.



Subshells

- Two ways to do command substitution with subshells:

```
i=`expr $i + 1`
i=$(expr $i + 1)
```

Backticks! Original method

Preferred method: POSIX compliant, doesn't need escaping when nested

- Both methods evaluate the expression and grab the results from stdout of the subshell
- Double parenthesis construct does arithmetic expansion and evaluation directly, no `expr` command needed:

```
i=$(( 9 + 9 ))
(( i++ ))
```

Common UNIX Commands - Trapping Signals

- Use the **`trap`** command to catch *signals* (like SIGINT generated by hitting CTRL+C) and clean up yer mess.

- Usage:

  `trap <code to execute> list of signals`

- Example:

```
#!/bin/bash
TMP="myCoolFilename$$"
trap "rm -f $TMP; echo 'CTRL+C received, exiting'; exit 1" INT HUP TERM
echo "lotsa text" > TMP
while [ 1 -ne 2 ]
do
        echo "Never ending loop - hit CTRL+C to exit!"
done
```

Common UNIX Commands - Trapping Signals

- Use the **`trap`** command to catch *signals* (like SIGINT generated by hitting CTRL+C) and clean up yer mess.

- Usage:

  `trap <code to execute> list of signals`

- Example:

```
#!/bin/bash
TMP="myCoolFilename$$"
trap "rm -f $TMP; echo 'CTRL+C received, exiting'; exit 1" INT HUP TERM
echo "lotsa text" > TMP
while [ 1 -ne 2 ]
do
        echo "Never ending loop - hit CTRL+C to exit!"
done
```

Lots more about these later

Error!

Common UNIX Commands - Trapping Signals

- Use the **`trap`** command to catch *signals* (like SIGINT generated by hitting CTRL+C) and clean up yer mess.

- Usage:

  `trap <code to execute> list of signals`

- Example:

```
#!/bin/bash
TMP="myCoolFilename$$"
trap "rm -f $TMP; echo 'CTRL+C received, exiting'; exit 1" INT HUP TERM
echo "lotsa text" > TMP
while [ 1 -ne 2 ]
do
        echo "Never ending loop
done
```

Lots more about these later

Error! Should be $TMP, otherwise the echo goes to a file called “TMP”, instead of a file with name stored in the TMP variable initialized above!

Bash Script Demos

Benjamin Brewster

readloop

```
$ cat readloop
#!/bin/bash
while read myLine
do
        echo "$myLine"
done < $1
$ readloop test_file
1       1       1       1       1
9       3       4       5       5
6       7       8       9       7
3       6       8       9       1
3       4       2       1       4
6       4       4       7       7
$
```

This line stuffs the contents of the file, whose name is the string in the first argument, into the while loop!

Purpose: Read each line in a file, then echo it back out

arrayloop

- Bash arrays have complicated syntax!

```
$ cat arrayloop
#!/bin/bash
array=( one two three )
for i in "${array[@]}"
do
        echo $i
done
$ arrayloop
one
two
three
$
```

Declares and fills array

array[n] fetches n-th element in array
array[@] fetches ALL elements in array

Purpose: Output the contents of an array

forloop

```
$ cat forloop
#!/bin/bash
oneline="1    2    3    4    5"
for i in $oneline
do
        echo "i is: $i"
done
$ forloop
i is: 1
i is: 2
i is: 3
i is: 4
i is: 5
```

Tab delimited, but works for spaces, too

Could also sum them up, etc. here

Purpose: Manipulate each number on a single line, when the length of that line is unknown

sumloop

```
$ cat sumloop
#!/bin/bash
sum=0
TMP1=./tempfile
echo -e "8\n7\n6" > $TMP1

while read num
do
        echo "In Loop"
        echo "num: $num"
        sum=`expr $sum + $num`
        echo "sum: $sum"
        echo -e "End of Loop\n"
done < $TMP1
```

One number per line

Unlike readloop, this is not $1; this uses the filename specified above for input

```
$ sumloop
In Loop
num: 8
sum: 8
End of Loop

In Loop
num: 7
sum: 15
End of Loop

In Loop
num: 6
sum: 21
End of Loop
```

Purpose: Sum up the numbers in a file consisting of one number per line

stdinread

```
$ cat stdinread
#!/bin/bash
cat > "tempfile"
cat tempfile
$ stdinread
I like cheese
I like cheese
$ cat test_file | stdinread
1       1       1       1       1
9       3       4       5       5
6       7       8       9       7
3       6       8       9       1
3       4       2       1       4
6       4       4       7       7
```

The terminal waits here for input, only continuing when you hit CTRL-D (EOF)

Purpose: Capture data from stdin and direct it to a temp file

trtest

```
#!/bin/bash
# This script converts a row file ./tempinputfile into a column file ./tempcolfile,
# then back into a row file ./temprowfile<PID>
inputFile="tempinputfile"
tempCol="tempcolfile"
tempRow="temprowfile"

# Make the input row file
echo -e "1\t2\t3\t4\t5" > $inputFile

# Append each number onto the end of a temporary column file by cutting specific columns
cut -c 1 $inputFile > $tempCol
cut -c 3 $inputFile >> $tempCol
cut -c 5 $inputFile >> $tempCol
cut -c 7 $inputFile >> $tempCol
cut -c 9 $inputFile >> $tempCol

# Convert the column file back into a row file
cat $tempCol | tr '\n' '\t' > "$tempRow$$"

# Add a newline char to the end of the row file, for easier printing
echo >> "$tempRow$$"
```

Purpose: Convert a row file into a column file and back again

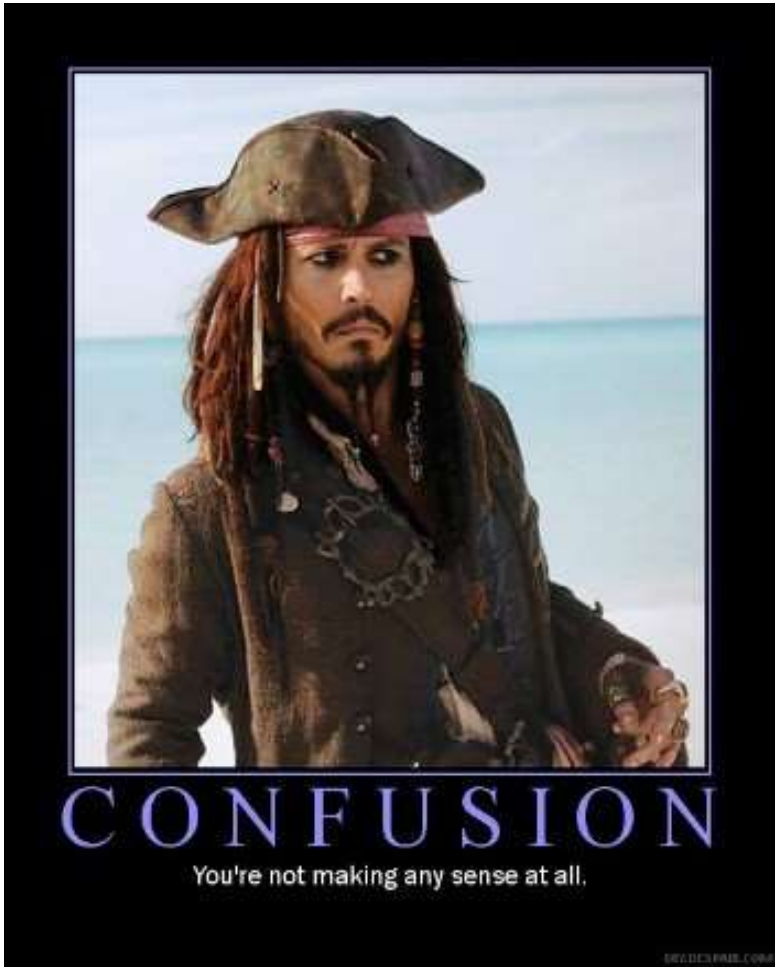
Solving Problems & Debugging

Benjamin Brewster

The Challenge

- You have been asked to write a piece of software
  - You have been given specifications

- You are not sure where to start
  - Perhaps you don’t understand the underlying technology
  - Perhaps you don’t understand all the specifications
  - Perhaps you are not sure how the different components work together



Advice to Follow

- The following advice applies to every single software project that you will encounter in your life!

- And certainly much more than software.

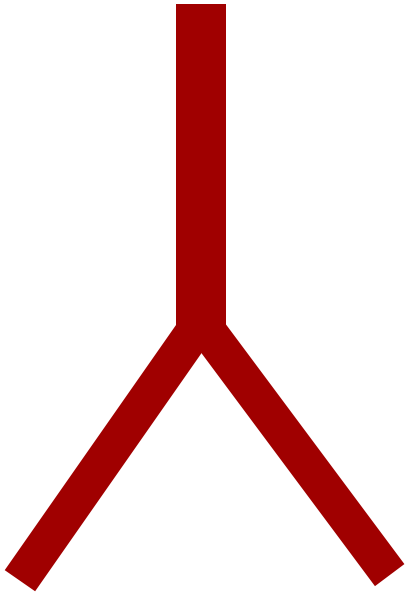
[Warning: philosophy]

Making a Large Problem Seem Easy

- Divide and conquer!

- Break the problem into smaller and smaller pieces - ask yourself about each piece:
  - Do I already know how to do this?
  - If not, does this seem easy?

Example: Program 1



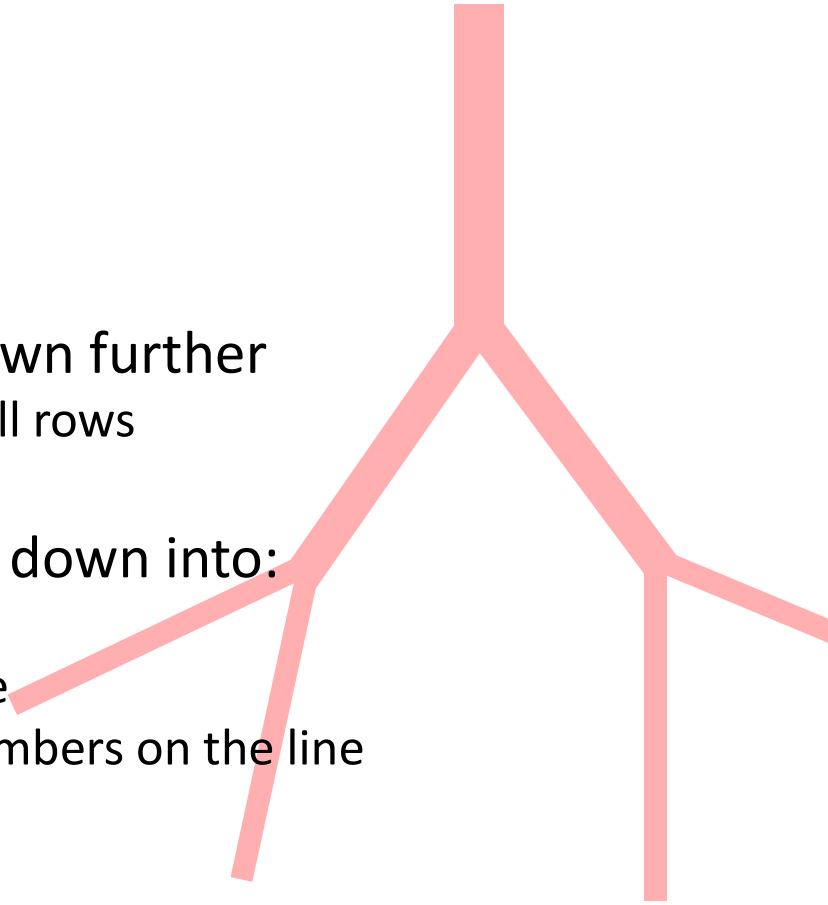
- Identify the core task
  - Compute statistics
  - Ignore the rest (handling signals, etc.)
- Break the core task into pieces
  - Need to compute statistics on rows AND on columns
- Focus on one of the sub-pieces
  - How do we compute stats *just* for rows?

Example: Program 1

- Still not sure what to do, so let's break it down further
  - Try and compute stats for one *single* row, not all rows

- Computing stats for one row can be broken down into:
  1. We need to read one line from a file
  2. Then we need to sum the numbers in the line
  3. Then we need to divide by the number of numbers on the line
  4. Then we need to print out the result

- Each of *these* look doable!



Searching with a Mission - Step 1

- **How do we read one line from a file?**
- Let's read through all of the sources available to you, this time looking for a specific solution to reading a single line
- Re-read technical reference documentation once you have a compact problem
  - You'll find that you'll pay closer attention
  - You'll remember more of what you read
  - You'll find the answers to your problem
  - Before, you didn't have a goal, so your brain didn't bother to internalize or remember the information

Common UNIX Commands – Reading Data

- **read** – Get data from either stdin or a file (with -u option)

```
$ cat readtest
#!/bin/bash
echo "Enter in a string, then hit enter:"
read myvar
echo "You entered: $myvar"
echo $myvar > "tempfilename$$"
echo $myvar >> "tempfilename$$"
echo $myvar >> "tempfilename$$"
echo "The temp file contents are:"
cat "tempfilename$$"
echo "The first line of the file is:"
read entireline < "tempfilename$$"
echo "$entireline"
read firstword restofline < "tempfilename$$"
echo "First word of first line: \"$firstword\""
echo "Rest of line: \"$restofline\""
rm -f "tempfilename$$"
```

```
$ readtest
Enter in a string, then hit enter:
THIS SENTENCE IS FALSE
You entered: THIS SENTENCE IS FALSE
The temp file contents are:
THIS SENTENCE IS FALSE
THIS SENTENCE IS FALSE
THIS SENTENCE IS FALSE
The first line of the file is:
THIS SENTENCE IS FALSE
First word of first line: "THIS"
Rest of line: "SENTENCE IS FALSE"
```

Learning to Read - Step 1

- You figured out how to read a single line from stdin into a variable:
  - `read singlelinevar`
- Test your knowledge
  - Write a tiny shell script to test it
  - You also figure out how to print the line to the screen to test it

```
$ cat readtest
#!/bin/bash
read singlelinevar
echo $singlelinevar
$ cat data
6 4 4 7 7
$ cat data | readtest
6 4 4 7 7
```

Summing up - Step 2

- Next step: **sum the numbers on a line**
- Decompose further:
  - Forget the line, how do I add two numbers?
  - Then worry about getting those numbers from the line

- You read the docs (finding it faster this time!), and figure out how to add two numbers and store them in a variable

```
sum=`expr 3 + 4`

sum=`expr $var1 + $var2`
```

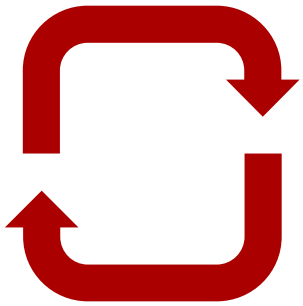
Lotsa Summing - Step 2

- Now, how to take those numbers from the line you read?
- And - a new challenge: what if you don't know how many numbers are on a line?
  - `read X` reads the entire line into a variable - how do you break it up?

- Raw documentation might not be as useful as putting the question up on a discussion board, asking a friend, or searching online

Loop it - Step 2

- Let's loop it - we've learned about the bash **for** loop:



```
$ cat data
6 4 4 7 7
$ cat readlooptest
#!/bin/bash
read myLine
sum=0
for i in $myLine
do
        sum=`expr $sum + $i`
done
echo "sum is: $sum"
$ cat data | readlooptest
-bash: ./readlooptest: Permission denied
$ chmod +x readlooptest
$ cat data | readlooptest
sum is: 28
```

Dividing and Printing - Step 3 & 4

- Next step: **divide by the number of numbers on the line**
- Decompose further:
  - How can we tell how many numbers are on a line?
  - How do we divide two numbers?
- We can simply add a count variable to our previous code, and use the division operator instead of addition
- Then use printf to combine it all!

```
$ cat data
6 4 4 7 7
$ cat countdividelooptest
#!/bin/bash
read myLine
sum=0
count=0
for i in $myLine
do
        sum=`expr $sum + $i`
        count=`expr $count + 1`
done
mean=`expr $sum / $count`
echo "$count entries sums to: $sum"
echo "mean average is: $mean"
$ cat data | countdividelooptest
5 entries sums to: 28
mean average is: 5
```

Huh. `expr` only works with integers!

Problem Solving Summary

- Break down the problem until you have a tiny problem that looks solvable
- Then use docs/friends/online searches to find the solution
  - You will learn much more if you are reading with a well-defined problem
- Solve the tiny problem
  - Make sure that you write the code!
  - Feel good about the success – have a donut:

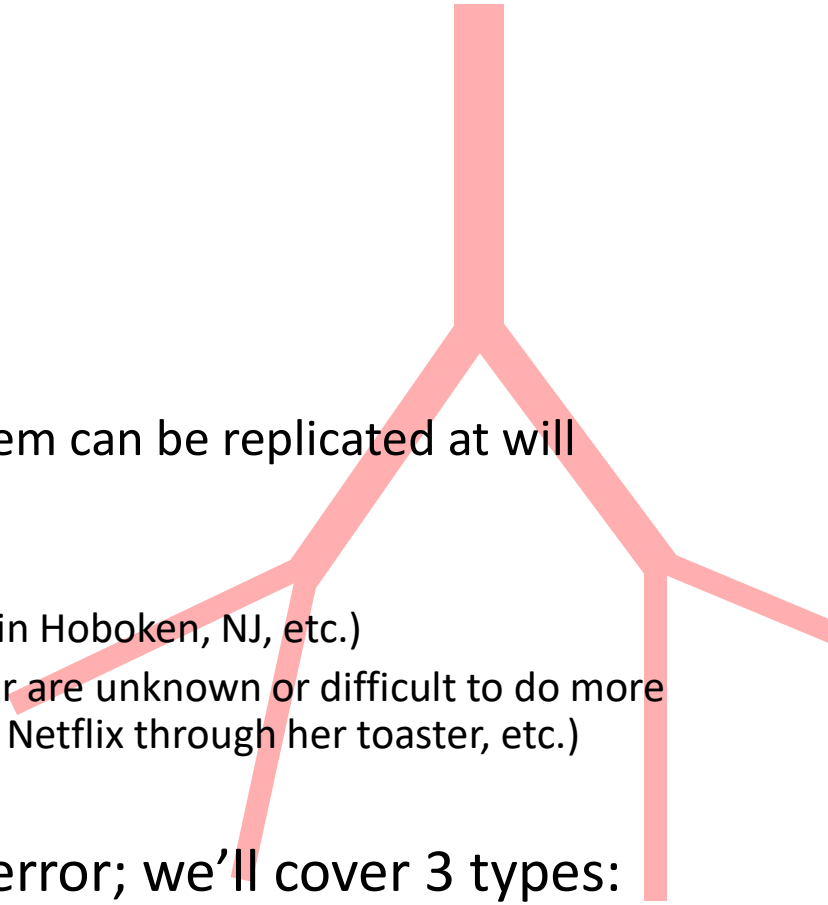
- Now work upwards
  - Integrate your solution into the (slightly) larger problem
- Repeat!
- See the online Shell Script Compendium page for an example script (“bigloops”) that reads ALL lines from a file, sums them, and computes the averages

Introduction to Debugging

- Debugging is the art of figuring out why something has gone horribly, awfully wrong
- Testing is done to *find* bugs, debugging *removes* them

Debugging Process

1. Reproduce the problem reliably
   - Simplify input and environment until the problem can be replicated at will
     - e.g. Wolf Fence algorithm (next slide)
   - Challenges:
     - Unique environment (space station, aunt Edna's PC in Hoboken, NJ, etc.)
     - Particular sequence of events leading up to the error are unknown or difficult to do more than once (lightning strike, aunt Edna tries to watch Netflix through her toaster, etc.)
2. Examine the process state at the time of error; we'll cover 3 types:
   1. Live Examination
   2. Post-mortem Debugging
   3. Trace Statement



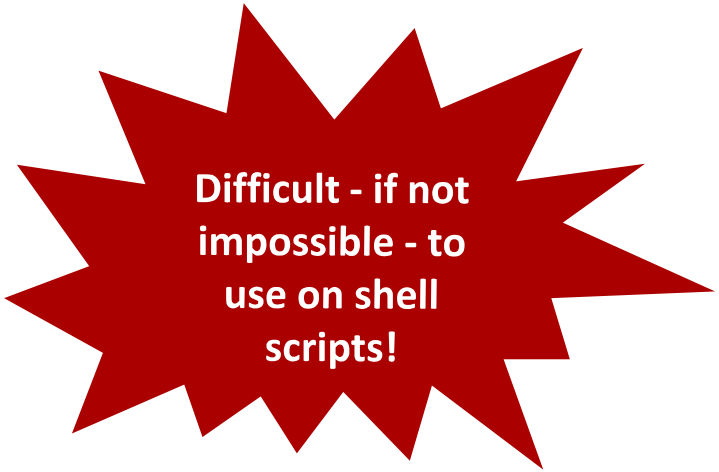
Wolf Fence Algorithm

- If you can't find the source of the problem, or can't reproduce it, narrow things down by dividing and conquering:
  - Bifurcate the code by commenting out half of it, and running it again
  - Change the code to ignore all of the input but one piece and run it again

- "There's one wolf in Alaska; how do you find it? First build a fence down the middle of the state, wait for the wolf to howl, determine which side of the fence it is on. Repeat process on that side only, until you get to the point where you can see the wolf."

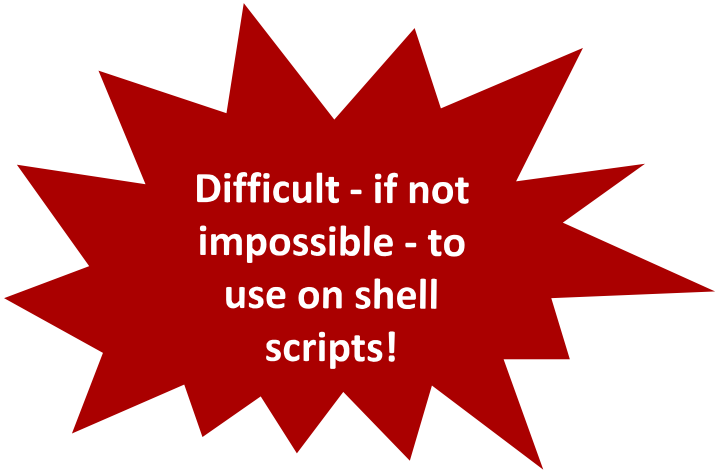
  --E. J. Gauss (1982). "Pracniques: The "Wolf Fence" Algorithm for Debugging", in Communications of the ACM

Technique: Live Examination



- A separate application allows you to pause process execution and view the contents of variables and stack trace directly
- Common tools:
  - IDEs: Microsoft Visual Studio, Eclipse, Monodevelop, etc.
  - Apps: gdb, valgrind
- Variable values can even be edited on the fly!
- Debugger applications and IDEs can be connected to processes on other machines (aka remote debugging)

Post-mortem Debugging



- After the program crashes, UNIX will leave behind a core dump file that can be examined

- The core file stores the contents of that program's virtual memory contents, including CPU registers, program counter, stack pointer, and other OS info

- gdb is used to analyze core files on UNIX:

```
$ gdb myprogram -c coredumpfile
```

Trace Statements

- Trace statements - *what* and *where*
  - Print out the value of your variables as you go (what)
  - Print out where you currently are (where)
- What and where:

```
About to Begin Loop:   j = 0
Now At Start of Loop:  i = 0, j = 0
Sum() function ran:    i = 1, j = 1844835813859
Executed file read:    i = 1, j = ^^^^^^^°ØĆ@«
Contents of file read: fj8283jJ*#Jf8j32@fj
Now at End of Loop:    i = MDKMDKMDK, j = 42941492
Start missile laun^C ^C ^C ^C
```

Garbage! Sum() is busted!

Hull breach!

The horror!

No survivors!

Halt and catch fire!

Debugging Bash Shell Scripts

- Enable printing of command traces before executing each command:
  - Add -x to the end of the first line of the script:

    ```
    #!/bin/bash -x
    ```

    OR

  - Launch the script with a shell that has command traces enabled:

    ```
    $ bash -x sumloop
    ```

Debugging Bash Shell Scripts - Example

```
$ sumloop
In Loop
num: 8
sum: 8
End of Loop

In Loop
num: 7
sum: 15
End of Loop

In Loop
num: 6
sum: 21
End of Loop
```

```
$ bash -x sumloop
+ sum=0
+ TMP1=./TMP1
+ echo -e '8\n7\n6'
+ read num
+ echo 'In Loop'
In Loop
+ echo 'num: 8'
num: 8
++ expr 0 + 8
+ sum=8
+ echo 'sum: 8'
sum: 8
+ echo -e 'End of Loop\n'
End of Loop
+ read num
+ echo 'In Loop'
In Loop
+ echo 'num: 7'
num: 7
++ expr 8 + 7
+ sum=15
+ echo 'sum: 15'
sum: 15
+ echo -e 'End of Loop\n'
End of Loop
+ read num
+ echo 'In Loop'
In Loop
+ echo 'num: 6'
num: 6
++ expr 15 + 6
+ sum=21
+ echo 'sum: 21'
sum: 21
+ echo -e 'End of Loop\n'
End of Loop

+ read num
```

(Quick Note: You cannot pipe to read)

```
$ readpipetest
Contents of readpipetest file:
#####
#!/bin/bash
echo "Contents of readpipetest file:"
echo '#####'
cat readpipetest
echo '#####'
echo
echo "Contents of readpipetest being piped through read:"
cat readpipetest | read firstline
echo '#####'
echo "\"$firstline\""
echo '#####'
#####

Contents of readpipetest being piped through read:
#####
""
#####
```

- The pipe operator runs the next command in a subshell, whose created variables exist only while that subshell is being executed - thus `read` needs to be stuffed full of data and used immediately

WARNING
This lecture contains usages of the Papyrus and Comic Sans fonts.
Observer discretion is advised.

Everything is a File

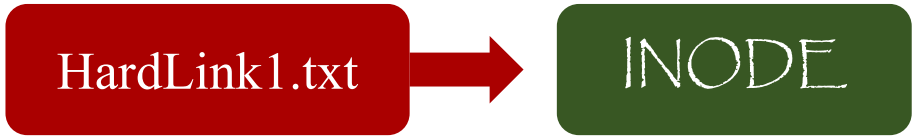
Benjamin Brewster

TRIGGER WARNING
Papyrus font on next slide

Files!

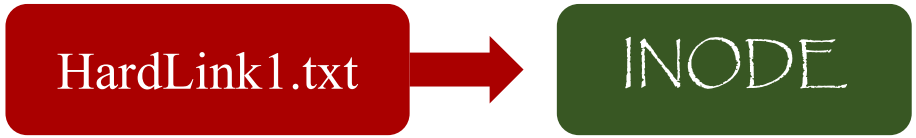
- Files are “inodes” with “hard links” that point to them.

- Inodes are maintained by the file system itself and contain:
  - Pointers to actual file data
  - All meta-information (size, permissions, etc)
  - A “reference count”: how many hard links point to the inode
  - A unique “inode number”

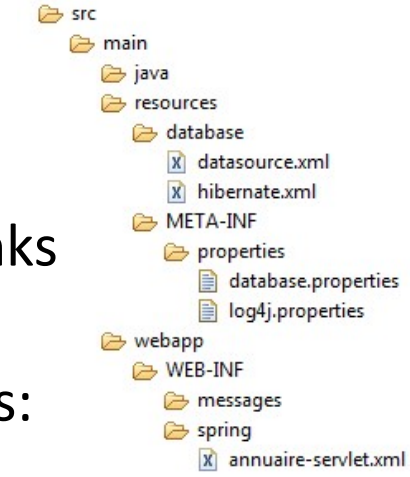


Hard Links

- A text entry in a file system directory that points to an inode
- Connects a **text filename** to an inode on disk



Directories!



- Directories are essentially text files that organize hard links hierarchically
- Create or remove directories with these UNIX commands:
  - mkdir, rmdir
- Because directories *are* files, you can also read the contents of a directory (in C):
  - opendir(), closedir(), readdir(), rewinddir()
- Directory hard link that represents itself: **.**
- Directory hard link that represents its parent directory: **..**

What's in a directory file?

```
$ vim .
" ============================================================================
" Netrw Directory Listing                                        (netrw v149)
"   /nfs/stak/faculty/b/brewsteb/codesamples
"   Sorted by      name
"   Sort sequence: [\/]$,\<core\%(\.\d\+\)\=\>,\.h$,\.c$,\.cpp$,\~\=\*$,*,\.o$,\
"   Quick Help: <F1>:help  -:go up dir  D:delete  R:rename  s:sort-by  x:exec
" ============================================================================
../
./
Curses.pdf
IntroToUnixShell.html
OnE1.txt
OnE1FAQ.txt
OnE1_sol.txt
Prog1.html
Prog1.test
Prog1FAQ.txt
Prog2.html
Prog2FAQ.txt
cursesDemo.c
index.html
```

Directories Can Be Read Like a File

```c
#include <stdio.h>
#include <string.h>
#include <dirent.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>

void main() {
  int newestDirTime = -1; // Modified timestamp of newest subdir examined
  char targetDirPrefix[32] = "brewsteb.rooms."; // Prefix we're looking for
  char newestDirName[256]; // Holds the name of the newest dir that contains prefix
  memset(newestDirName, '\0', sizeof(newestDirName));

  DIR* dirToCheck; // Holds the directory we're starting in
  struct dirent *fileInDir; // Holds the current subdir of the starting dir
  struct stat dirAttributes; // Holds information we've gained about subdir

  dirToCheck = opendir("."); // Open up the directory this program was run in

  if (dirToCheck > 0) { // Make sure the current directory could be opened
    while ((fileInDir = readdir(dirToCheck)) != NULL) { // Check each entry in dir
      if (strstr(fileInDir->d_name, targetDirPrefix) != NULL) { // If entry has prefix
        printf("Found the prefex: %s\n", fileInDir->d_name);
        stat(fileInDir->d_name, &dirAttributes); // Get attributes of the entry

        if ((int)dirAttributes.st_mtime > newestDirTime) { // If this time is bigger
          newestDirTime = (int)dirAttributes.st_mtime;
          memset(newestDirName, '\0', sizeof(newestDirName));
          strcpy(newestDirName, fileInDir->d_name);
          printf("Newer subdir: %s, new time: %d\n", fileInDir->d_name, newestDirTime);
        }
      }
    }
  }

  closedir(dirToCheck); // Close the directory we opened
  printf("Newest entry found is: %s\n", newestDirName);
}
```

This code searches for the most recently modified/created directory whose name matches a certain prefix

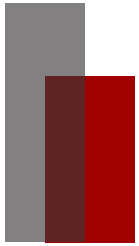
Note the funky single equals sign!

Creating Links

- When you create a file (using `touch`, a C function, etc.), an inode is allocated and a hard link is automatically created

- However, you can create multiple hard links to the same inode
  - So a file can appear in multiple directories at the same time!
  - The same file can also appear under different names
    - Even in the same directory
      - That's pretty weird

- To create a link, use the `ln` or `link` commands

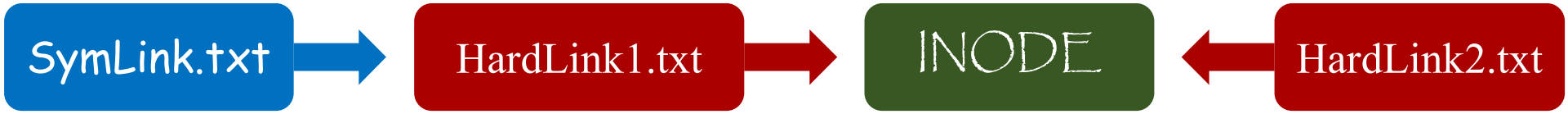


Removing Files in C

- Removing is approximately unlinking everything
  - The inode is garbage collected when ref count == 0
- One way to "remove" a file:
  - `unlink(file_name)`
  - can't unlink directories
- Another way to "remove" a file
  - `remove(file_name)`
  - unlike `unlink()`, remove will delete empty directories
  - if file, `remove()` is identical to `unlink()`
  - if directory, `remove()` is identical to `rmdir()`

Symbolic Link

- A symbolic link is like a Windows shortcut - it's not actually a file, it points to a hard link
- If you delete, rename, or move the hard link a symbolic link points to, the symbolic link become unusable
- You can symbolically link to directories or to files across the network
- Hard links are only available on the local file system

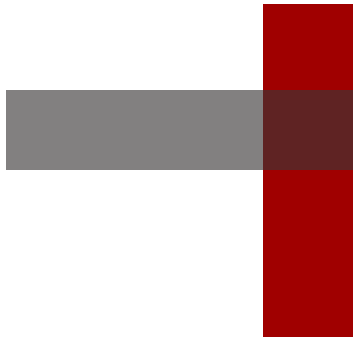


Symbolic Link Example

- I’m on a new server and want access to my home directory’s files stored on my original server

```
NewServer$ ln -s /nfs/rack/u2/b/brewsteb ./myFilesLink
NewServer$ cd myFilesLink
```

- I am now in my filesystem on the new server



Linking Example

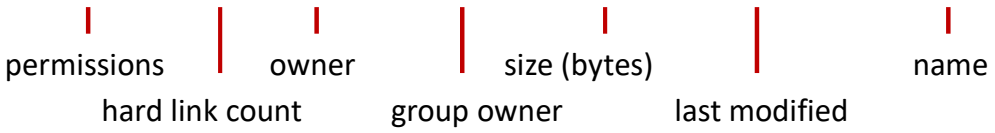
```
$ mkdir 1
$ mkdir 2
$ touch ./1/hardlink1
$ ln ./1/hardlink1 ./2/hardlink2
$ ln -s ./1/hardlink1 ./symlink
$ find -samefile ./1/hardlink1
./1/hardlink1
./2/hardlink2
$ ls -plaR
.:
drwxrwx---. 1 brewsteb upg57541 124 Aug 29 15:11 ./
drwxrwx---. 1 brewsteb upg57541 636 Aug 29 14:56 ../
drwxrwx---. 1 brewsteb upg57541  84 Aug 29 15:11 1/
drwxrwx---. 1 brewsteb upg57541  84 Aug 29 15:11 2/
lrwxrwxrwx. 1 brewsteb upg57541  13 Aug 29 15:11 symlink -> ./1/hardlink1
./1:
drwxrwx---. 1 brewsteb upg57541  84 Aug 29 15:11 ./
drwxrwx---. 1 brewsteb upg57541 124 Aug 29 15:11 ../
-rw-rw----. 2 brewsteb upg57541   0 Aug 29 15:11 hardlink1
./2:
drwxrwx---. 1 brewsteb upg57541  84 Aug 29 15:11 ./
drwxrwx---. 1 brewsteb upg57541 124 Aug 29 15:11 ../
-rw-rw----. 2 brewsteb upg57541   0 Aug 29 15:11 hardlink2
```

Find all files that share an inode with this hard link

Reference count: 2

What's in a directory?

```
% ls -pla
drwxr-xr-x  2 brewsteb upg22026   512 Jun 22 16:44 ./
drwxr-xr-x  8 brewsteb ftp       1024 Jun 22 15:46 ../
-rw-r--r--  1 brewsteb upg22026  1027 Jun 22 15:47 cursesDemo.c
-rw-r--r--  1 brewsteb upg22026 42558 Jun 22 15:55 Curses.pdf
-rw-r--r--  1 brewsteb upg22026  4208 Jun 22 16:24 index.html
-rw-r--r--  1 brewsteb upg22026 61554 Jun 22 15:46 IntroToUnixShell.html
-rw-r--r--  1 brewsteb upg22026    38 Jun 22 15:46 OnE1FAQ.txt
-rw-------  1 brewsteb upg22026   467 Jun 22 15:46 OnE1_sol.txt
-rw-r--r--  1 brewsteb upg22026   288 Jun 22 15:46 OnE1.txt
-rw-r--r--  1 brewsteb upg22026    38 Jun 22 15:55 Prog1FAQ.txt
-rw-r--r--  1 brewsteb upg22026  8098 Jun 22 15:46 Prog1.html
-rw-r--r--  1 brewsteb upg22026  7114 Jun 22 15:46 Prog1.test
-rw-r--r--  1 brewsteb upg22026    38 Jun 22 15:46 Prog2FAQ.txt
-rw-r--r--  1 brewsteb upg22026  4517 Jun 22 16:14 Prog2.html
```

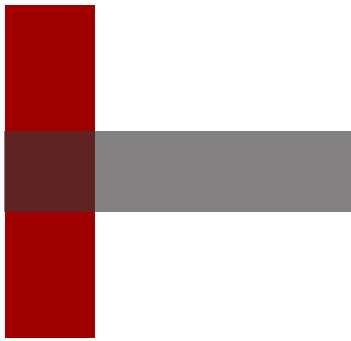


Possibilities include:

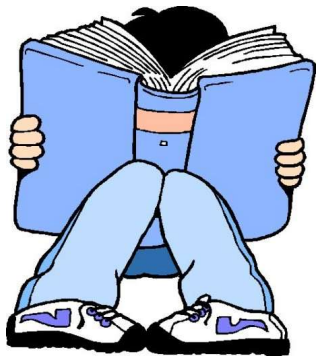
- Hard links
- Symbolic links
- Named pipes
- Device character special file
- Device block special file
- Named socket

Permissions

- Files in UNIX have access permissions for three classes of users:
  - **u**ser (the owner of the file - can set all permissions)
  - **g**roup
  - all **o**thers
- Three kinds of access permissions for each:
  - **r**ead
  - **w**rite
  - E**x**ecute
- Every file belongs to exactly one user and one group



Read Permissions



- File
  - The file's contents can be read
- Directory
  - The directory's contents can be read (ie, a listing of the files in the directory can be returned)

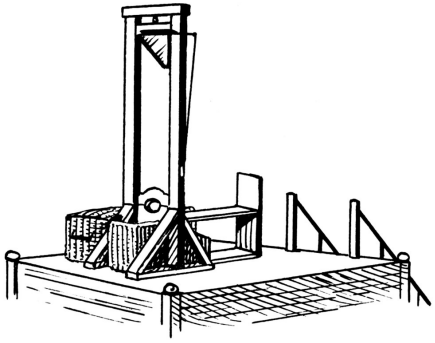
Write Permissions

- File
  - The file can be written to (ie, the contents of the file can be changed)
- Directory
  - Files can be added/removed/renamed/etc. to/in the directory

Execute Permissions

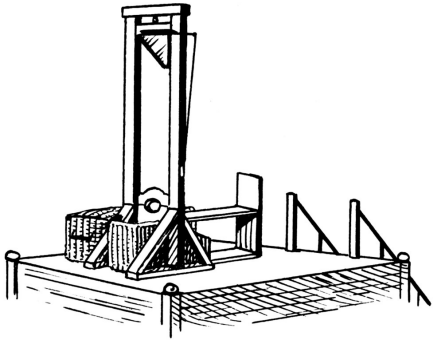
- File
  - The file can be executed (program, shell script)

- Directory
  - The directory can be cd'd into
  - File contents can be listed, and meta-information accessed, if name is known



Execute Permissions

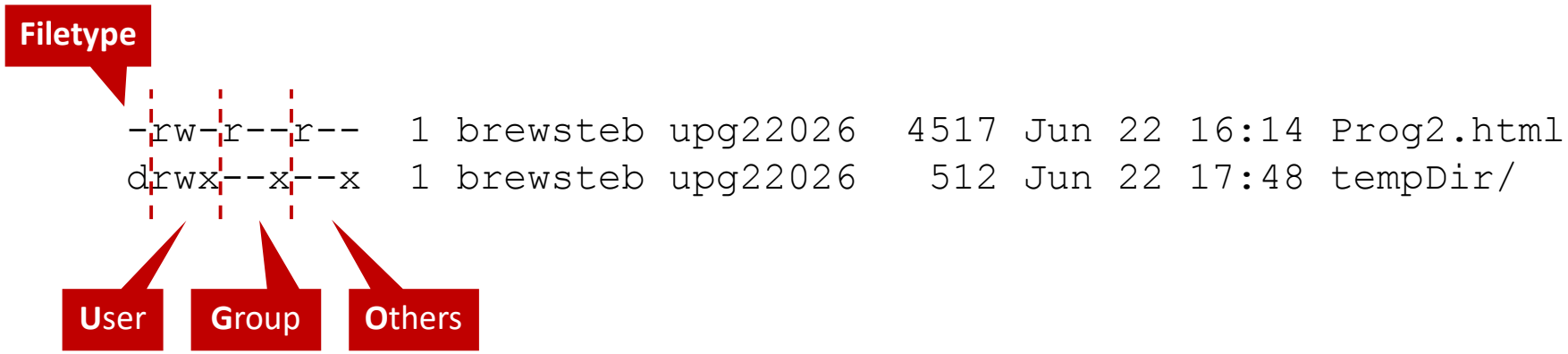
- File
  - The file can be executed (program, shell script)
- Directory
  - The directory can be cd'd into
  - File contents can be listed, and meta-information accessed, if name is known



chmod

- You can change the permissions on a file by using the `chmod` (**ch**ange **mod**e) command

- Here is a sample file listing (generated by `ls -pla`) of a file and dir:



chmod - octal math

- The traditional method of assigning permissions with chmod uses octal
- Why does everything in UNIX have to be so hard?
  - r = 4
  - w = 2
  - x = 1

```
-rw-r--r--   1 brewsteb upg22026  4517 Jun 22 16:14 Prog2.html
drwxr-xr-x   1 brewsteb upg22026   512 Jun 22 17:48 tempDir/
```

4 + 2 + 1 = 7

4 + 0 + 1 = 5

```
$ chmod 644 Prog2.html
```

Standard rights for a publicly viewable **webpage**

```
$ chmod 755 tempDir
```

Standard rights for a publicly viewable **directory**

Setting Permissions the Easy Way

- With this file:

```
----------  1 brewsteb upg22026  4517 Jun 22 16:14 Prog2.html
```

- Set permissions like this:

```
$ chmod u+rwx Prog2.html
-rwx------  1 brewsteb upg22026  4517 Jun 22 16:14 Prog2.html

$ chmod g+rx,o+rwx Prog2.html
-rwxr-xrwx  1 brewsteb upg22026  4517 Jun 22 16:14 Prog2.html

$ chmod o-w Prog2.html
-rwxr-xr-x  1 brewsteb upg22026  4517 Jun 22 16:14 Prog2.html
```

`umask`

- The *creation mask* setting defines the *default permissions* for new files.
- You can set this mask with the UNIX utility `umask`
- If no argument is included, `umask` displays the current setting

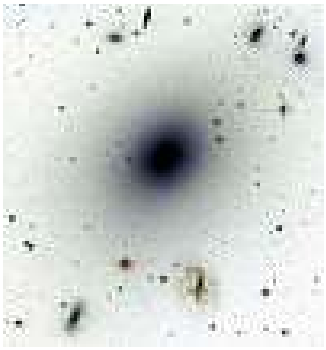
`umask`

- Since this is a mask we're talking about, it's inverted from what we saw with `chmod`

- Set the default permissions on new files to give the owner full privileges, while the group and all others do not have write privileges:

```
$ umask 022
```

- Note that execute permissions still are often not set by default, even if your umask indicates that they should



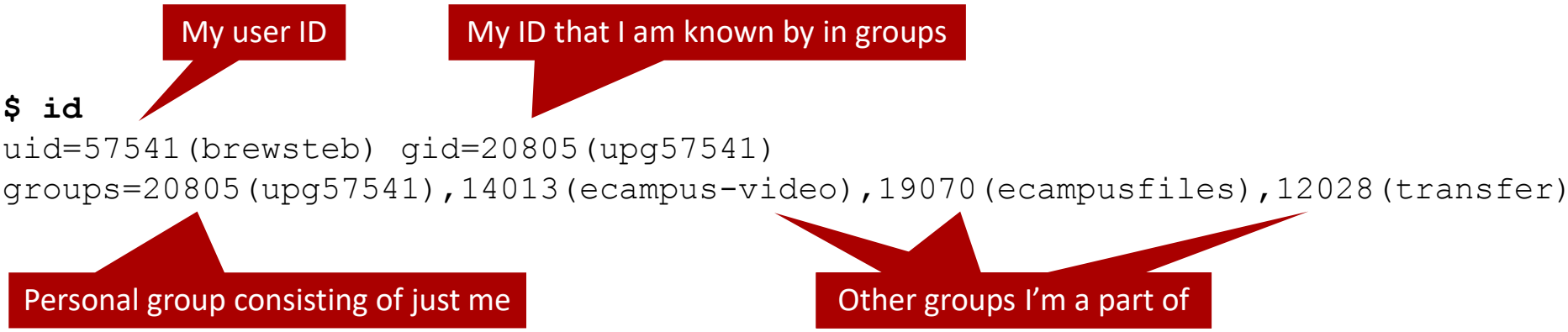
umask

```
$ umask
0007
$ touch tempfile
$ ls -pla tempfile
-rw-rw----. 1 brewsteb upg57541 0 Mar 30 08:17 tempfile
$ rm tempfile
rm: remove regular empty file ‘tempfile’? y
$ umask 022
$ umask
0022
$ touch tempfile
$ ls -pla tempfile
-rw-r--r--. 1 brewsteb upg57541 0 Mar 30 08:18 tempfile
```

Yes, there are actually four digits: the farthest left controls the SUID, SGID, and sticky bits, which are rarely used - you can always leave them off

Note that not all leading zeroes are necessary

What are UNIX groups?



- These groups are the same groups referred to when using the `chmod` command

Common UNIX Commands - Directories & Files

- **du** - Returns the total usage in kilobytes of the specified directory

```
$ du .
4       ./inodetest/1
4       ./inodetest/2
16      ./inodetest
8       ./permissionstests
96      .
```

Common UNIX Commands - Directories & Files

- **`df`** - Returns the total usage in kilobytes of filesystems

```
$ pwd
/nfs/stak/faculty/b/brewsteb/tempdir
$ df .
Filesystem                          1K-blocks        Used  Available  Use%  Mounted on
128.193.40.234:/stak_faculty/data  2147483648  1404256256  743227392   66%  /nfs/stak/faculty
```

Common UNIX Commands - Directories & Files

- **`stat`** - Get details about a file

```
$ stat testfile
  File: ‘testfile’
  Size: 9               Blocks: 8          IO Block: 262144 regular file
Device: 35h/53d Inode: 3238033175  Links: 1
Access: (0000/----------)  Uid: (57541/brewsteb)   Gid: (20805/upg57541)
Context: system_u:object_r:nfs_t:s0
Access: 2016-08-30 09:38:11.386951000 -0700
Modify: 2016-08-30 09:40:01.075970000 -0700
Change: 2016-08-30 09:40:11.156727000 -0700
 Birth: -
$ touch testfile
$ stat testfile
  File: ‘testfile’
  Size: 9               Blocks: 8          IO Block: 262144 regular file
Device: 35h/53d Inode: 3238033175  Links: 1
Access: (0000/----------)  Uid: (57541/brewsteb)   Gid: (20805/upg57541)
Context: system_u:object_r:nfs_t:s0
Access: 2016-08-30 11:19:33.386465000 -0700
Modify: 2016-08-30 11:19:33.386465000 -0700
Change: 2016-08-30 11:19:33.386480000 -0700
 Birth: -
```

Timestamp types

- **Access:** When file was last read
- **Modify:** When file contents were last modified
- **Change:** When meta data about file was last changed (renaming, permissions, etc.)
- **Birth:** Creation date of file (unsupported on Linux, but works on BSD & Windows)

Common UNIX Commands - Directories & Files

- **touch** - Create files and modify time stamps

```
$ ls -l
$ touch testfile
$ ls -l
-rw-rw----. 1 brewsteb upg57541 0 Aug 30 12:02 testfile
$ echo "testtext" > testfile
$ ls -l
-rw-rw----. 1 brewsteb upg57541 9 Aug 30 12:02 testfile
$ touch -d 20120101 fakefile
$ ls -l
-rw-rw----. 1 brewsteb upg57541 0 Jan  1  2012 fakefile
-rw-rw----. 1 brewsteb upg57541 9 Aug 30 12:02 testfile
$ touch -r fakefile testfile
$ ls -l
-rw-rw----. 1 brewsteb upg57541 0 Jan  1  2012 fakefile
-rw-rw----. 1 brewsteb upg57541 9 Jan  1  2012 testfile
```

- Security warning: this is how easy it is to modify the Access and Modify values!
- Only the “Change” value remains unchanged
- You can also arbitrarily update “Change” by setting the system time (with root) to whatever you want, running these commands, then changing the time back

Standard Directories

- Root dir:
  - `/`
- Home dir:
  - `~`
- Bad idea:
  - `rm -rf /*`

Easier to delete everything than you might think!

If you want to delete all of the files in your current directory, and all directories underneath, the command is:

$ rm -rf ./*

I know because I've done it. :(

The Vicious Interface

Benjamin Brewster

If It's So Vicious and Old, Why Study vi?

- History
  - Understanding why it exists and why it was created informs us about the underlying OS (UNIX) and the language it was developed in (C)
- Power
  - There are LOTS of things you can do in vi you can't do anywhere else
  - Important for manipulating large data files, repetitive commands, etc.
- Ubiquity
  - Installed on every UNIX and UNIX-like system!
- Necessity
  - Sometimes you'll have no other options because of the environment

Text Editors

- There are many text editors available on UNIX
  - ed (a line editor only)
  - ex (an extended line editor; vi's predecessor)
  - emacs

- vi was written by Bill Joy in 1976 for BSD
  - Added full-screen visibility to ex

- Its name comes from the shortest unambiguous abbreviation of **vi**sual

What's a line editor?

Image by user Chlor~enwiki, CC SA 3.0

"[ed is] the most user-hostile editor ever created."
-Peter H. Salus, computer historian

```
$ ed
a
ed is the standard Unix text editor.
This is line number two.
.
2i

.
,l
ed is the standard Unix text editor.$
$
This is line number two.$
3s/two/three/
,l
ed is the standard Unix text editor.$
$
This is line number three.$
w edfile
65
Q
$ cat edfile
ed is the standard Unix text editor.

This is line number three.
```

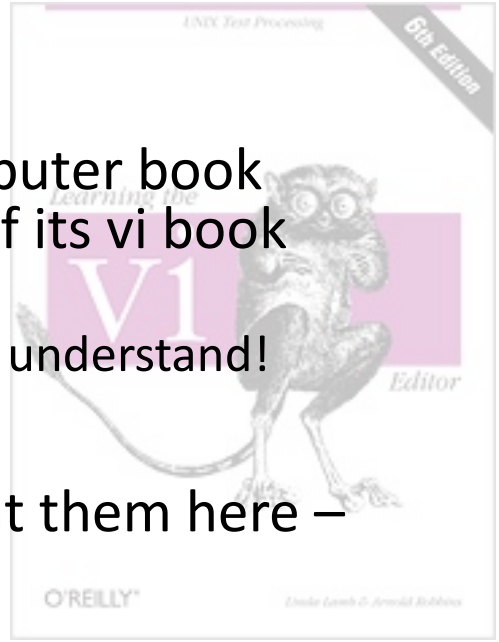
Script courtesy of Wikipedia

vi = Vicious Interface

- Just because it’s got a full-screen interface, it doesn’t mean it’s easy to use - but it is very powerful!

- In 1999, Tim O'Reilly, founder of the eponymous computer book publisher, stated that his company sold more copies of its vi book than its emacs book…
  - Not because more people *like* vi, but because it’s harder to understand!

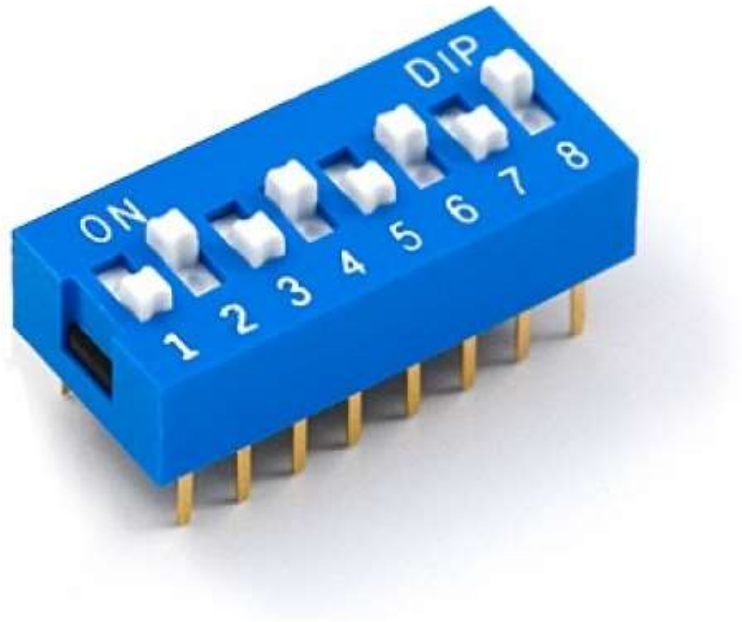
- Don’t try to memorize all of the keystrokes as I present them here – just be aware they exist!



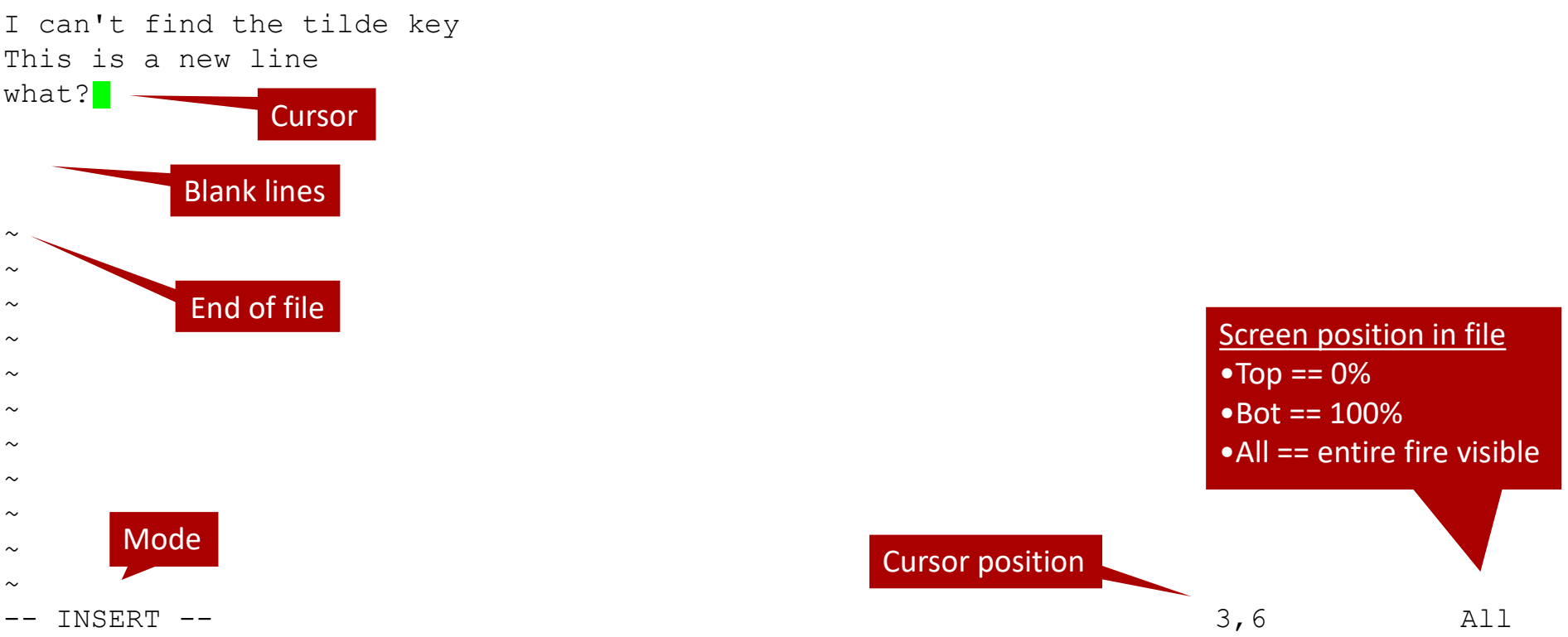
Modes, modes, modes

- vi features one of the first visual environments, instead of line editors

- Primary paradigm: vi is modal
  - Normal mode
  - Insert mode
  - Command mode
  - Visual mode
  - and a few others



A Visual View of vi



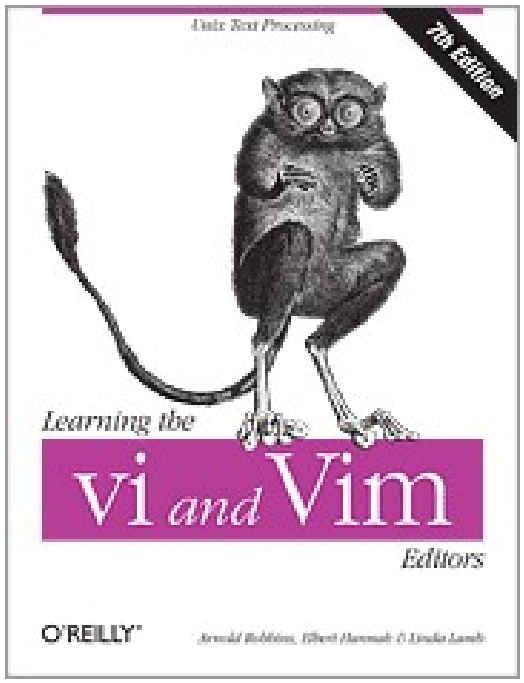
Actually, that was vim

- vim is **vi** **im**proved
  - Better than vi, though the basic commands we're covering work in both

- vim is:
  - Still in development
  - vi is often mapped to simply start vim

- Starting vim

```
$ vim newFile
$ vim existingFile
```



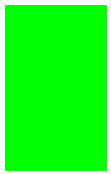
Modes for Real

Normal

- Move around the document
- Perform one-shot edit commands on characters, paragraphs, or even larger blocks of text

Insert

- Insert text into the document
- What normal WYSIWYG editors can only do



Modes for real

Replace

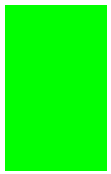
- Overwrite mode

Visual

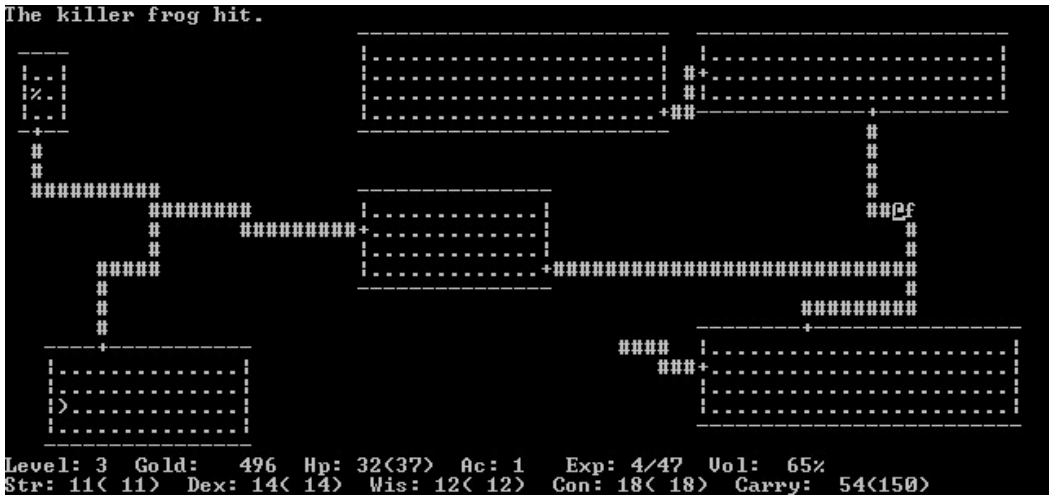
- Selects an area of text to which subsequent commands can be applied

Command

- Whole file commands
  - Save, quit, search, etc.



Normal Mode



- Movement
  - Cursor movement
    - `h,j,k,l` - the Rogue keyset
    - `$` – move to the end of the line
    - `0` – move to the beginning of the line
    - `w` – move to beginning of next word to the right
    - `b` - move back to beginning of the previous word on the left
  - Screen movement
    - `^d` – move screen down half a screen
    - `^u` – move screen up half a screen

Normal Mode

- How to get into Normal Mode
  - vi starts in Normal Mode
  - Hit escape
    - You can always hit escape – the key can never do anything but take you to Normal mode
    - In fact, hit it a bunch of times
    - Will beep if you're in Normal mode already

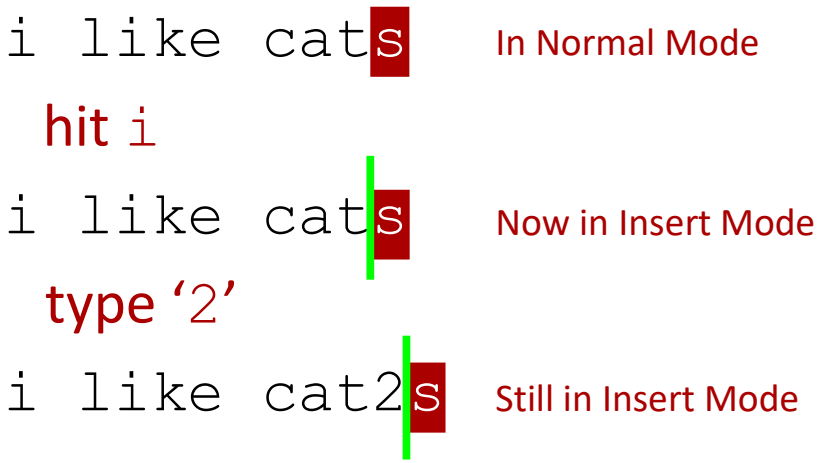
Insert Mode

- Type like normal

- Move around with the arrow keys
  - Commands (including movement commands) from Normal mode will not work - you get characters instead

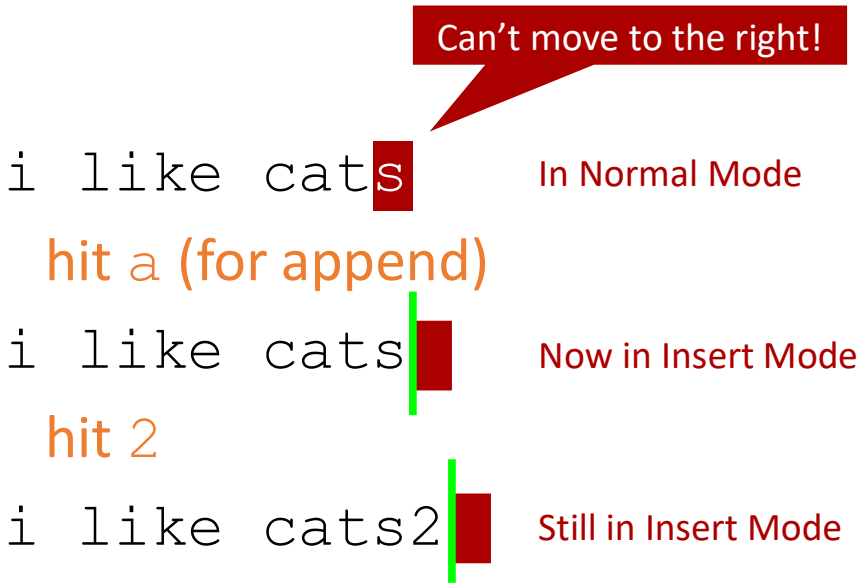
Insert Mode

- How to get into insert mode
  - From Normal mode, hit **`i`**



Insert Mode

- In this situation, use `a`:



Replace

Like when you accidentally hit the Insert key in Word

- Overwrite mode
  - Non-insertion typing

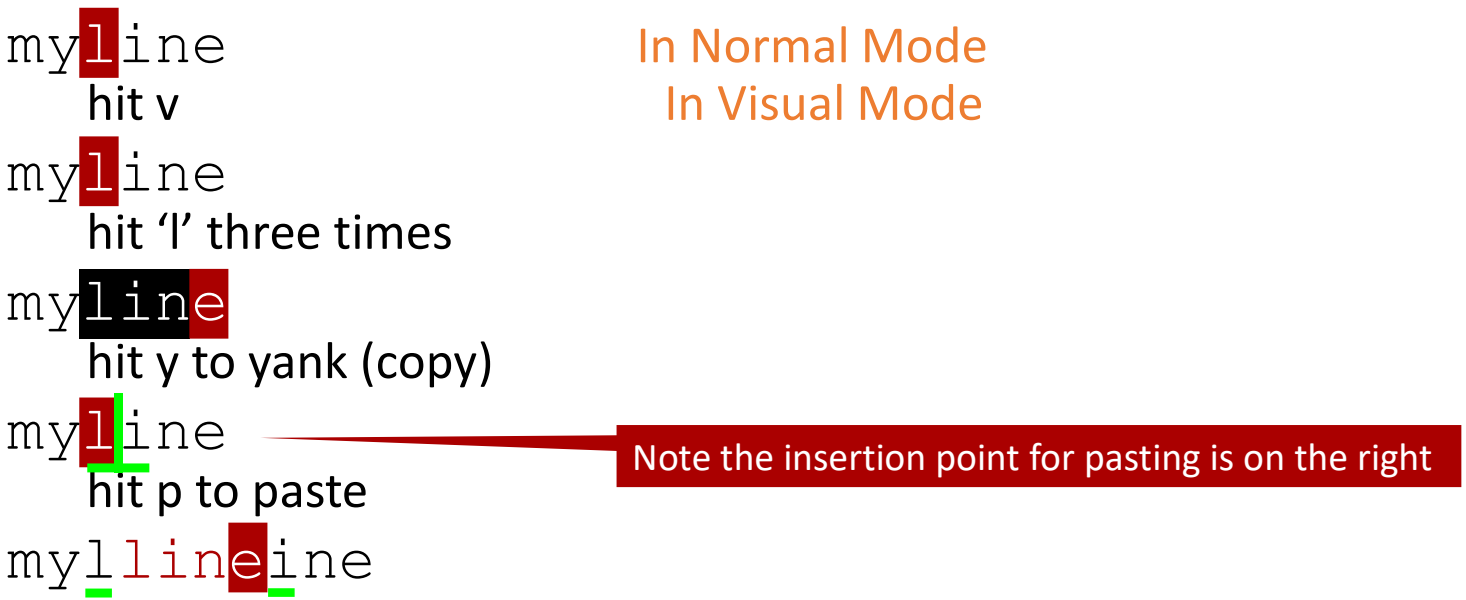
- Two ways to get into Replace Mode
  - `r` – replace the character that the cursor is over with the next one typed
  - `R` – enter Replace Mode until we manually leave it (eg, by escape back to Normal Mode)

Visual Mode

- Visual mode allows you to select text and then apply commands to it
- What you have selected is marked by inverted characters

Visual Mode Demo

- Let's cut, copy, and paste:



Command Mode

- Used to enter commands that pertain (mostly) to the entire file
- These commands are actually carried over from the line editor ed!
- To **save** your file, enter command mode (:), hit w, then enter:

```
:w
```

- If you started vi without a filename, you'll have to type in a name and then hit enter:

```
:w myNewFileName
```

- Can also be used to save a *copy*, but you'll still be editing the *original* file:

```
:w thesis_backupcopy
```

Quitting vi

- Quit:

  `:q`
- Save, and then quit

  `:wq`
- To exit without saving:

  `:q!`
- From Normal Mode, you can save the current file and exit immediately:

  `ZZ`

Note the lack of colon here

Search and Replace

- To find a string pattern in the file:

  `/pattern`

  `n` will move you to the next instance of that `pattern`

  `N` will move you to the previous instance of that `pattern`

Note the lack of colon here

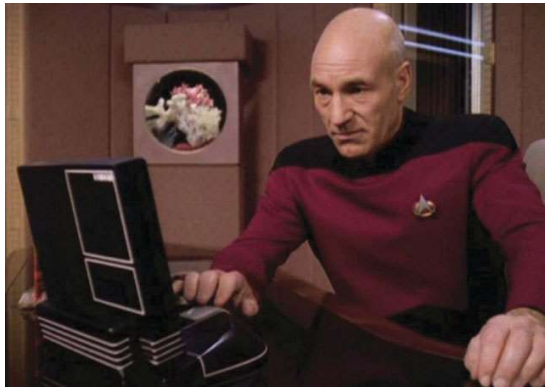
- Remove highlighting after search:

  `:nohl`

- Global search and replace:

  `:%s/wrongtext/righttext/g`

Advanced Command Mode



- Run a single UNIX command (from inside vi):

  `:!`*`UNIXCOMMAND`*

- Run a single UNIX command and insert the output into the file:

  `:r!`*`UNIXCOMMAND`*

- Put vi in the background, and start a new shell in the foreground (defaults to what is in your SHELL environment variable):

  `:sh`

Back to (Advanced) Normal Mode

- cut

  In visual mode, use `d` instead of `y`

- Delete/cut a line

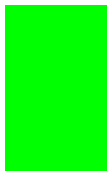
  `dd`

- Copy the current line

  `yy`

- Undo the last Normal Mode command

  `u`



Advanced Normal Mode

- Delete the current character

  `x`

- Delete the current word

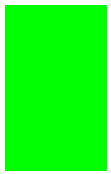
  `dw`

- Transpose current and next char

  `xp`

- Go into Append/Insert Mode at the end of the line

  `A`



Advanced Normal Mode

- Open new line above the current line in Insert Mode
  `O (big oh)`
- Open new line below the current line in Insert Mode
  `o (little oh)`
- Delete the rest of the line from where the cursor is
  `d$`
- Delete the current char and enter Insert Mode
  `s`

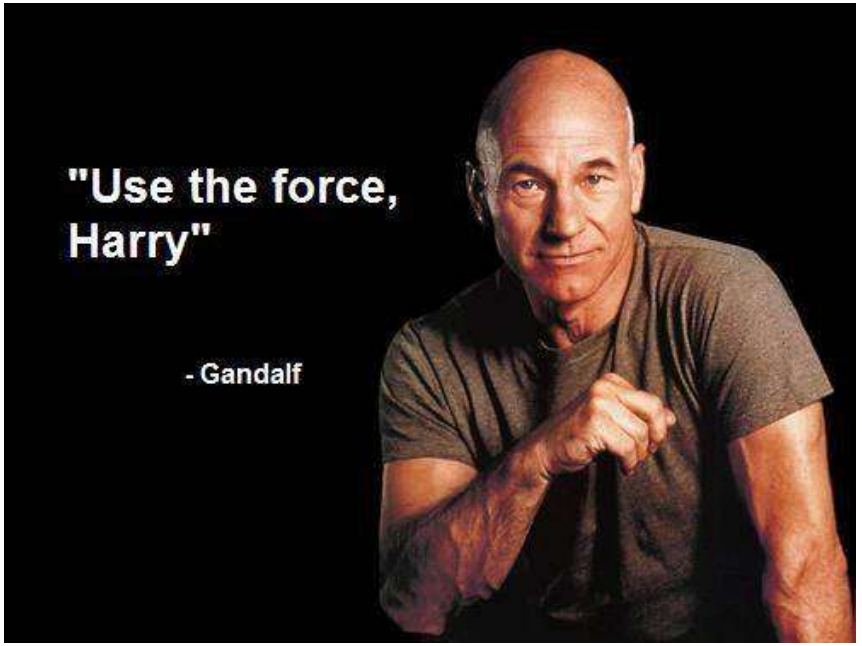
Advanced Normal Mode

- Join two lines

```
firstlineALLONELINEfirstline
ALLONELINE
secondlineALLONELINEsecondli
neALLONELINE
~
```

Hit `J`

```
firstlineALLONELINEfirstline
ALLONELINE secondlineALLONEL
INEsecondlineALLONELINE
~
```



Advanced Normal Mode

- Delete the five lines starting at the current cursor position

```
5dd
```

- You can find further goofiness online on the “vi Resources” page
  - Like ~ which switches the case of a letter

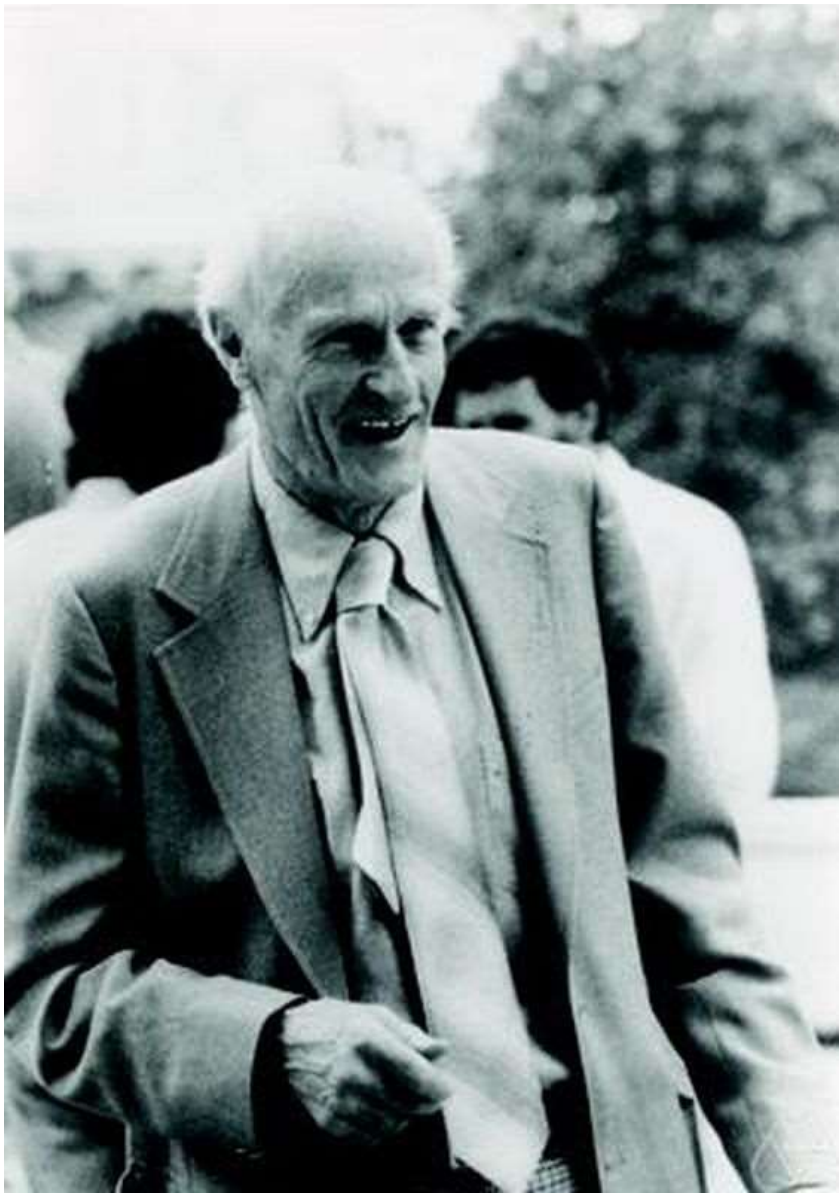
Regular Expressions in UNIX

Benjamin Brewster

Regular Expressions

- Regular expressions are a way to specify a pattern of strings that you'd like returned as part of a search

- Invented by Stephen Kleene in the 1950s



Regular Expressions

- REs are used by many UNIX programs:
  - grep, sed, vi, emacs, regexp, etc.
- Used extensively by many scripting languages:
  - Python, Perl, Tcl/Tk
- There is an entire course (CS321) that goes over REs, and other grammars

Common UNIX Commands - Filtering with grep

- **grep** - search for an occurrence of a string that matches a pattern

```
$ cat fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
fifth line
sixth lFINDMEine
$ grep "FINDME" fileToSearch
FINDME first line
second FINDME line
third line FINDME
sixth lFINDMEine
```

Another `grep` example

`ps` returns a list of processes

```
$ ps -ef | grep brewsteb
root     29541 3760  0 11:26 ?     00:00:00 sshd: brewsteb [priv]
brewsteb 29543 29541 0 11:26 ?     00:00:00 sshd: brewsteb@pts/1
brewsteb 29544 29543 0 11:26 pts/1 00:00:00 -csh
brewsteb 30737 29544 0 11:44 pts/1 00:00:00 ps -ef
brewsteb 30738 29544 0 11:44 pts/1 00:00:00 grep brewsteb
```

Basic REs - Operators

***** (asterisk) – Matches 0 or more of the *previous character*

- *Warning – this is different than Windows and UNIX command line usage!*
- Known as the Kleene Star in the regular grammar field

^ (circumflex) – When placed at the beginning of a RE, indicates the RE must start at the beginning of the string

$ (dollar sign) – When placed at the end of an RE, matches the end of the string

The Asterisk – 0 or more

Pattern	**Matches**
A*	A or AA or AAA or …
Ab*	Ab or Abb or Abbb or …
FINDME*	FINDME or FINDMEE or FINDMEEE…

Note: ***Not*** FINDMEFINDME or …

Binding to Beginning and End

- Unless you use the ^ and $ operators, a RE will match substrings

Pattern	**Matches**
Jon	Will match any string that contains Jon anywhere
^abc	Any string that *starts* with abc
XYZ$	Any string that *ends* with XYZ
^Ben Brewster$	Any string that matches “Ben Brewster” *exactly*

Single Character Matching

- The following operators are available:

.	Matches any single character
\	Causes the following special character to simply be itself
	Like the period character itself
[abc]	Matches *any one* character inside the brackets
[^abc]	Matches any character *except any* of the ones inside

- Any other non-special character matches itself

Period Example 1

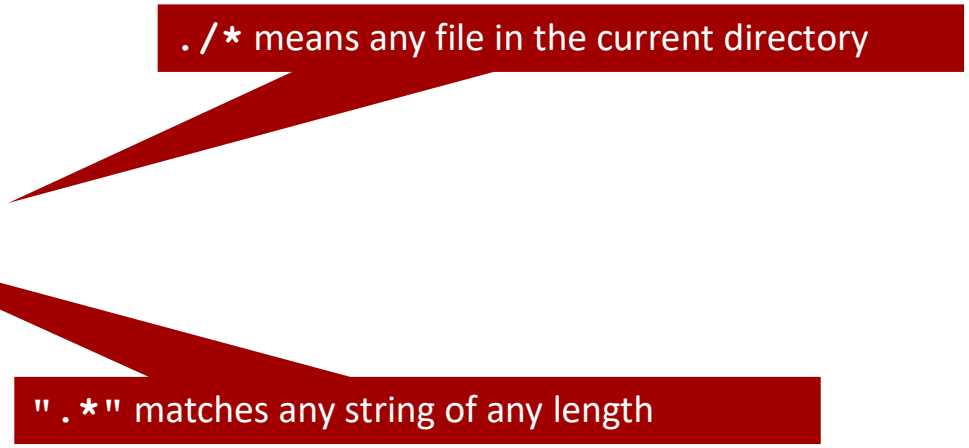
```
$ cat fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
fifth line
sixth lFINDMEine
$ grep "FINDM." fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
sixth lFINDMEine
```

Period Example 2 - Two Different Asterisks

.* means match any char any number of times

- This is the “anything, any length” wildcard

```
$ cat datafile
abcdefghi
abcdef
ghi
aabbccddee
$ grep ".*" ./*
abcdefghi
abcdef
ghi
aabbccddee
…
```



Backslash Example

- The backslash causes the REs to literally interpret special characters

Pattern	**Matches**
\.	.
\$	$
*	*

Brackets Example 1

```
$ cat fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
fifth line
sixth lFINDMEine
$ grep "FINDM[E3]" fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
sixth lFINDMEine
```

Brackets Example 2

```
$ cat fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
fifth line
sixth lFINDMEine
$ grep "FINDM[^3]" fileToSearch
FINDME first line
second FINDME line
third line FINDME
sixth lFINDMEine
```

Brackets Example 3

```
$ cat fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
fifth line
sixth lFINDMEine
$ grep "[^3]" fileToSearch
FINDME first line
second FINDME line
third line FINDME
fourth line FINDM3
fifth line
sixth lFINDMEine
```

Ranges

- When using the square brackets [ ], you can specify ranges of characters to match
- The proper ordering is defined by the ASCII character set:
  - http://www.asciitable.com/

Pattern	**Matches**
[a-z]	a b c d e f … y z
[^a-z]	Anything but the characters a-z

Or: |

```
$ cat catsdogs
i like cats
i like dogs
i like catdogs
i like dogsdogs
$ grep "cat|dog" catsdogs
$ grep "cat\|dog" catsdogs
i like cats
i like dogs
i like catdogs
i like dogsdogs
$ grep "i like \(cat\|dog\)" catsdogs
i like cats
i like dogs
i like catdogs
i like dogsdogs
```

Note that we parenthesize the 'or', here, and that each syntax symbol is escaped

Matching a Repeated Pattern

- We can search for a pattern that is repeated at least once

```
$ cat catsdogs
i like cats
i like dogs
i like catdogs
i like dogsdogs
$ grep "\(dogs\)\1" catsdogs
i like dogsdogs
```

Matching a Repeated Pattern

- Curly braces specify the number of repeats (at least) that we're looking for to register a match

```
$ cat digs
dig
digdig
digdigdig
digdigdigdig
digdigdigdigdig

$ grep "\(dig\)\{3\}" digs
digdigdig
digdigdigdig
digdigdigdigdig
```

Backreferences

`\(  \)` (parentheses) These operators will capture a matched string for later use

`\1, \2, etc.` (escaped integer) This allows you to specify that the string should match the *n*th pattern that you have previously captured, where *n* is the number following the backslash

Backreference Example

```
$ cat likes
You like You
I like You
$ grep "\(You\) like \1" likes
You like You
```

Backreference

- Note that the repeat example from earlier is actually a backreference:

```
$ grep "\(dogs\)\1" catsdogs
i like dogsdogs
```

Repeat

Finding Things - `grep`

- Find all lines in all files that match a string:

r :: Search recursively down into each subdirectory
n :: Return the line number of matching lines

Look everywhere from my home directory and deeper

```
$ grep -rn "dogsdogs" ~
/nfs/stak/faculty/b/brewsteb/tempdir/greptests/catsdogs:4:i like dogsdogs
```

Finding Things - `find` - Example 1

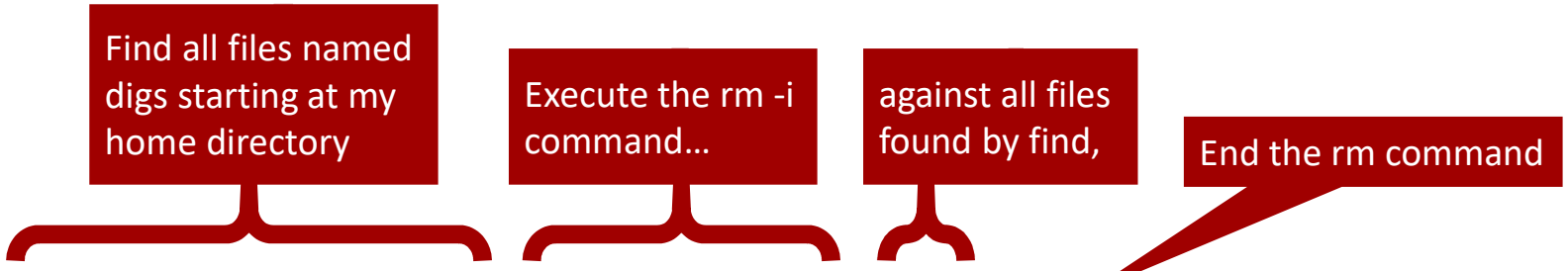
- Find a file by name and many other features
- Can also execute commands against the files found(!)

Find all files named digs starting at my home directory (recursion into directories is implied)

```
$ find ~ -name digs
/nfs/stak/faculty/b/brewsteb/tempdir/greptests/digs
```

Finding Things - `find` - Example 2

- Find a file by name and many other features
- Can also execute commands against the files found(!)




```
$ find ~ -name digs -exec rm -i '{}' \;
rm: remove regular empty file
'/nfs/stak/faculty/b/brewsteb/tempdir/greptests/digs'? y
```

Finding Things - `find` - Example 3

- Find a file by name and many other features
- Find based on regular expression

```
$ find ~ -regex ".*fork.*" | sort
/nfs/stak/faculty/b/brewsteb/tempdir/doublefork
/nfs/stak/faculty/b/brewsteb/tempdir/doublefork.c
/nfs/stak/faculty/b/brewsteb/tempdir/forkexec
/nfs/stak/faculty/b/brewsteb/tempdir/forkexec.c
/nfs/stak/faculty/b/brewsteb/tempdir/forkFPsharing
/nfs/stak/faculty/b/brewsteb/tempdir/forkFPsharing.c
/nfs/stak/faculty/b/brewsteb/tempdir/forktest
/nfs/stak/faculty/b/brewsteb/tempdir/forktest.c
/nfs/stak/faculty/b/brewsteb/tempdir/forkwaittest
/nfs/stak/faculty/b/brewsteb/tempdir/forkwaittest.c
/nfs/stak/faculty/b/brewsteb/tempdir/forkyouzombie
/nfs/stak/faculty/b/brewsteb/tempdir/forkyouzombie.c
/nfs/stak/faculty/b/brewsteb/tempdir/pipeNfork
/nfs/stak/faculty/b/brewsteb/tempdir/pipeNfork.c
/nfs/stak/faculty/b/brewsteb/tempdir/pipeNfork_directEdit.c
/nfs/stak/faculty/b/brewsteb/tempdir/pipeNforkFIFO
/nfs/stak/faculty/b/brewsteb/tempdir/pipeNforkFIFO.c
```

Finding Things - `locate`

- Finds files using a database
- Faster than find, since it doesn't actually search the directory hierarchy for the indicated files
- Will be outdated since last file location check!
- Not available on every system
- Operates differently on those systems that do have it, due to different versions being installed on different Operating Systems
- …just use find, which is ancient and universal