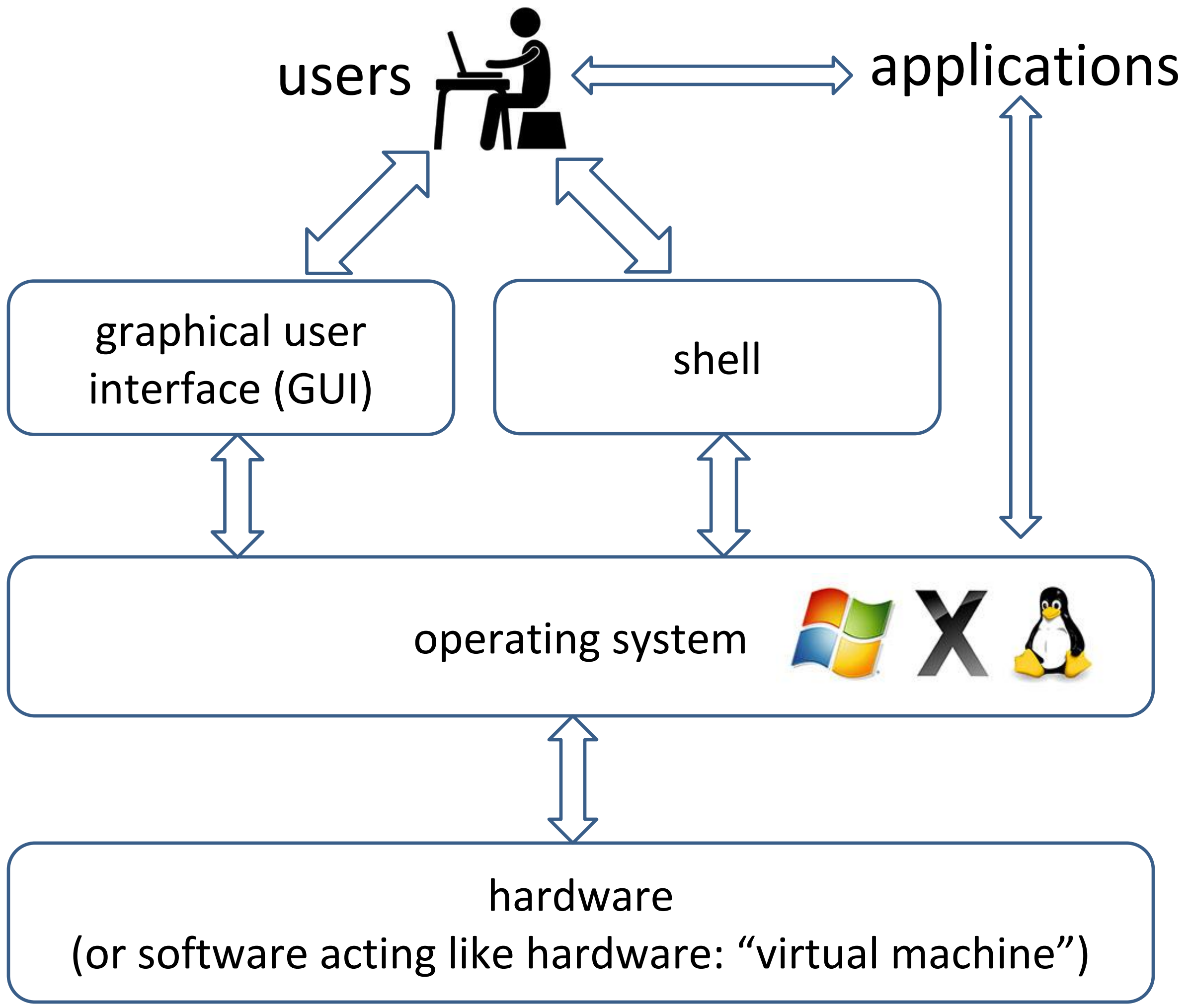
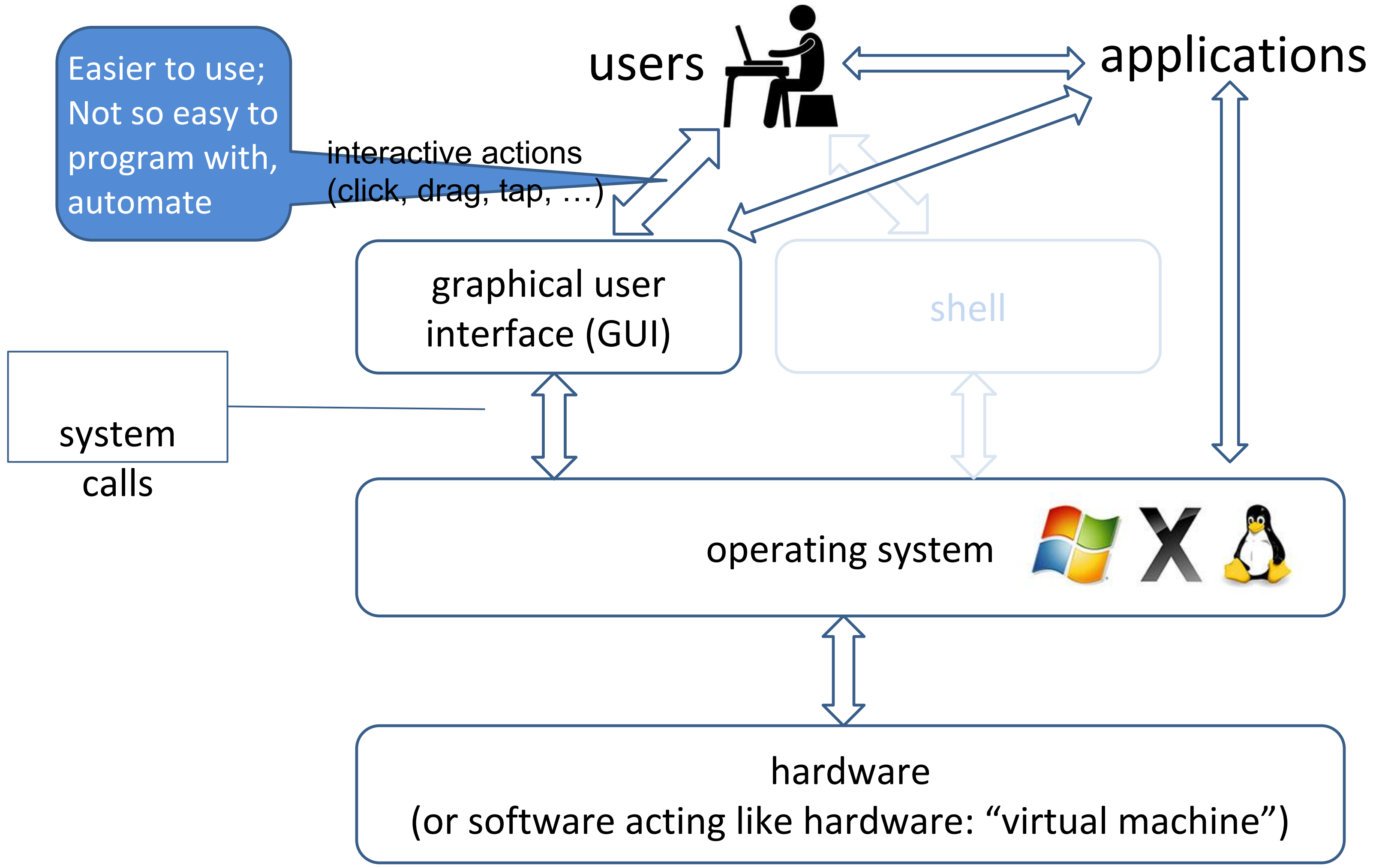
Command Line Interface (Shell)

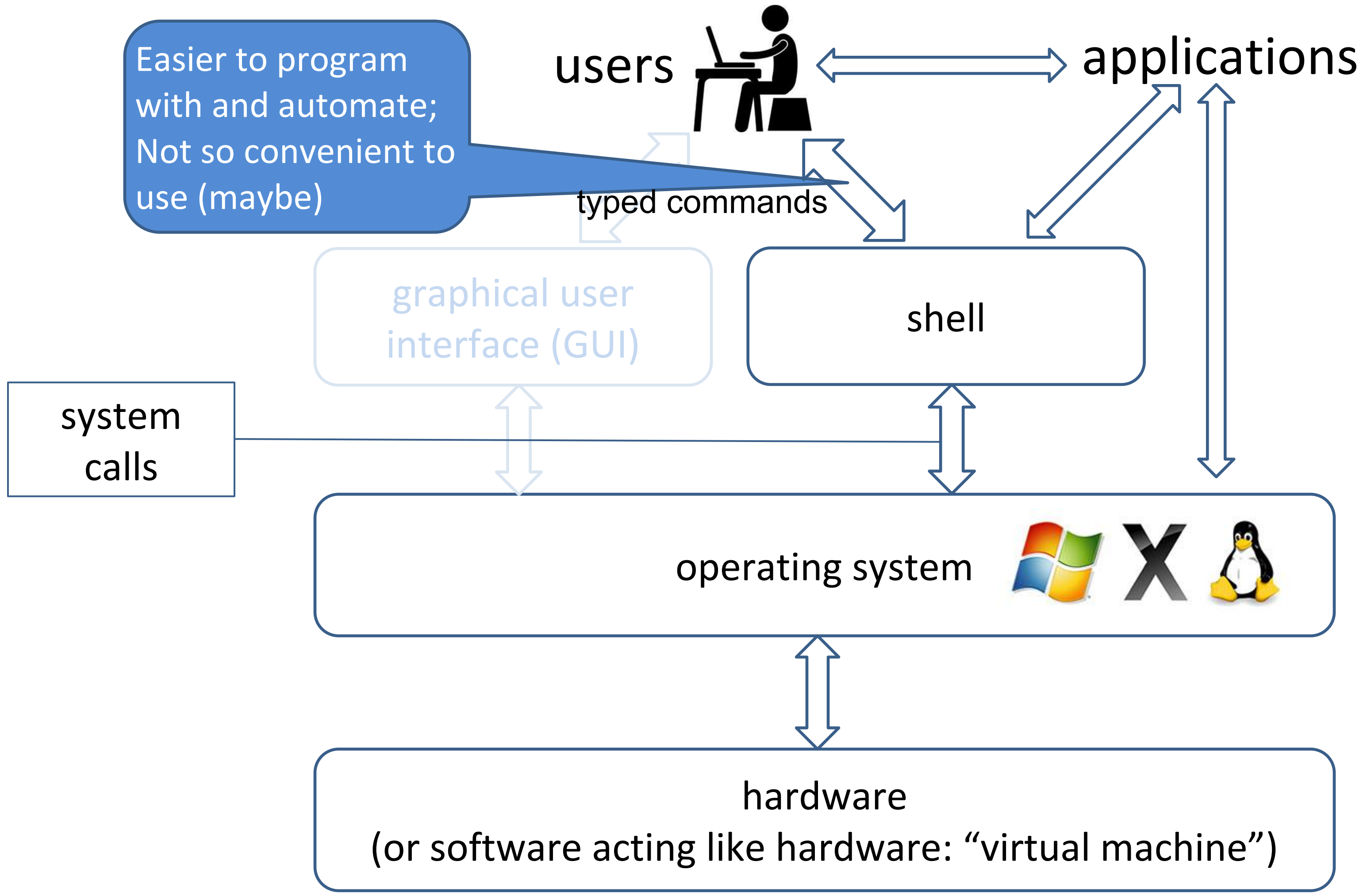
Organization of a computer system


users
applications
graphical user interface (GUI)
shell
operating system
hardware
(or software acting like hardware: “virtual machine”)

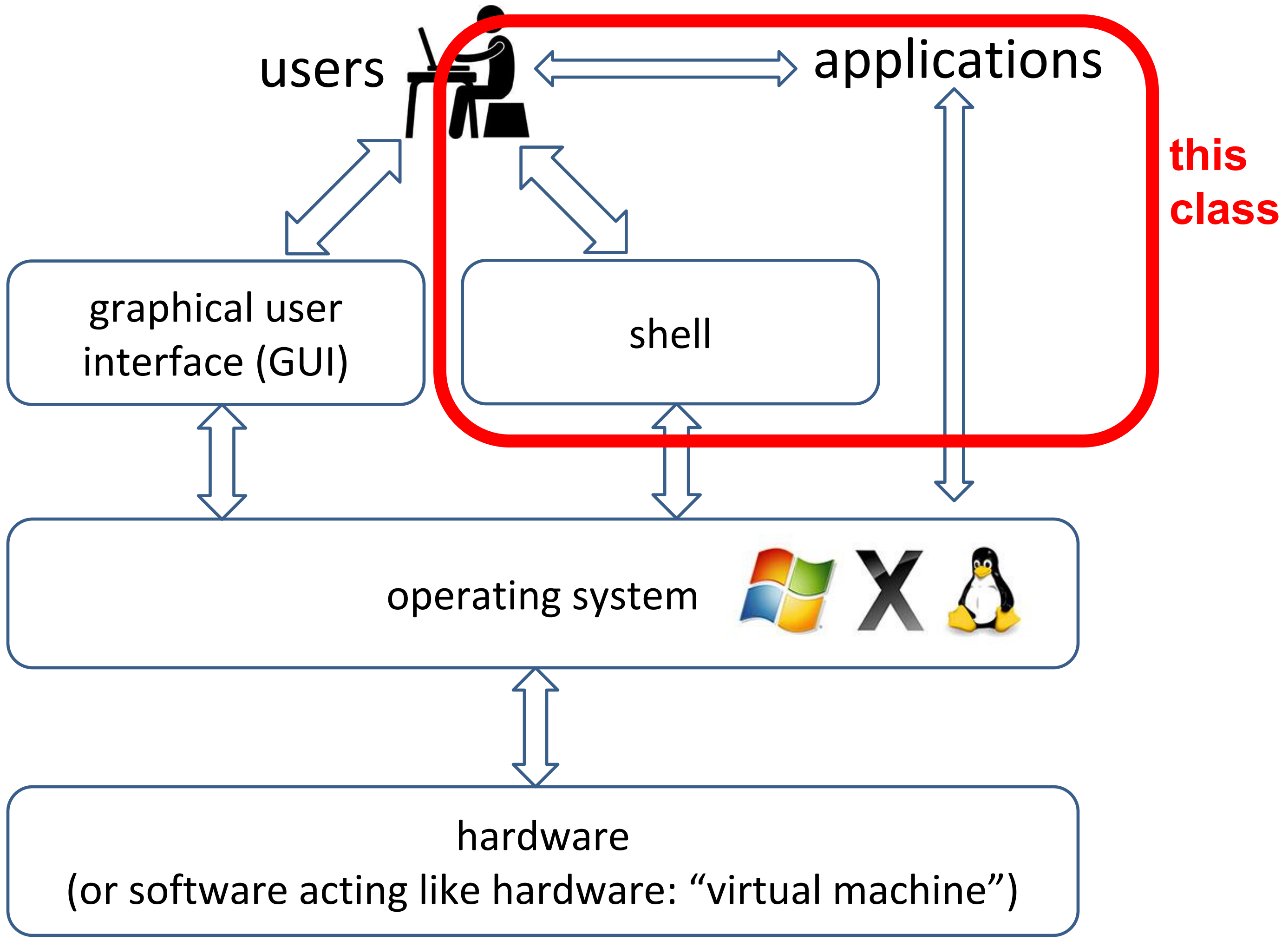
Organization of a computer system


Easier to use;
Not so easy to program with, automate
users
applications
interactive actions
(click, drag, tap, …)
graphical user interface (GUI)
shell
system calls
operating system
hardware
(or software acting like hardware: “virtual machine”)

Organization of a computer system


Easier to program with and automate; Not so convenient to use (maybe)
users
applications
typed commands
graphical user interface (GUI)
shell
system calls
operating system
hardware
(or software acting like hardware: "virtual machine")

Organization of a computer system



What is a Command Line Interface?

- Interface: Means it is a way to interact with the Operating System.

What is a Command Line Interface?

- **Interface**: Means it is a way to interact with the Operating System.
- **Command Line**: Means you interact with it through typing commands at the keyboard.

What is a Command Line Interface?

- **Interface**: Means it is a way to interact with the Operating System.
- **Command Line**: Means you interact with it through typing commands at the keyboard.

So a Command Line Interface (or a shell) is a program that lets you interact with the Operating System via the keyboard.

Why Use a Command Line Interface?

A. In the old days, there was no choice

Why Use a Command Line Interface?

A. In the old days, there was no choice
    a. No commercial computer had a GUI until Apple released the Lisa in 1983 (at $10, 000!!!)

Why Use a Command Line Interface?

A. In the old days, there was no choice
    a. No commercial computer had a GUI until Apple released the Lisa in 1983 (at $10, 000!!!)
    b. There might still be no choice if you are interacting with a computer via a non-graphical terminal.

Why Use a Command Line Interface?

A. In the old days, there was no choice
   a. No commercial computer had a GUI until Apple released the Lisa in 1983 (at $10, 000!!!)
   b. There might still be no choice if you are interacting with a computer via a non-graphical terminal.

B. Many tasks are faster than in a GUI
   a. Suppose you wanted to see all the files in a directory that had the word “lecture” in their name.

Why Use a Command Line Interface?

A. In the old days, there was no choice
   a. No commercial computer had a GUI until Apple released the Lisa in 1993 (at $10, 000!!!)
   b. There might still be no choice if you are interacting with a computer via a non-graphical terminal.

B. Many tasks are faster than in a GUI
   a. Suppose you wanted to see all the files in a directory that had the word “lecture” in their name.

C. Most shells let you write scripts (programs) to automate complex tasks which you could not do with a GUI

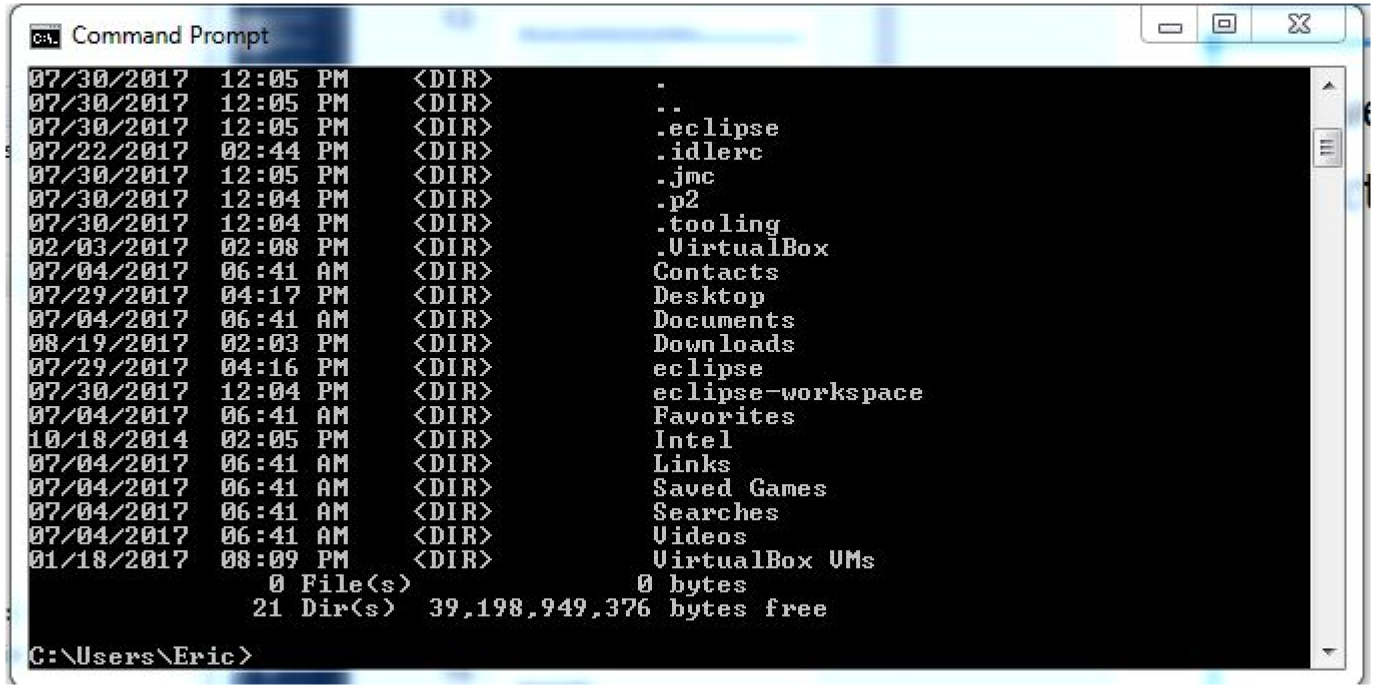
Three Different “Shells”

There are (and have been) 100’s (maybe thousands) of shells. We will briefly mention 3 of them still in use:

1. Command Prompt
2. Windows Powershell
3. Bash

Command Prompt

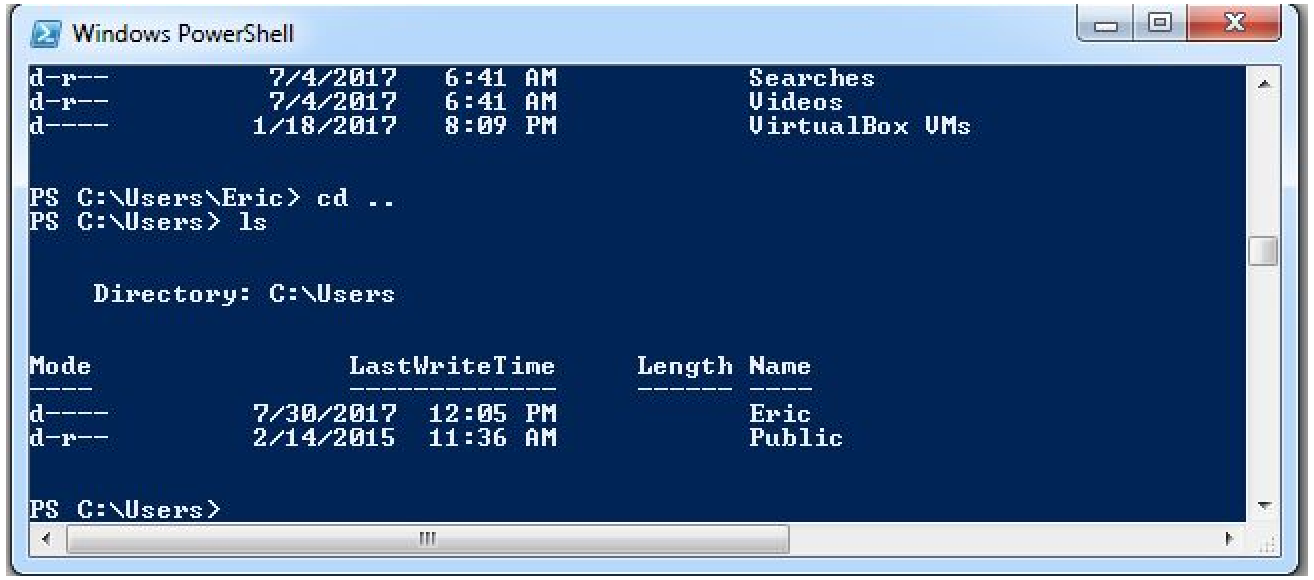
All versions of Windows have included a Command Prompt program. It acts like a MSDOS interface to the computer.



```
07/30/2017  12:05 PM    <DIR>          .
07/30/2017  12:05 PM    <DIR>          ..
07/30/2017  12:05 PM    <DIR>          .eclipse
07/22/2017  02:44 PM    <DIR>          .idlerc
07/30/2017  12:05 PM    <DIR>          .jmc
07/30/2017  12:04 PM    <DIR>          .p2
07/30/2017  12:04 PM    <DIR>          .tooling
02/03/2017  02:08 PM    <DIR>          .VirtualBox
07/04/2017  06:41 AM    <DIR>          Contacts
07/29/2017  04:17 PM    <DIR>          Desktop
07/04/2017  06:41 AM    <DIR>          Documents
08/19/2017  02:03 PM    <DIR>          Downloads
07/29/2017  04:16 PM    <DIR>          eclipse
07/30/2017  12:04 PM    <DIR>          eclipse-workspace
07/04/2017  06:41 AM    <DIR>          Favorites
10/18/2014  02:05 PM    <DIR>          Intel
07/04/2017  06:41 AM    <DIR>          Links
07/04/2017  06:41 AM    <DIR>          Saved Games
07/04/2017  06:41 AM    <DIR>          Searches
07/04/2017  06:41 AM    <DIR>          Videos
01/18/2017  08:09 PM    <DIR>          VirtualBox VMs
               0 File(s)              0 bytes
              21 Dir(s)  39,198,949,376 bytes free

C:\Users\Eric>
```

Windows Powershell

Windows Powershell was an improved shell for Windows first released in 2006. The latest version came out in 2016.



```
d-r--          7/4/2017   6:41 AM            Searches
d-r--          7/4/2017   6:41 AM            Videos
d----         1/18/2017   8:09 PM            VirtualBox VMs


PS C:\Users\Eric> cd ..
PS C:\Users> ls


    Directory: C:\Users


Mode                LastWriteTime     Length Name
----                -------------     ------ ----
d----         7/30/2017  12:05 PM            Eric
d-r--         2/14/2015  11:36 AM            Public


PS C:\Users>
```

Bash and UNIX

In this class we will be learning a little bit about the Bash shell, which is currently the most popular shell used on the UNIX family of systems.

This is NOT a UNIX class, and we will not go into UNIX in any depth, but we will talk about Bash and some common UNIX commands.

What is Unix?

- Unix is an operating system
  - sits between the hardware and the user/applications
  - provides high-level abstractions (e.g., files) and services (e.g., multiprogramming)

- Linux:
  - a “Unix-like” operating system
    - Whether an OS is Unix or “Unix-like” is mostly about whether the OS has a tiny bit of original Unix code and/or the activity of lawyers

How Do I Get Access to Bash?

If you’re using a MAC, then you have it. Just open up a terminal window. That is a Bash shell.

(MAC OS have been based on UNIX since OS X)

How Do I Get Access to Bash?

If you have a Windows Machine you have the 3 options given in the Introduction Lecture.

How Do I Get Access to Bash?

If you have a Windows Machine you have the 3 options given in the Introduction Lecture.

1. For Windows 10, you can use the Windows Subsystem for Linux
   a. This might be the future of “running UNIX” on Windows systems
   b. This won’t work if you’re using something earlier than Windows 10
   c. Also this may take a little more work to configure

How Do I Get Access to Bash?

If you have a Windows Machine you have the 3 options given in the Introduction Lecture.

1. For Windows 10, you can use the Windows Subsystem for Linux
   a. This might be the future of “running UNIX” on Windows systems
   b. This won’t work if you’re using something earlier than Windows 10
   c. This also might take a little more work to configure
2. You can download and run Cygwin
   a. This should work on all Windows systems later than XP

How Do I Get Access to Bash?

If you have a Windows Machine you have the 3 options given in the Introduction Lecture.

1. For Windows 10, you can use the Windows Subsystem for Linux
    a. This might be the future of "running UNIX" on Windows systems
    b. This won't work if you're using something earlier than Windows 10
    c. This also might take a little more work to configure
2. You can download and run Cygwin
    a. This should work on all Windows systems later than XP
3. You can connect to the cs dept computer lectura using a terminal
    a. This requires the least resources on your computer
    b. You must be connected to the Internet, lectura can get slow, files not local, etc

Unix Commands

- We tell the shell what to do on our behalf by *typing* commands

- Each command performs [variations of] a single task
  - “options” can be used to modify what a command does
  - different commands can be “glued together” to perform more complex tasks

- Syntax:

  command options arguments

- options and/or arguments are not always needed
  - they might have defaults or might not be relevant

Executing commands

- Typing a command name at the **bash** prompt and pressing the **ENTER** key causes the command to be executed.
- The command's output, if any, is displayed on the screen. Examples:

```
% hostname
lectura.cs.arizona.edu
% whoami
eanson
% true
% date
Sat Aug 15 18:54:39 MST 2015
% ps
  PID TTY          TIME CMD
22758 pts/18   00:00:00 bash
30245 pts/18   00:00:00 ps
```

Command-line arguments

- Most commands accept one or more *arguments*:

```
% cal 9 2015
   September 2015
Su Mo Tu We Th Fr Sa
       1  2  3  4  5
 6  7  8  9 10 11 12
13 14 15 16 17 18 19
20 21 22 23 24 25 26
27 28 29 30

% echo Hello, world!
Hello, world!

% factor 223092870
223092870: 2 3 5 7 11 13 17 19 23
```

Command-line options

- Many commands accept *options* that adjust the behavior of the command.
- Options almost always begin with a '-' (minus sign). Options are usually specified immediately following the command. For most programs the ordering of options is not significant but that is a convention, not a rule.
- Examples:

```
% date
Thu Jan 13 02:19:20 MST 2005

% date -u
Thu Jan 13 09:19:22 UTC 2005

% wc Hello.java
      5      14     127 Hello.java

% wc -l -w Hello.java
      5      14 Hello.java
```

- The command **`wc -l -w Hello.java`** has two options and one *operand* (the file Hello.java).

Options, continued

- Whitespace is often significant in command lines. For example, the following commands are all invalid: (Try them!)

```
% date-u

% wc -l-w Hello.java

% wc -- notes Hello.java
```

Finding out about commands (continued)

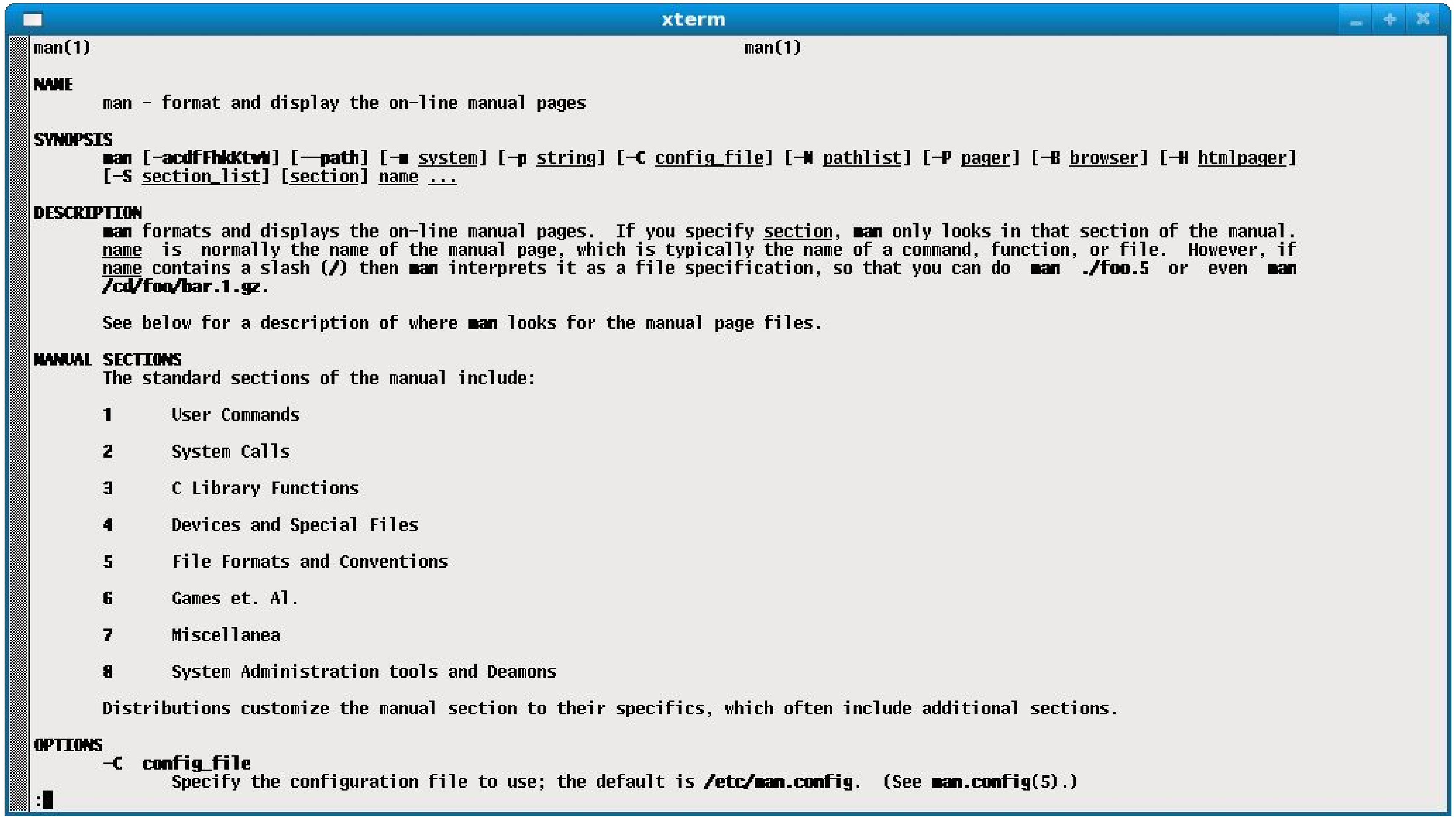
Figuring out how to use a command

man *command*

"displays the on-line manual pages"

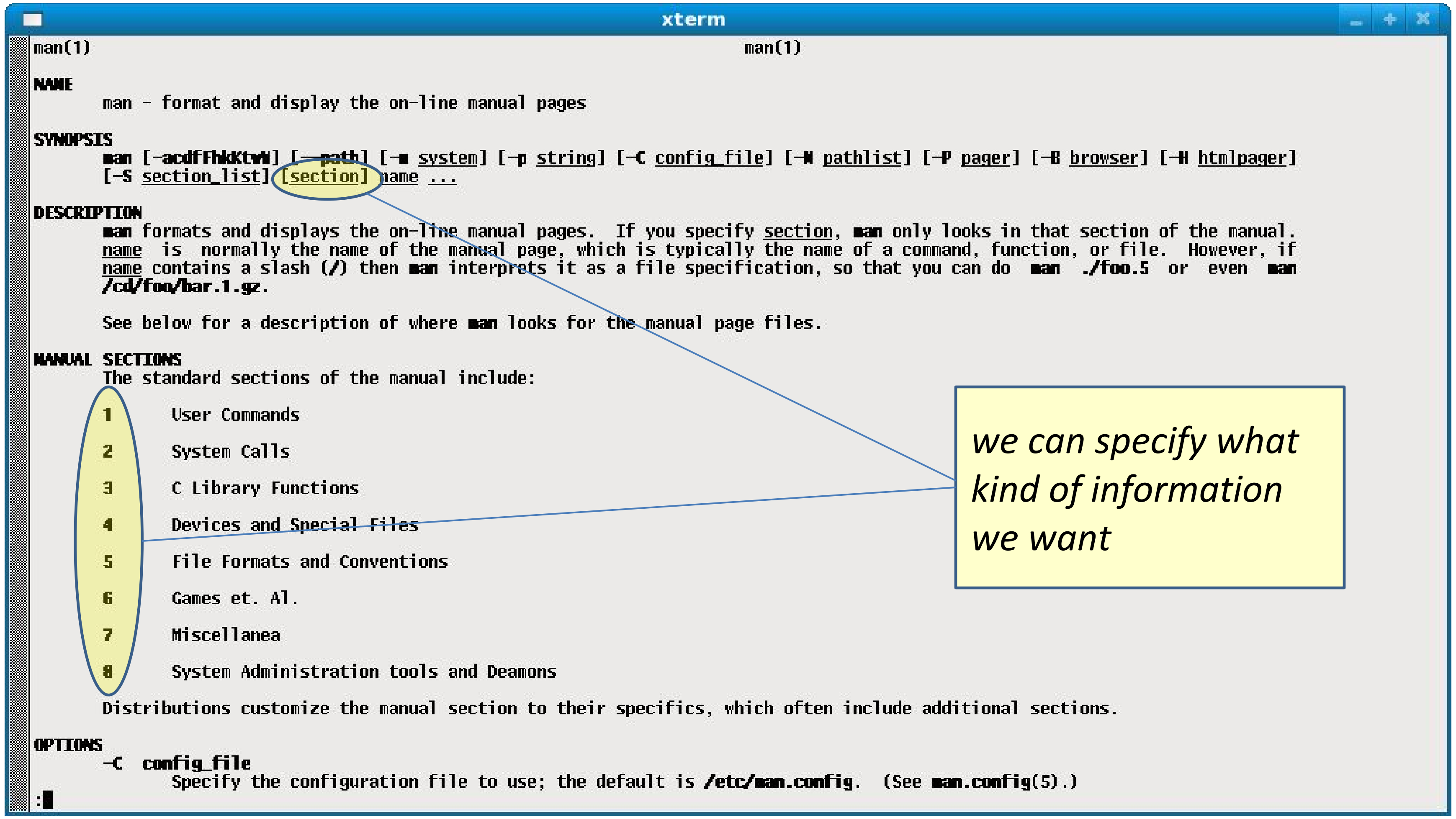
- Provides information about command options, arguments, return values, bugs, etc.

Example: “man man”



```
man(1)                                                                  man(1)

NAME
       man - format and display the on-line manual pages

SYNOPSIS
       man [-acdfFhkKtwW] [--path] [-m system] [-p string] [-C config_file] [-M pathlist] [-P pager] [-B browser] [-H htmlpager]
       [-S section_list] [section] name ...

DESCRIPTION
       man formats and displays the on-line manual pages.  If you specify section, man only looks in that section of the manual.
       name  is  normally the name of the manual page, which is typically the name of a command, function, or file.  However, if
       name contains a slash (/) then man interprets it as a file specification, so that you can do  man  ./foo.5  or  even  man
       /cd/foo/bar.1.gz.

       See below for a description of where man looks for the manual page files.

MANUAL SECTIONS
       The standard sections of the manual include:

       1      User Commands

       2      System Calls

       3      C Library Functions

       4      Devices and Special Files

       5      File Formats and Conventions

       6      Games et. Al.

       7      Miscellanea

       8      System Administration tools and Deamons

       Distributions customize the manual section to their specifics, which often include additional sections.

OPTIONS
       -C  config_file
              Specify the configuration file to use; the default is /etc/man.config.  (See man.config(5).)
:
```

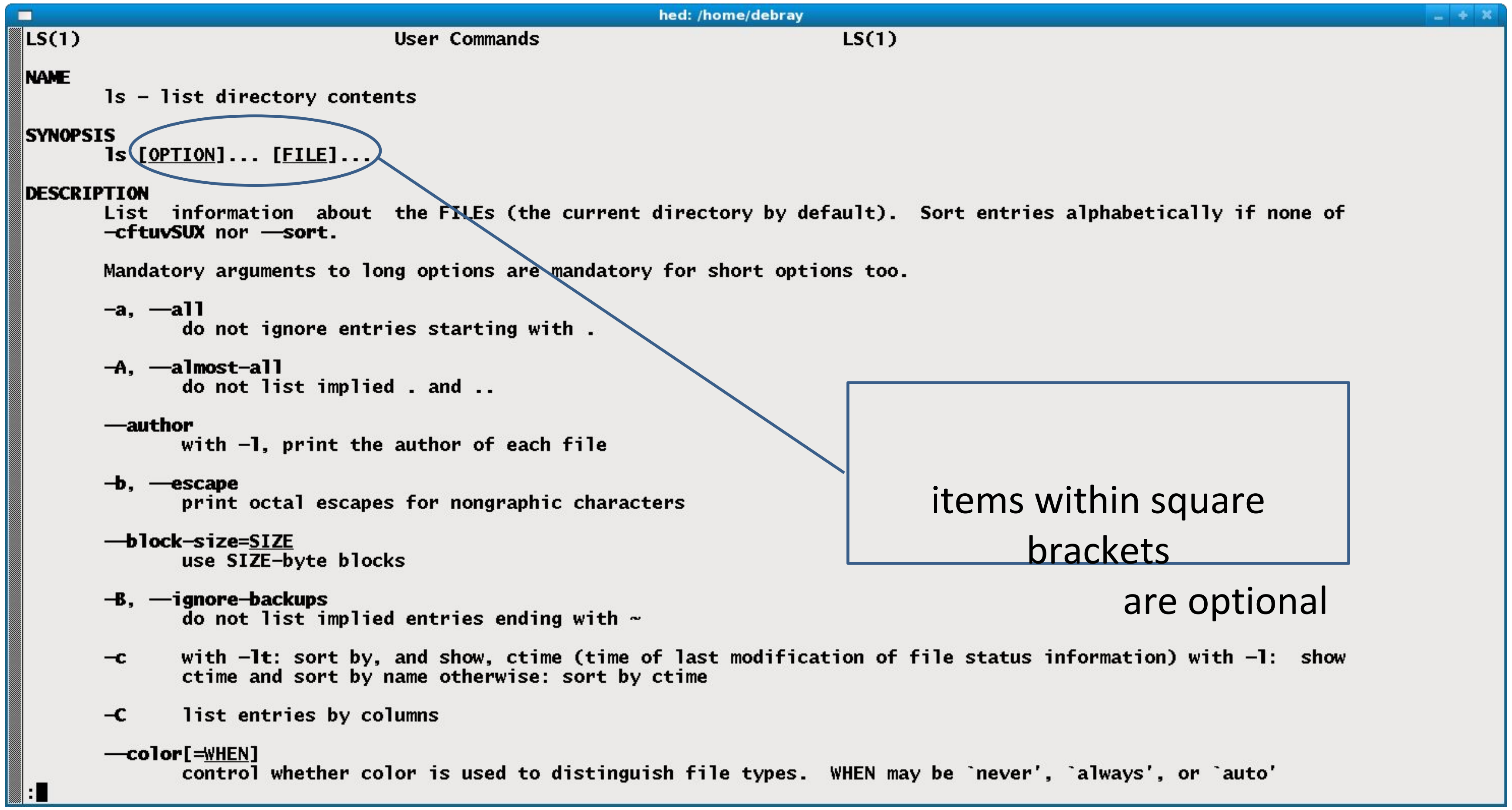
Example: “man man”



```
man(1)                                                                    man(1)

NAME
       man - format and display the on-line manual pages

SYNOPSIS
       man [-acdfFhkKtwW] [--path] [-m system] [-p string] [-C config_file] [-M pathlist] [-P pager] [-B browser] [-H htmlpager]
       [-S section_list] [section] name ...

DESCRIPTION
       man formats and displays the on-line manual pages.  If you specify section, man only looks in that section of the manual.
       name  is  normally the name of the manual page, which is typically the name of a command, function, or file.  However, if
       name contains a slash (/) then man interprets it as a file specification, so that you can do  man  ./foo.5  or  even  man
       /cd/foo/bar.1.gz.

       See below for a description of where man looks for the manual page files.

MANUAL SECTIONS
       The standard sections of the manual include:

       1      User Commands

       2      System Calls

       3      C Library Functions

       4      Devices and Special Files

       5      File Formats and Conventions

       6      Games et. Al.

       7      Miscellanea

       8      System Administration tools and Deamons

       Distributions customize the manual section to their specifics, which often include additional sections.

OPTIONS
       -C  config_file
              Specify the configuration file to use; the default is /etc/man.config.  (See man.config(5).)
:
```

we can specify what kind of information we want

Example: "man ls"



```
hed: /home/debray
LS(1)                          User Commands                         LS(1)

NAME
       ls - list directory contents

SYNOPSIS
       ls [OPTION]... [FILE]...

DESCRIPTION
       List  information  about  the FILEs (the current directory by default).  Sort entries alphabetically if none of
       -cftuvSUX nor --sort.

       Mandatory arguments to long options are mandatory for short options too.

       -a, --all
              do not ignore entries starting with .

       -A, --almost-all
              do not list implied . and ..

       --author
              with -l, print the author of each file

       -b, --escape
              print octal escapes for nongraphic characters

       --block-size=SIZE
              use SIZE-byte blocks

       -B, --ignore-backups
              do not list implied entries ending with ~

       -c     with -lt: sort by, and show, ctime (time of last modification of file status information) with -l:  show
              ctime and sort by name otherwise: sort by ctime

       -C     list entries by columns

       --color[=WHEN]
              control whether color is used to distinguish file types.  WHEN may be `never', `always', or `auto'
:
```

items within square brackets are optional

Finding commands

Figuring out which command to use

```
apropos   keyword
man  –k   keyword
```

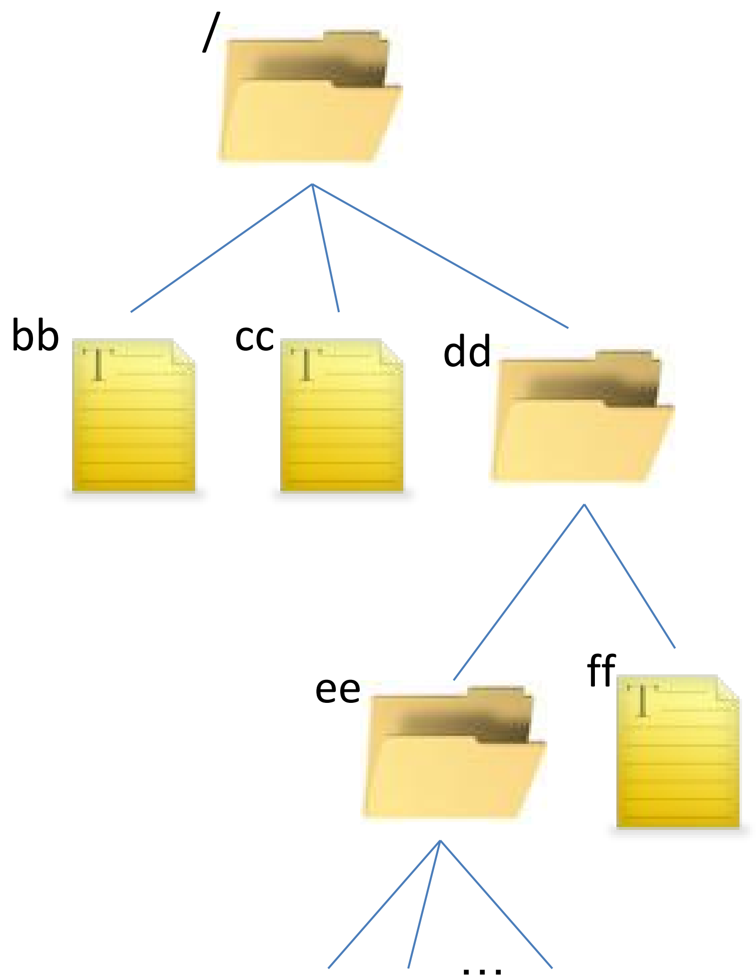
Two ways to do the same thing --- "searches a set of database files containing short descriptions of system commands for keywords"

- Helpful, but not a panacea:
  - depends on appropriate choice of keywords
    - may require trial and error
  - may return a lot of results to sift through
    - pipe through **more**
  - Google might be faster

The file system

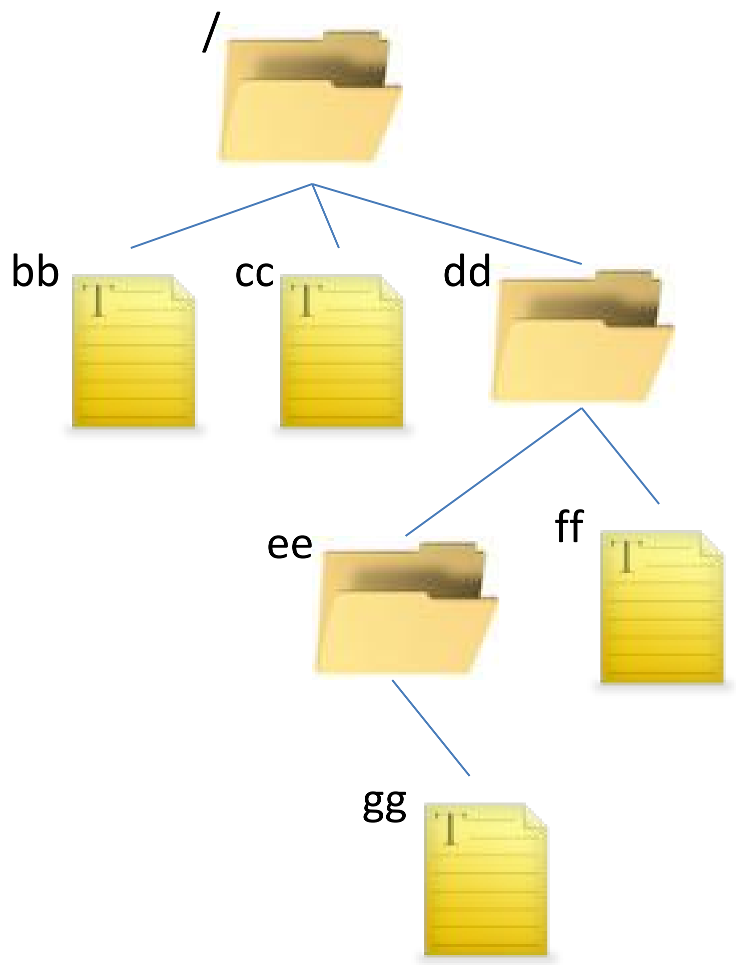
- Collections of files are grouped into ***directories*** (folders)

- A directory is itself a file
  - file system has a hierarchical structure (i.e., like a tree)
    - the root is referred to as “/”



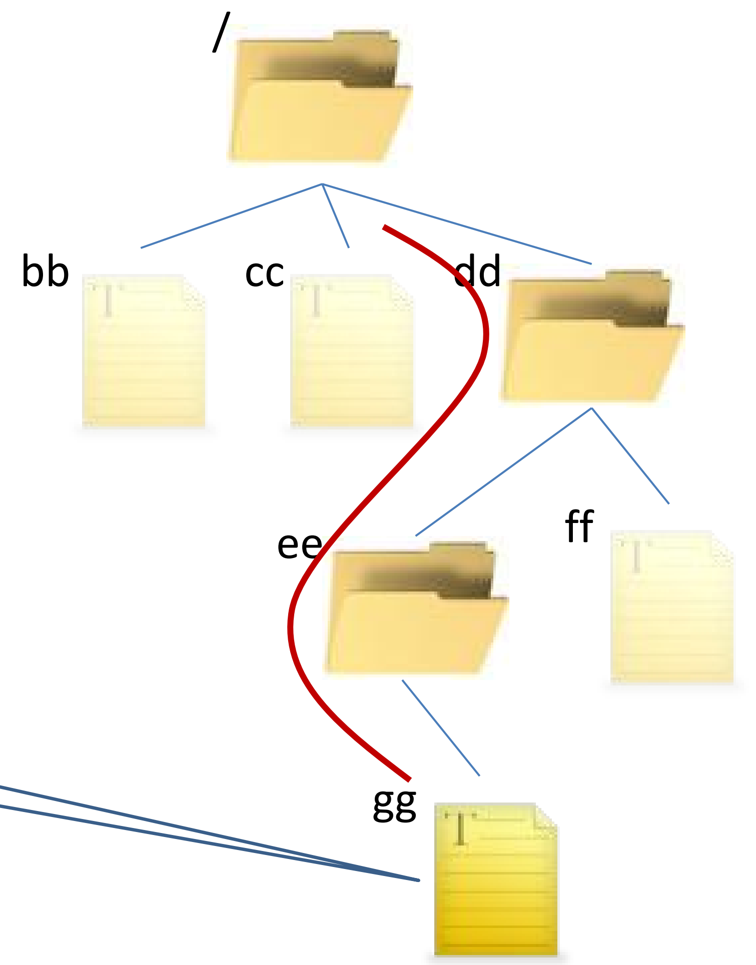
Referring to files: Absolute Paths

- The *absolute path* specifies a file (or directory) by how you to get to it starting at the file system root
  - The absolute path lists the directories on the path from the root (“/”), separated by “/”



Referring to files: Absolute Paths

- The *absolute path* specifies a file (or directory) by how you to get to it starting at the file system root
  - The absolute path lists the directories on the path from the root ("/"), separated by "/"



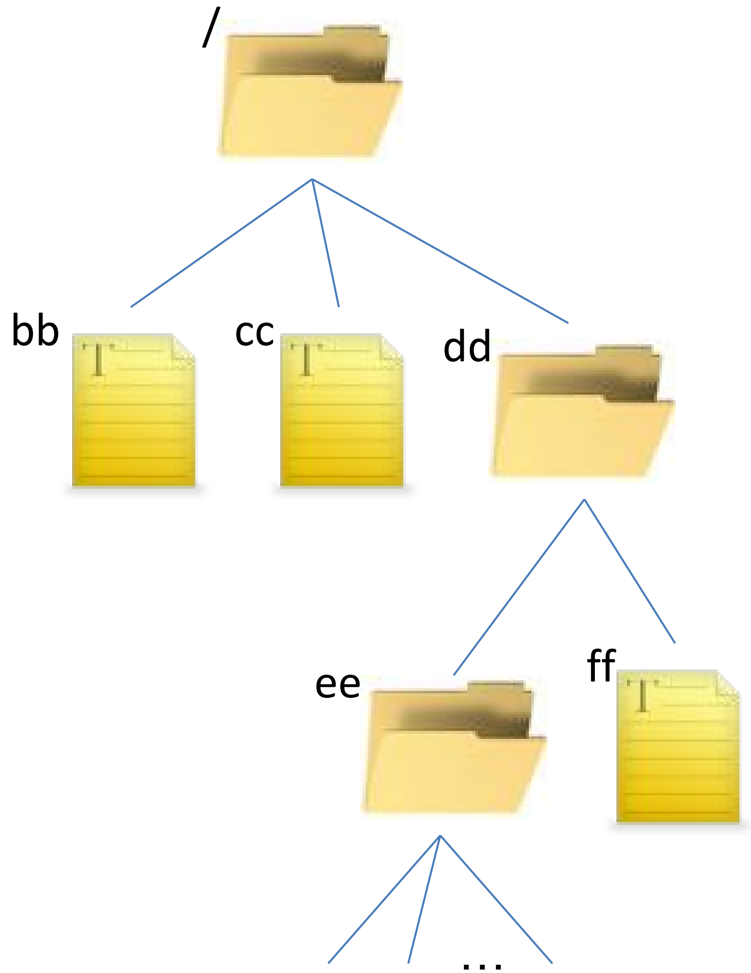
absolute path: **/dd/ee/gg**

The current directory

- Invariably you are in a particular directory in the tree called the current (working) directory
  - commands are executed in this context

- To find out the absolute path of the current directory, you can use the command “pwd”

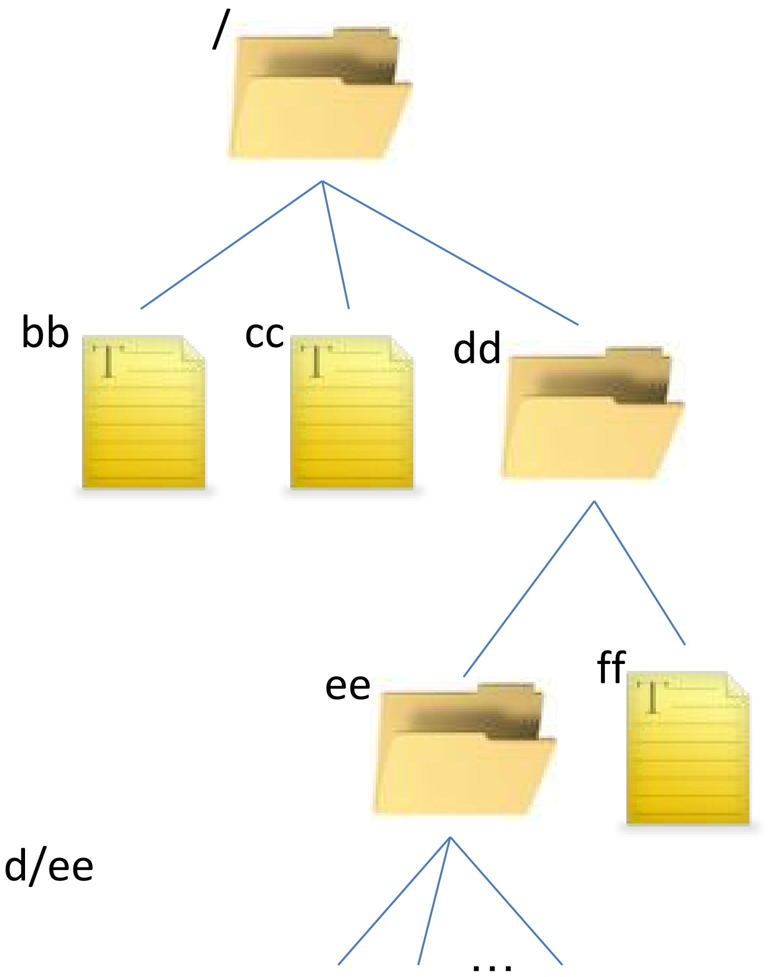
If ee is the current directory, then pwd should say ?



The current directory

- Invariably you are in a particular directory in the tree called the current (working) directory
  - commands are executed in this context

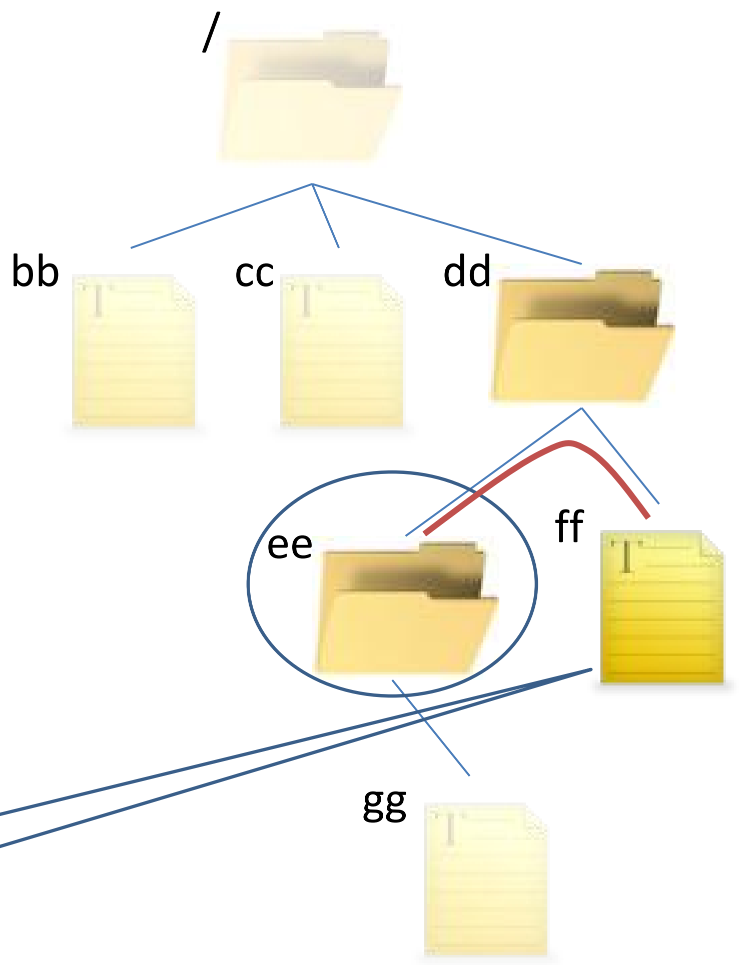
- To find out the absolute path of the current directory, you can use the command “pwd”



If ee is the current directory, then pwd should say /dd/ee

Referring to files: Relative Paths

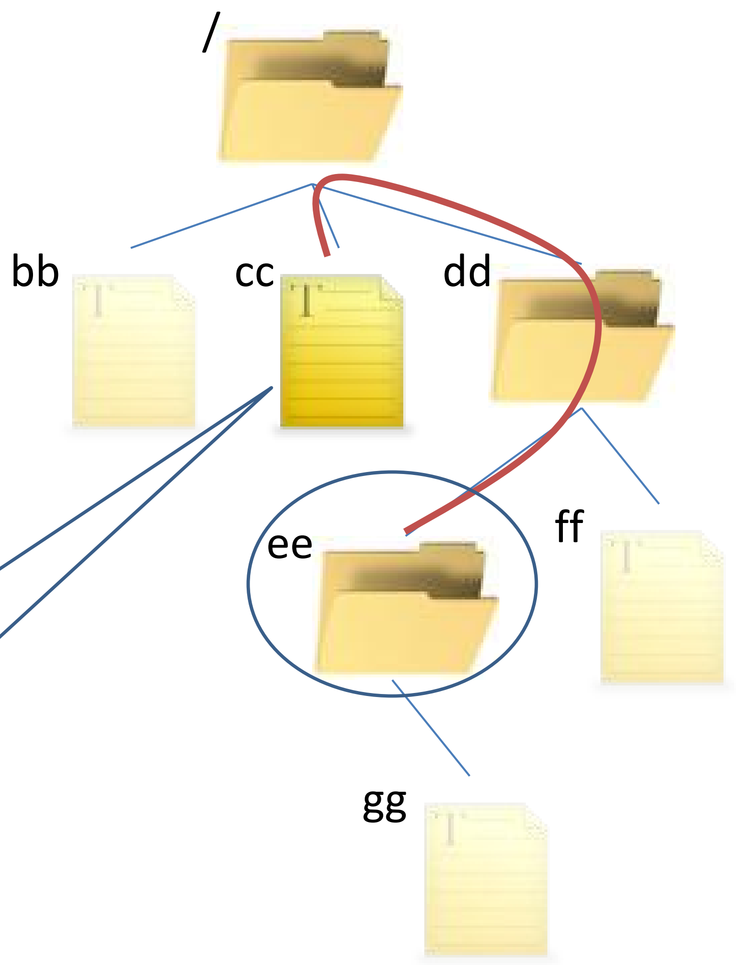
- A *relative path* specifies how to get to a file starting from the current directory
  - '..' means "move up one level"
  - '.' means current directory
  - lists the directories along the path separated by "/"



Example:
ff relative to **ee** is: **../ff**

Referring to files: Relative Paths

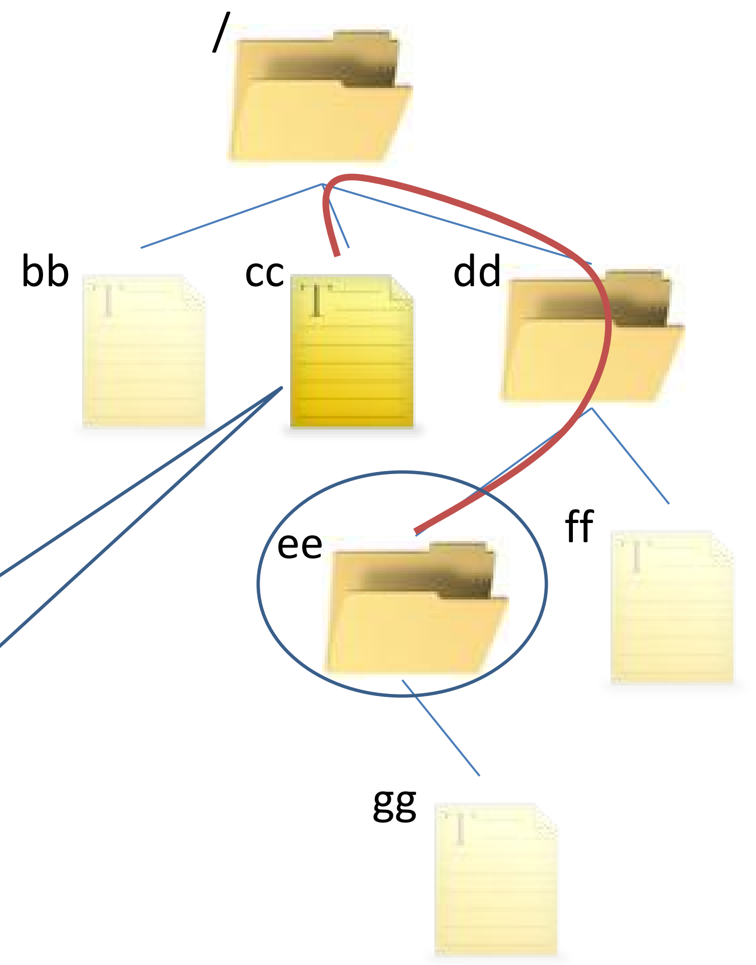
- A *relative path* specifies how to get to a file starting from the current directory
  - '..' means "move up one level"
  - '.' means current directory
  - lists the directories along the path separated by "/"



Example:
cc relative to **ee** is: **?**

Referring to files: Relative Paths

- A ***relative path*** specifies how to get to a file starting from the current directory
  - '**..**' means "move up one level"
  - '**.**' means current directory
  - lists the directories along the path separated by "/"



Example:
cc relative to **ee** is: **../../cc**

Home directories

- Each user has a “home directory”
  - specified when the account is created
  - given in the file **/etc/passwd**

- When you log in, your current directory is your home directory

- Notational shorthand:
  - absolute path to one’s own home directory: **~**
  - absolute path to some other user **joe**’s home directory: **~joe**

Absolute versus relative?

- Absolute paths start with “/” or shorthand symbols like “~”.
- Relative paths are anything else.

Unix commands for dealing with files and directories

- Moving (changing current directory) to another directory
  - **cd** *targetdir*

Examples:

cd /	move to the root of the file system
cd ~ (also: just “**cd**” by itself)	move to one’s home directory
cd /usr/local/src	move to /usr/local/src
cd ../..	move up two levels

Seeing What Files are in a Directory

- Command: **ls** — *lists the contents of a directory*
  - Examples:

ls	list the files in the current directory ⚠ *won't show files whose names start with '**.**'*
ls /usr/bin	list the files in the directory /usr/bin
ls -l	give a "long format" listing (provides additional info about files)
ls -a	list all files in the current directory, including those that start with '.'
ls -al /usr/local	give a "long format" listing of all the files (incl. those starting with '.') in /usr/local

Handy options for `ls`

- `-t` Sort by modification time instead of alphabetically.
- `-h` Show sizes with human-readable units like `K`, `M`, and `G`.
- `-r` Reverse the order of the sort.
- `-S` Sort by file size
- `-d` By default, when an argument is a directory, `ls` operates on the entries contained in that directory. `-d` says to operate on the directory itself. Try "`ls -l .`" and "`ls -ld .`".
- -R Recursively list all the subdirectories.
- There are many more and you might want to look at the man page and play with them

More commands for dealing with files

- **cp** *file*$_1$ *file*$_2$
  - copy *file*$_1$ to *file*$_2$
- **mv** *file*$_1$ *file*$_2$
  - move *file*$_1$ to *file*$_2$
- **rm** *file*
  - *removes the file*
  - *for directories use the option “-r”*
- **mkdir** dir
  - make a (sub) directory in the current directory
- **vi** [file]
  - the vi editor
- **vimtutor**
  - a tutorial for using vi

Three handy options for `cp`:

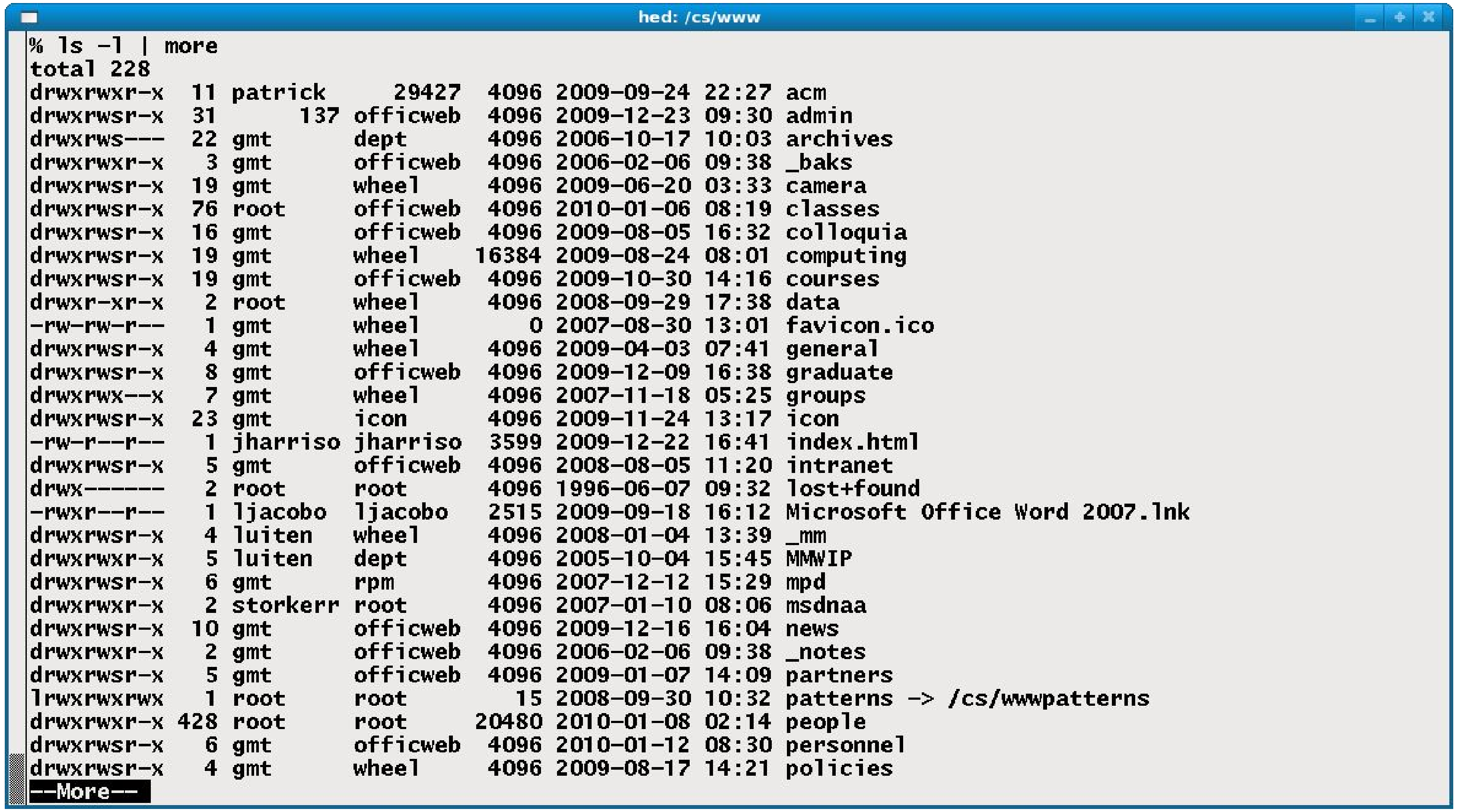
• `-R`	Recursively copy an entire directory tree (For many unix commands either “-R” or “-r” will mean do it recursively down whatever is below the directory)
• `-p`	Preserve file permissions, ownerships, and timestamps
• `-i`	Inquire before overwriting destination file.

Some other useful commands

- **wc** [*file*]
  - *word count*: counts characters, words, and lines in the input
  - (already used as an example)
- **cat** [$file_1$] [$file_2$]…
  - concatenates files and writes them to standard output
- **head** *–n* [*file*]
  - show the first *n* lines of the input
- **tail** *–n* [*file*]
  - show the last *n* lines of the input
- **touch** [$file_1$] [$file_2$]…
  - updates the timestamp on files, creating them as needed

Getting more information about files

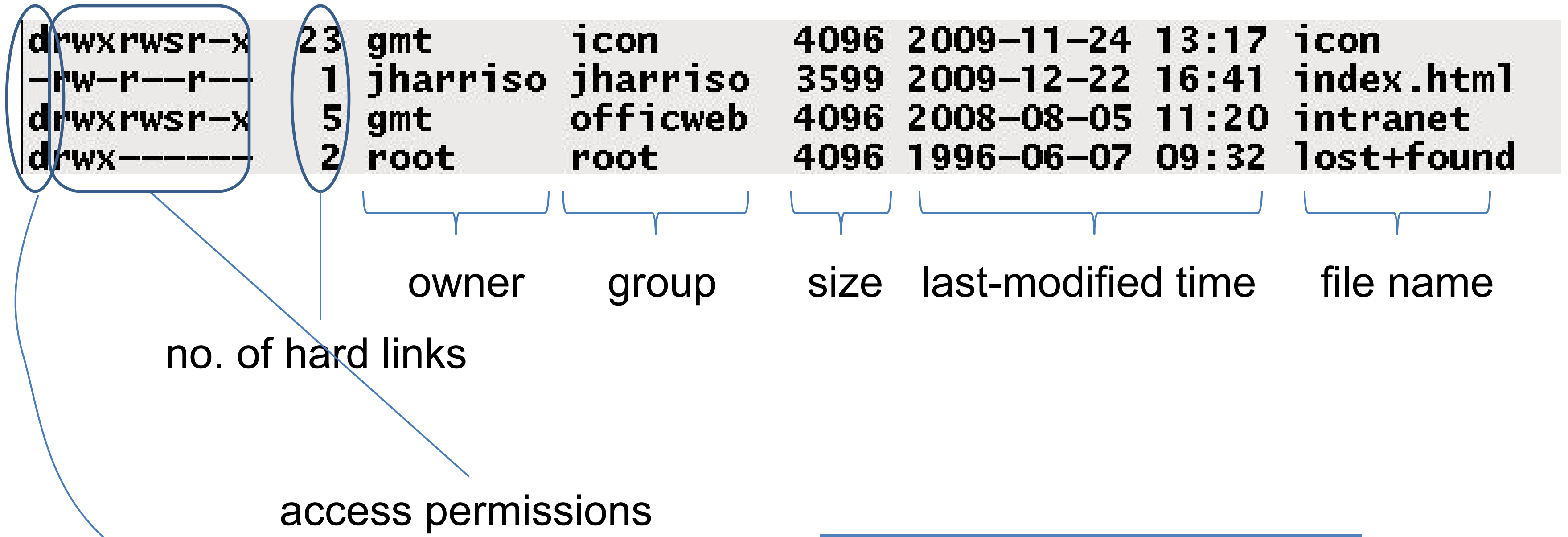
- ls –l : provides additional info about files



hed: /cs/www

```
% ls -l | more
total 228
drwxrwxr-x  11 patrick     29427  4096 2009-09-24 22:27 acm
drwxrwsr-x  31       137 officweb  4096 2009-12-23 09:30 admin
drwxrws---  22 gmt      dept      4096 2006-10-17 10:03 archives
drwxrwxr-x   3 gmt      officweb  4096 2006-02-06 09:38 _baks
drwxrwsr-x  19 gmt      wheel     4096 2009-06-20 03:33 camera
drwxrwsr-x  76 root     officweb  4096 2010-01-06 08:19 classes
drwxrwsr-x  16 gmt      officweb  4096 2009-08-05 16:32 colloquia
drwxrwsr-x  19 gmt      wheel    16384 2009-08-24 08:01 computing
drwxrwsr-x  19 gmt      officweb  4096 2009-10-30 14:16 courses
drwxr-xr-x   2 root     wheel     4096 2008-09-29 17:38 data
-rw-rw-r--   1 gmt      wheel        0 2007-08-30 13:01 favicon.ico
drwxrwsr-x   4 gmt      wheel     4096 2009-04-03 07:41 general
drwxrwsr-x   8 gmt      officweb  4096 2009-12-09 16:38 graduate
drwxrwx--x   7 gmt      wheel     4096 2007-11-18 05:25 groups
drwxrwsr-x  23 gmt      icon      4096 2009-11-24 13:17 icon
-rw-r--r--   1 jharriso jharriso  3599 2009-12-22 16:41 index.html
drwxrwsr-x   5 gmt      officweb  4096 2008-08-05 11:20 intranet
drwx------   2 root     root      4096 1996-06-07 09:32 lost+found
-rwxr--r--   1 ljacobo  ljacobo   2515 2009-09-18 16:12 Microsoft Office Word 2007.lnk
drwxrwsr-x   4 luiten   wheel     4096 2008-01-04 13:39 _mm
drwxrwxr-x   5 luiten   dept      4096 2005-10-04 15:45 MMWIP
drwxrwsr-x   6 gmt      rpm       4096 2007-12-12 15:29 mpd
drwxrwxr-x   2 storkerr root      4096 2007-01-10 08:06 msdnaa
drwxrwsr-x  10 gmt      officweb  4096 2009-12-16 16:04 news
drwxrwxr-x   2 gmt      officweb  4096 2006-02-06 09:38 _notes
drwxrwsr-x   5 gmt      officweb  4096 2009-01-07 14:09 partners
lrwxrwxrwx   1 root     root        15 2008-09-30 10:32 patterns -> /cs/wwwpatterns
drwxrwxr-x 428 root     root     20480 2010-01-08 02:14 people
drwxrwsr-x   6 gmt      officweb  4096 2010-01-12 08:30 personnel
drwxrwsr-x   4 gmt      wheel     4096 2009-08-17 14:21 policies
--More--
```

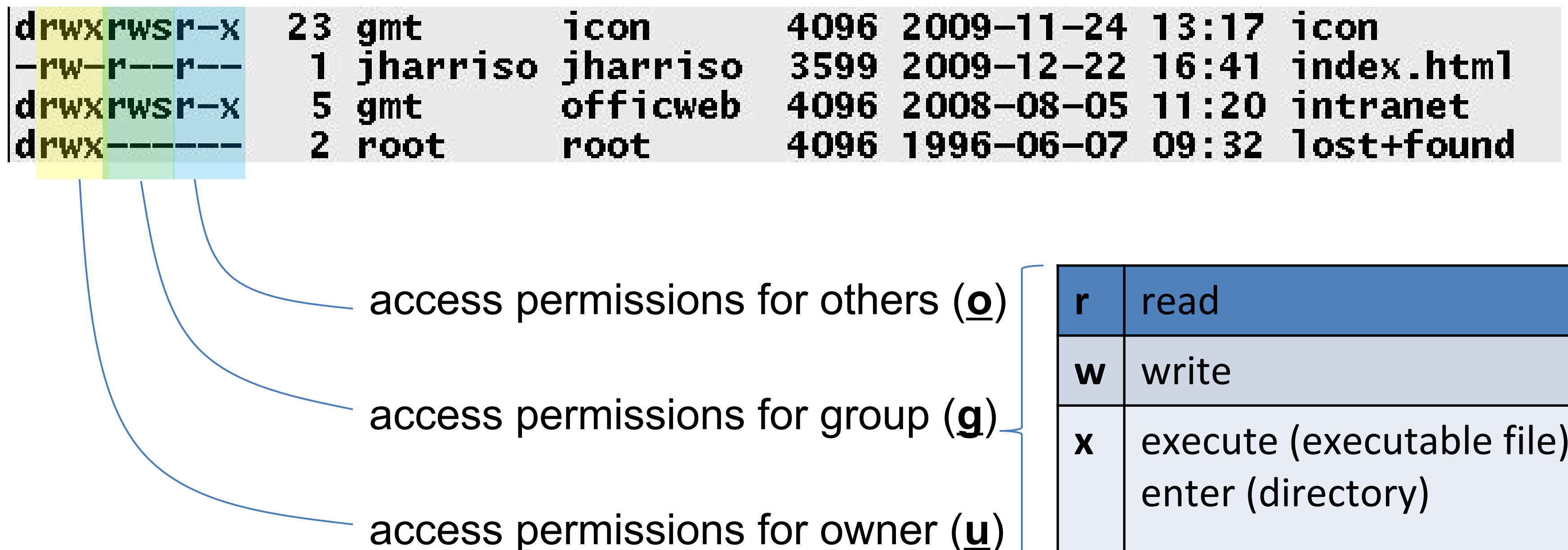
Getting more information about files… (1)


```
drwxrwsr-x  23 gmt      icon      4096 2009-11-24 13:17 icon
-rw-r--r--   1 jharriso jharriso  3599 2009-12-22 16:41 index.html
drwxrwsr-x   5 gmt      officweb  4096 2008-08-05 11:20 intranet
drwx------   2 root     root      4096 1996-06-07 09:32 lost+found
```

file type

–	normal file
d	directory
l (*ell*)	symbolic link

File access permissions



```
drwxrwsr-x  23 gmt      icon      4096 2009-11-24 13:17 icon
-rw-r--r--   1 jharriso jharriso  3599 2009-12-22 16:41 index.html
drwxrwsr-x   5 gmt      officweb  4096 2008-08-05 11:20 intranet
drwx------   2 root     root      4096 1996-06-07 09:32 lost+found
```

access permissions for others (**o**)

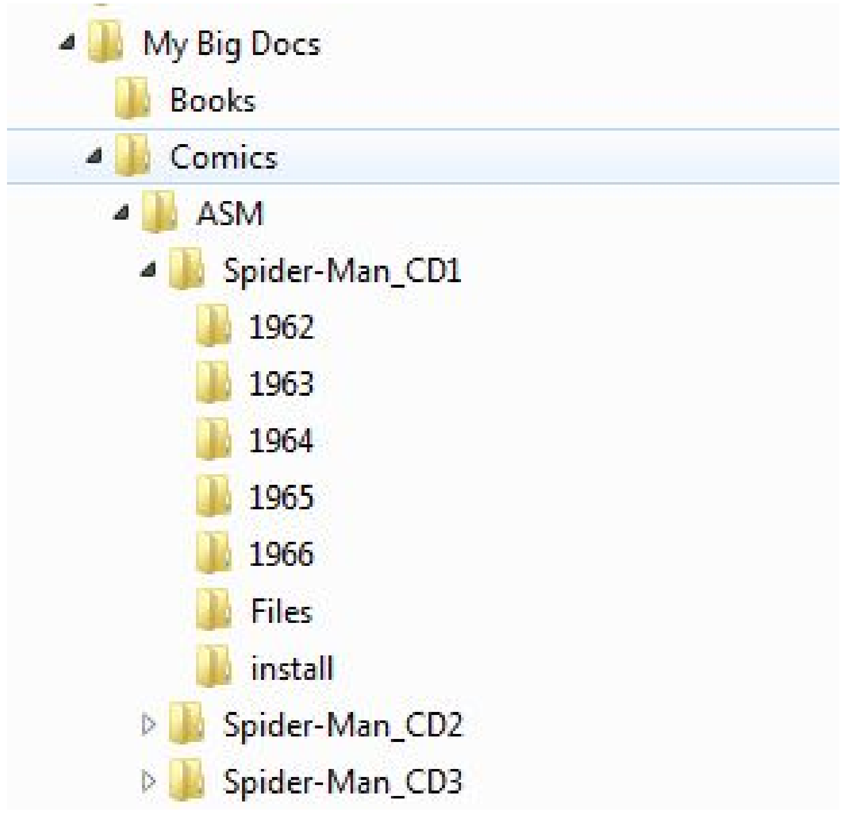
access permissions for group (**g**)

access permissions for owner (**u**)

r	read
w	write
x	execute (executable file) enter (directory)
–	no permission

How does this relate to Windows and File Explorer

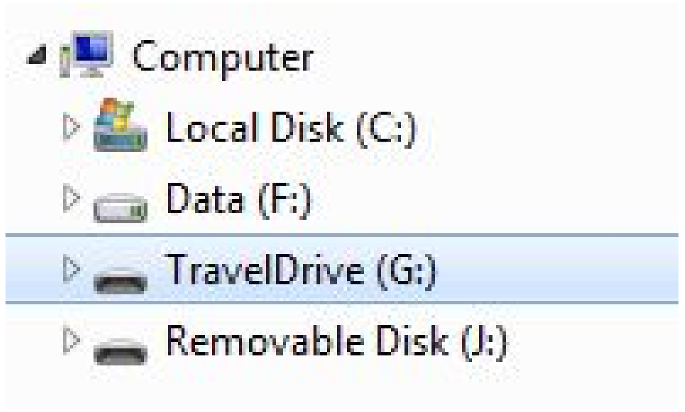
Directories are organized as trees and the GUI lets you navigate those trees:



```
◢ My Big Docs
    Books
  ◢ Comics
    ◢ ASM
      ◢ Spider-Man_CD1
          1962
          1963
          1964
          1965
          1966
          Files
          install
      ▷ Spider-Man_CD2
      ▷ Spider-Man_CD3
```

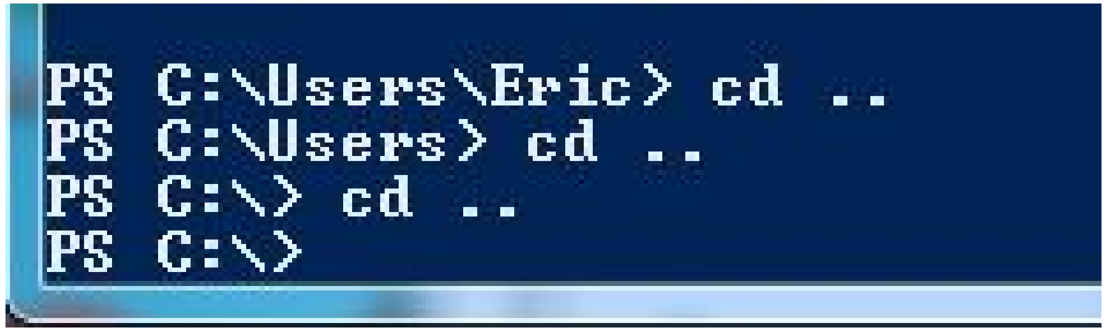
How does this relate to Windows and File Explorer

One difference in organization is that Windows has a tree per “drive” instead of one root.



Notice it kind of looks like “Computer” is the root of the tree, but it doesn’t show up in the command line.

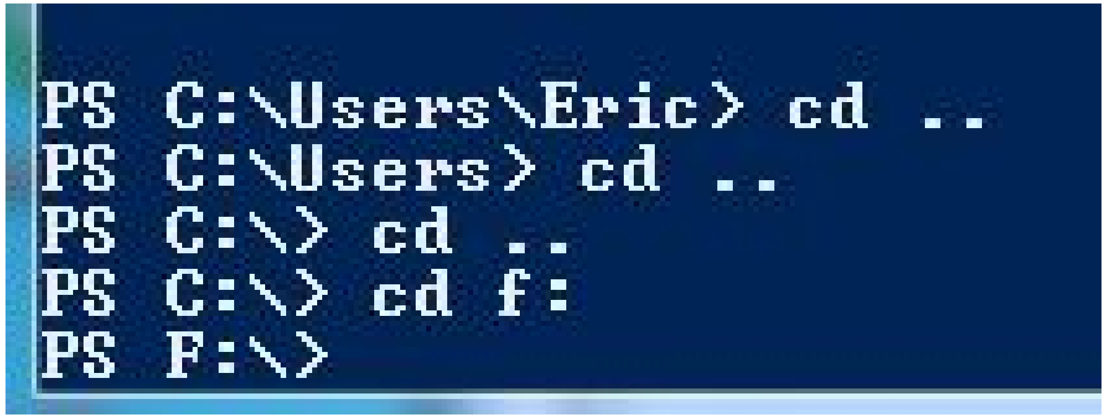
How does this relate to Windows and File Explorer



```
PS C:\Users\Eric> cd ..
PS C:\Users> cd ..
PS C:\> cd ..
PS C:\>
```

This is in the Windows Powershell

You can't change directories above the C: root. You can get to another drive by using cd <drive letter>:



```
PS C:\Users\Eric> cd ..
PS C:\Users> cd ..
PS C:\> cd ..
PS C:\> cd f:
PS F:\>
```

This works in the Powershell by NOT the Command Prompt.

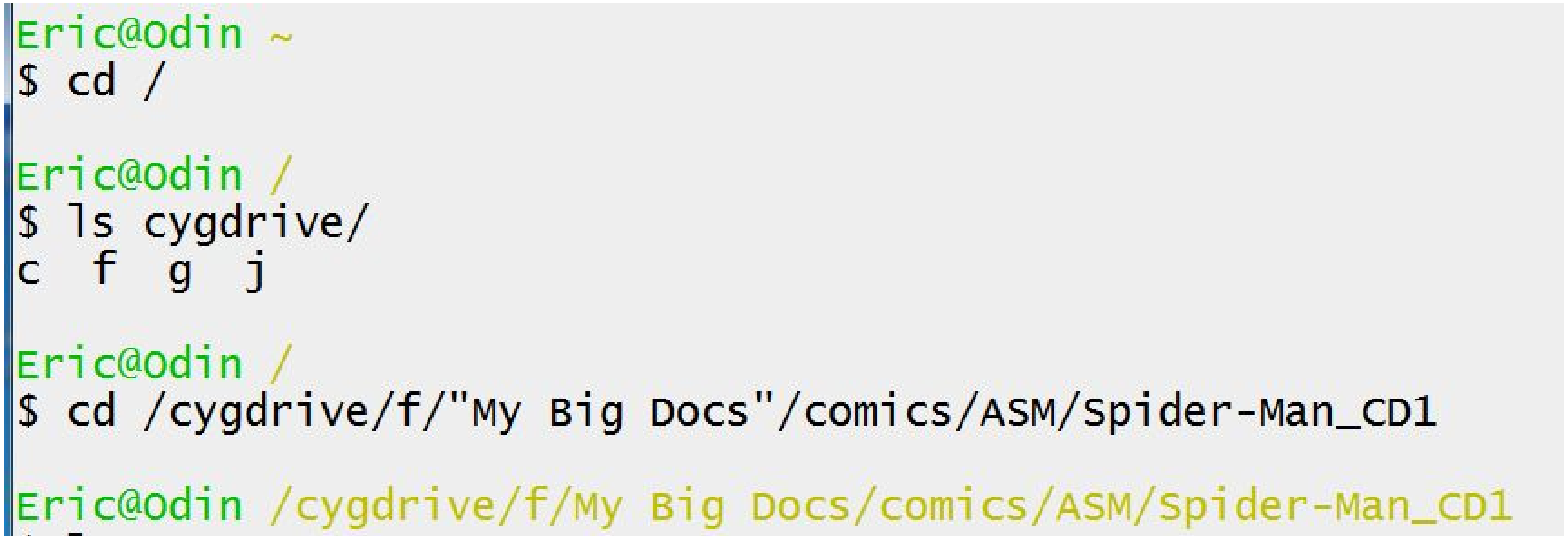
So What About Cygwin?

Cygwin sets up a root directory at the location where cygwin was installed. But it mounts the drive root directories under:

/cygdrive

This means if I want to access a file under say f:\”My Big Docs”\comics\ASM\Spider-Man_CD1
I could find them in cygwin at:
/cydrive/f/”My Big Docs”/comics/ASM/Spider-Man_CD1

So What About Cygwin?



```
Eric@Odin ~
$ cd /

Eric@Odin /
$ ls cygdrive/
c  f  g  j

Eric@Odin /
$ cd /cygdrive/f/"My Big Docs"/comics/ASM/Spider-Man_CD1

Eric@Odin /cygdrive/f/My Big Docs/comics/ASM/Spider-Man_CD1
```

Screenshot from previous slide. Using spaces in a directory or file name is a Windows thing, but should be avoided in UNIX (it is a pain).

And What About Linux Subsystem for Windows?

- The location of the root directory is hidden in the Windows File System
  - This is because files created by linux programs/commands in this area are not compatible with Windows programs

- The computer drives are in the directory /mnt
  - Files created in directories /mnt/<drive letter> will be compatible (can be opened by) windows programs.
- A good solution is to make a symbolic link to a directory you want to work with.

Symbolic Links

Motivation:

Often you will want to reach a “distant” directory from your home directory. For example, in the Linux Subsystem for Windows the files that can be modified by windows programs are in the directory /mnt/<drive letter>. On my system I want to work with files here:

```
/mnt/c/Users/Eric/Documents/UofA/cs210/WorkArea
```

My home directory is **`/home/eanson`**

Using the following command will set up a symbolic link to my desired directory

```
% ln -s /mnt/c/Users/Eric/Documents/cs210/WorkArea/ cs210
```

Symbolic Links

Doing an ls after creating the link I see:

```
% ls
cs210
```

This can be treated like a directory. I can usd **cd** to enter it. However, if I do a

```
% ls -l
lrwxrwxrwx 1 eanson eanson 48 Aug 22 10:49 cs210 ->
/mnt/c/Users/Eric/Documents/cs210/WorkArea/
```

That lowercase "L" at the start of the line indicates that `cs210` is a *symbolic link*, often shortened to "*symlink*".

The `cs210 -> /mnt/c/Users/Eric/Documents/cs210/WorkArea/` indicates that `cs352` references (or "points to") that entry.

Symbolic Links

```
% ls cs210
simple.py
```

The `cs210` symlink creates the illusion that `my home directory` has an `cs210` subdirectory

```
~/352/a2 % ls /mnt/c/Users/Eric/Documents/cs210/WorkArea
simple.py
```

Symbolic Links

Key point:

Symbolic links are handled by the operating system.

Benefit:

A program doesn't have to do anything special to follow a symlink to its destination.

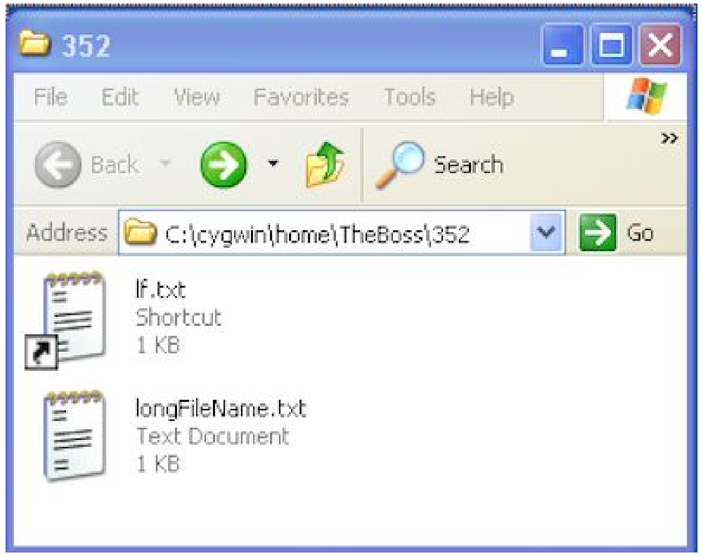
Symbolic Links

- A symlink is kind of like a "Windows shortcut done right.”
- Ditto for Mac “aliases”
- Macs are UNIX underneath and also have symbolic links
- Unix also has quite a different kind of link (hard) which we won’t say anything more about (just be aware that you usually want symbolic).

Sidebar: Windows shortcuts

I've made a Windows shortcut named `lf.txt` that references `longFileName.txt`.

I can open either with Explorer but watch what `type`, the Windows analog of `cat(1)`, does:



C:\cygwin\home\TheBoss\352

lf.txt
Shortcut
1 KB

longFileName.txt
Text Document
1 KB

- `C:>`**`type longFileName.txt`**
- `Tue, Sep 01, 2015  5:50:25 PM`

- `C:>`**`type lf.txt.lnk`**
- `L▒F▒ ▒[▒▒□$_▒▒▒▒[▒▒]P▒O▒ ▒:i▒+00▒/C:\:1▒Bcygwin$▒▒<A"G▒cygwin41▒<L home ▒▒<▒"G▒home<1! ...`

- If a Windows program wants to handle shortcuts, it's got to have special code to do it!

Symbolic Links

- File-related utility programs often have special handling for symbolic links.
- One example is `ls`, whose `-L` option says to "follow" the link.

```
% ls -l cs210
lrwxrwxrwx 1 eanson eanson 48 Aug 22 10:49 cs210
 -> /mnt/c/Users/Eric/Documents/cs210/WorkArea/

~/inClass % ls -lL cs210
-rwxrwxrwx 1 root root 62 Aug 22 1111 simple.py
```

Input and output

- Data are read from and written to i/o *streams*

- There are three predefined streams:

  stdin : "standard input" **–** usually, keyboard input

  stdout : "standard output" **–** usually, the screen

  stderr : "standard error" **–** for error messages (usually, the screen)

- Other streams can be created using system calls (e.g., to read or write a specific file)

I/O Redirection

- Default input/output behavior for commands:
  - **stdin**: keyboard; **stdout**: screen; **stderr**: screen

- We can change this using I/O redirection:

cmd < *file*	redirect **cmd**'s stdin to read from *file*
cmd > *file*	redirect **cmd**'s stdout to *file*
cmd >> *file*	append **cmd**'s stdout to *file*
cmd &> *file*	redirect **cmd**'s stdout and stderr to *file*
$\mathbf{cmd}_1$ | $\mathbf{cmd}_2$	redirect $\mathbf{cmd}_1$'s stdout to $\mathbf{cmd}_2$'s stdin

Combining commands

- The output of one command can be fed to another command as input.
  - Syntax: $\mathit{command}_1 \underbrace{\;|\;}_{\text{“pipe”}} \mathit{command}_2$

Example:

ls	lists the files in a directory
more *foo*	shows the file *foo* one screenful at a time
ls | more	lists the files in a directory one screenful at a time

How this works:
- **ls** writes its output to its **stdout**
- **more**’s input stream defaults to its **stdin**
- the pipe connects **ls**’s stdout to **more**’s stdin
- the piped commands run “in parallel”

- A key element of the UNIX philosophy is to use *pipelines* to combine programs to solve a problem, rather than writing a new program.

- Problem: How many unique users are on lectura?

v1: Get login names

```
% who| cut -f1 -d " "
ken
dmr
ken
francis
rob
walt24
dmr
rob
wnj
dmr
ken
```

v2: Sort login names

```
% who|cut -f1 -d" "|sort
dmr
dmr
dmr
francis
ken
ken
ken
rob
rob
walt24
wnj
```

v3: Get unique login names

```
% who|cut -f1 -d" "|sort|uniq
dmr
francis
ken
rob
walt24
wnj
```

v4: Get the count

```
% who|cut -f1 -d" "|sort|uniq|
wc -l
6
```

stdin and stdout and Python

Side note: You will want to be able to run Python from your command line. MAC users should already be set.

If you're using Cygwin, you might want to install python3, and likewise if you're using WSL (Windows Subsystem for Linux). Here is a link that talks about installing Python (and other things) on WSL

https://blogs.windows.com/buildingapps/2016/07/22/fun-with-the-windows-subsystem-for-linux/

stdin and stdout and Python

You have already read from stdin and written to stdout many times. Look at the very simple program in a file called **`s1.py`**

```
line = input()
print(line)
```

When run from the command line this program waits for a line to be typed in, and then prints it out to the terminal. By default input() reads from stdin and print() outputs to stdout.

stdin and stdout and Python

The sys module allows you to be more explicit about the fact you're writing to **`stdin`** or **`stdout`**. This program does the same as the last:

```
import sys
line = sys.stdin.readline()
sys.stdout.write(line)
```

stdin and stdout and Python

Putting this in a loop, we can read and write as long as there is input:

```
% cat partCat.py
import sys
line = sys.stdin.readline()
while line:
   sys.stdout.write(line)
   line = sys.stdin.readline()
```

This program may seem silly and useless, but it can be somewhat useful with file redirection:

stdin and stdout and Python

We can use it to show the contents of an ascii file:

```
 $ python3 partCat.py <s1.py
line = input()
print(line)
```

We see **`s1.py`** contains the two lines of our earlier simple program.

stdin and stdout and Python

We can also use it as a very simple way to create a text file:

```
 $ python3 partCat.py >README
I typed this in

$ cat README
I typed this in
```

We tell Bash that we are finished with input from stdin by typing Ctrl-d

python and command line arguments

Our **`partCat.py`** program is almost like the UNIX **`cat`** command.

If **`cat`** is invoked with no arguments it also reads from **`stdin`** and writes to **`stdout`**. However, if you can invoke **`cat`** with the names of files it will read from those instead of **`stdin`**.

```
$ cat file1 file2 file3
```

will print the contents of file1 followed by file2 followed by file3 to **stdout**

python and command line arguments

You can use the **sys** module to read command line arguments in a python program. **sys** contains the list **argv** which is a list of the arguments. The first element is the name of the program itself.

python and command line arguments

```
$ cat argReader.py
import sys
for arg in sys.argv:
  print(arg)

$ python3 argReader.py John Paul George
argReader.py
John
Paul
George
```

A Simple Python Program acting like cat

```
$ cat myCat.py
import sys
if len(sys.argv) == 1:
   line = sys.stdin.readline()
   while line:
      sys.stdout.write(line)
      line = sys.stdin.readline()
else:
   for i in range(1, len(sys.argv)):
      f = open(sys.argv[i])
      line = f.readline()
      while line:
         sys.stdout.write(line)
         line = f.readline()
      f.close()
```

Text Editors

- You will want to find a good text editor to create programs and test files.
  - You have done and will do much of this work using an IDE (Integrated Development Environment) that usually combines a text editor, compiler or interpreter, I/O display, and some kind of file management.
  - But it is often useful or necessary to load up a program file in a text editor.
- You do NOT want to use a word processor for this work (like Word)
  - A word processor includes information in the file about formatting
  - A text editor just creates the ASCII text you write.

Text Editors

- You will probably want to learn a little bit about using a nongraphical editor that runs on a terminal. Three common ones in UNIX are:
- emacs
- vim (vi iMproved)
- nano (pico clone)
- None are installed by default in Cygwin. ☹ But all are available ☺

Text Editors

- You may also want to learn and use an editor that uses windows and a mouse.
  - Notepad comes installed on Windows, but it is not good.
  - UNIX and Windows (and MACs?) use different ASCII characters to mark the end of a line. A good editor for programmers will let you convert these.
  - Some editors do auto indent (can be a useful or a pain), color coding, and/or bracket matching, etc.
- A FEW of the commonly used text editors are listed on the next slide

Text Editors

•Notepad++	Windows	Free
•Sublime Text	All	$70 (but unlimited free trial)
•BBEdit (TextWrangler)	MAC	Free limited version
•Brackets	All	Free
•jEdit	All	Free
•Atom	All	Free

There are MANY more. Find one you like and learn to use it.

You may want to visit https://en.wikipedia.org/wiki/Comparison_of_text_editors

The diff command

- The diff command looks for differences between files
- You will probably want to know this command since it will be used in grading.
- It also can be helpful when you have two version of a program and want to see how they differ.

Back to the shell – command line editing and shortcuts

bash supports simple command line recall and editing with the "arrow keys" but many control-key and escape sequences have meaning too. Here are a few:

`^A/^E`	Go to start/end of line.
`^W`	Erase the last "word".
`^U`	Erase whole line. (`^C` works, too.)
`^R`	Do incremental search through previous commands.
`ESC-f/b`	Go forwards/backwards a word. (Two keystrokes: **ESC**, then **f**)
`ESC-.`	Insert last word on from last command line. (Very handy!)

bash also does command and filename completion with **TAB**:

Hit **TAB** to complete to longest unique string.

If a "beep", hit **TAB** a second time to see alternatives.

example 1: Try typing “xc” followed by a TAB (or two)

example 2: Try typing “ls “ followed by a TAB (or two) (in a directory with some files!)

Shell metacharacters

- Many non-alphanumeric characters have special meaning to shells.

- Characters that have special meaning are often called *metacharacters*. Here are the **`bash`** metacharacters:

- `~ ` ` ! # $ & * ( ) \ | { } [] ; ' " < > ?`

Shell metacharacters

- Many non-alphanumeric characters have special meaning to shells.
- Characters that have special meaning are often called *metacharacters*. Here are the **`bash`** metacharacters:
- `~ ` ` ` ` ! # $ & * ( ) \ | { } [] ; ' " < > ?`
- If an argument has metacharacters or white space we suppress their special meaning by enclosing the argument in single quotes.
  - Double quotes suppresses most of the special meanings, but not all

Shell metacharacters

- Many non-alphanumeric characters have special meaning to shells.

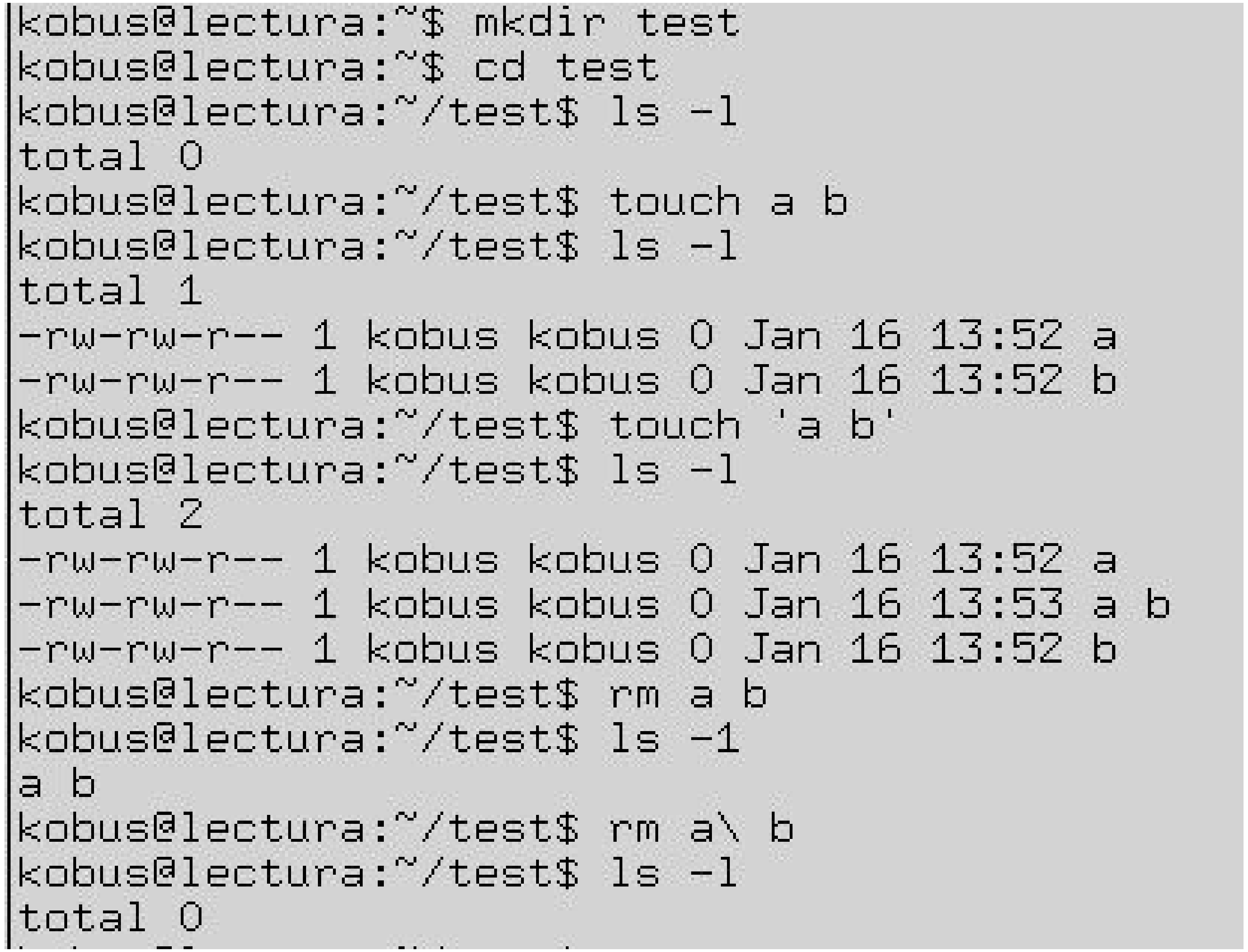
- Characters that have special meaning are often called *metacharacters*. Here are the **`bash`** metacharacters:

- `~ ` ` ` ! # $ & * ( ) \ | { } [] ; ' " < > ?`

- If an argument has metacharacters or white space we suppress their special meaning by enclosing the argument in quotes.
  - Double quotes suppresses most of the special meanings, but not all

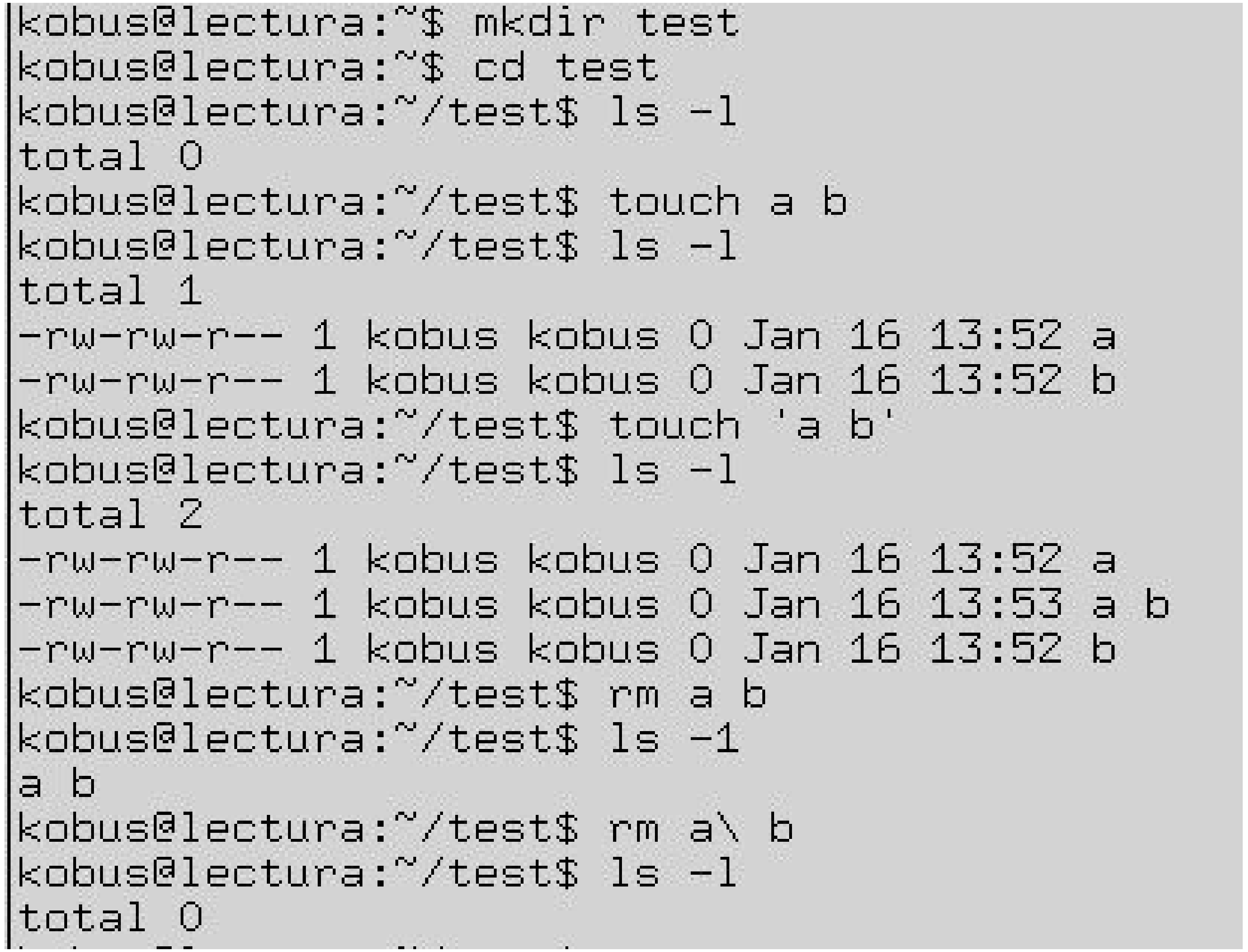
- An alternative is to use a backslash to "escape" each metacharacter.

Examples of escaping



```
kobus@lectura:~$ mkdir test
kobus@lectura:~$ cd test
kobus@lectura:~/test$ ls -l
total 0
kobus@lectura:~/test$ touch a b
kobus@lectura:~/test$ ls -l
total 1
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 a
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 b
kobus@lectura:~/test$ touch 'a b'
kobus@lectura:~/test$ ls -l
total 2
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 a
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:53 a b
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 b
kobus@lectura:~/test$ rm a b
kobus@lectura:~/test$ ls -1
a b
kobus@lectura:~/test$ rm a\ b
kobus@lectura:~/test$ ls -l
total 0
```

Examples of escaping



```
kobus@lectura:~$ mkdir test
kobus@lectura:~$ cd test
kobus@lectura:~/test$ ls -l
total 0
kobus@lectura:~/test$ touch a b
kobus@lectura:~/test$ ls -l
total 1
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 a
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 b
kobus@lectura:~/test$ touch 'a b'
kobus@lectura:~/test$ ls -l
total 2
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 a
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:53 a b
-rw-rw-r-- 1 kobus kobus 0 Jan 16 13:52 b
kobus@lectura:~/test$ rm a b
kobus@lectura:~/test$ ls -1
a b
kobus@lectura:~/test$ rm a\ b
kobus@lectura:~/test$ ls -l
total 0
```

This is a fun example, but filenames with spaces in them should be AVOIDED if possible.

One has to know how to deal with them, but using them in unix leads to grief.

Wildcards

- Some metacharacters act as *Wildcards* allow the user to specify files and directories using text patterns in bash commands.

- The simplest wildcard metacharacter is `?`, a question mark. A question mark matches any one character.

Wildcards

Observe:

```
% ls
a   out   x   xy   z
```

```
% ls ?
a x z
```

```
% ls ???
out
```

Wildcards, continued

- `echo` is also good for exploring wildcards, and uses less vertical space on slides:

```
% ls
a   out   x   xy   z

% echo ?
a x z

% echo ??? ??
out xy
```

Wildcards, continued

```
% ls
a   out   x   xy   z
```

- Predict the output:

```
% echo ? ? ?
a x z a x z a x z

% echo x? ?y
xy xy

% echo ? ?? x ??? ????
a x z xy x out ????
```

If there's no match, like with `????`, the argument is passed through as-is.

The * wildcard

- A more powerful wildcard is `*` (asterisk). It matches any sequence of characters, including an empty sequence.
  - `*` Matches every name
  - `*.java` Matches every name that ends with `.java`
  - `*x*` Matches every name that contains an `x`
    - Examples: `x`, `ax`, `axe`, `xxe`, `xoxox`

The * wildcard

- What would be matched by the following?

  `*x*y`

  – Names that contain an `x` and end with `y`.

  `*.*`

  – Names that contain at least one dot.

  `*.*.*`

  – Names that contain at least two dots.

  `a*e*i*o*u`

  – Names that start with an `a`, end with a `u`, and have `e`, `i`, `o`, in sequence in the middle.

Combining wildcards

- Wildcards can be combined. Examples:

  `??*` Matches names that are two or more characters long

  `*.?` Matches names whose next to last character is a dot

What would be matched by the following?

`?x?*`

Names that are at least three characters long and have an `x` *as their second character.*

`*-?-*`

Names that contain two dashes separated by a single character.

The character set wildcard

- The character set wildcard specifies that a single character must match one of a set.
  - `% ls`
  - `a  b  e  n  out  x  xy  z`

  - `% echo [m-z]` `#` *one-character names in the range* `m-z`
  - `n x z`

- Another:
  - `% ls`
  - `Makefile  fmt.c  utils.c  utils.h`

  - `% echo *.[chy]`
  - `fmt.c utils.c utils.h`

The character set wildcard

- More:
  - `[A-Z]*.[0-9]`
    - Matches names that start with a capital letter and end with a dot and a digit.
  - `*.[!0-9]`   *(Leading ! complements the set.)*
    - Matches names that end with a dot and a non-digit character.
    - Equivalent: `*.[^0-9]`
  - `[Tt]ext`
    - Matches `Text` and `text`.

Lots more with wildcards

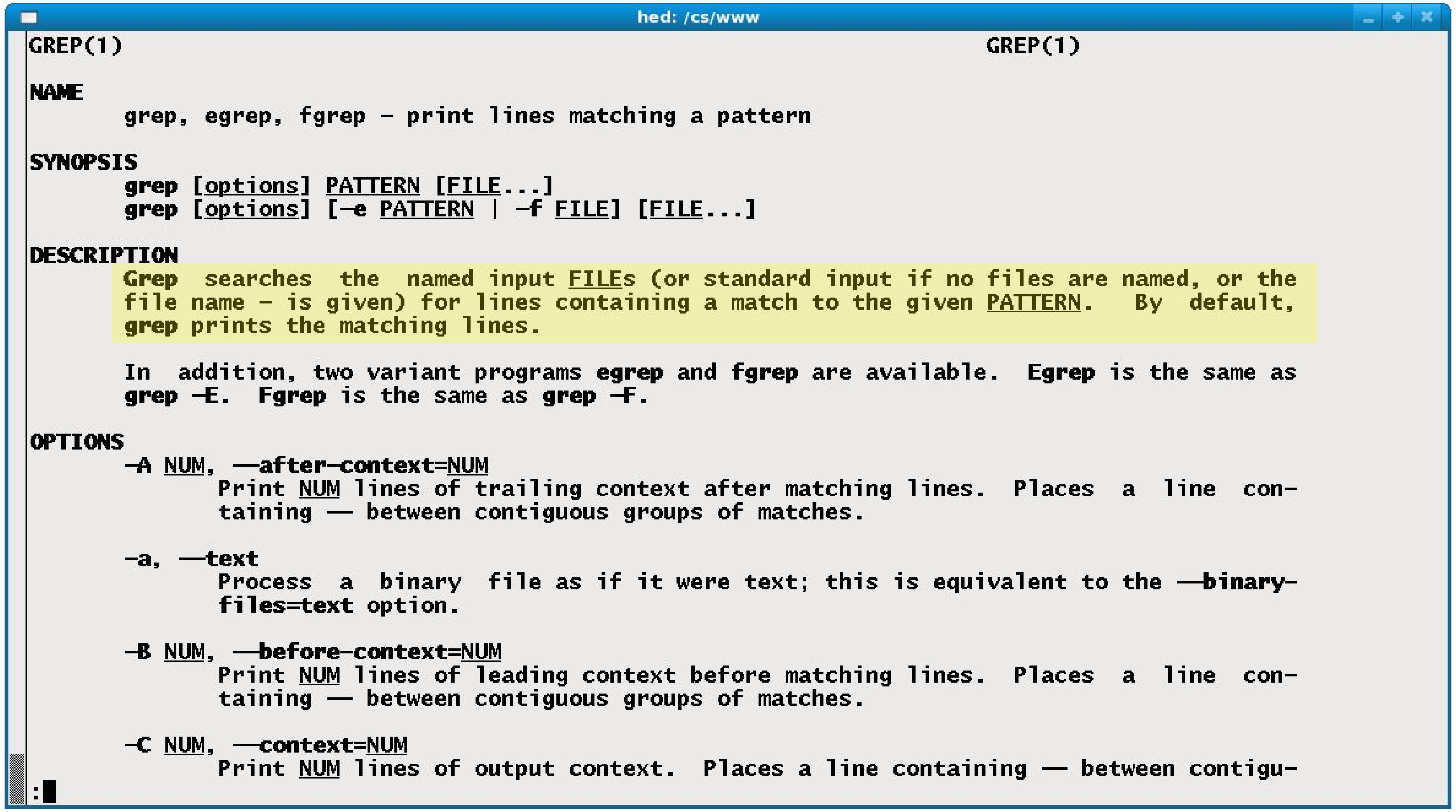
- The bash man page uses the term "pathname expansion" for what I've called wildcard expansion.

- Another term that's used is "globbing". Try searching for "glob" on the bash man page.

- Wildcards don't match hidden files unless the pattern starts with a dot, like `.*rc`.

- There are other wildcard specifiers but `?`, `*`, and `[...]` are the most commonly used.

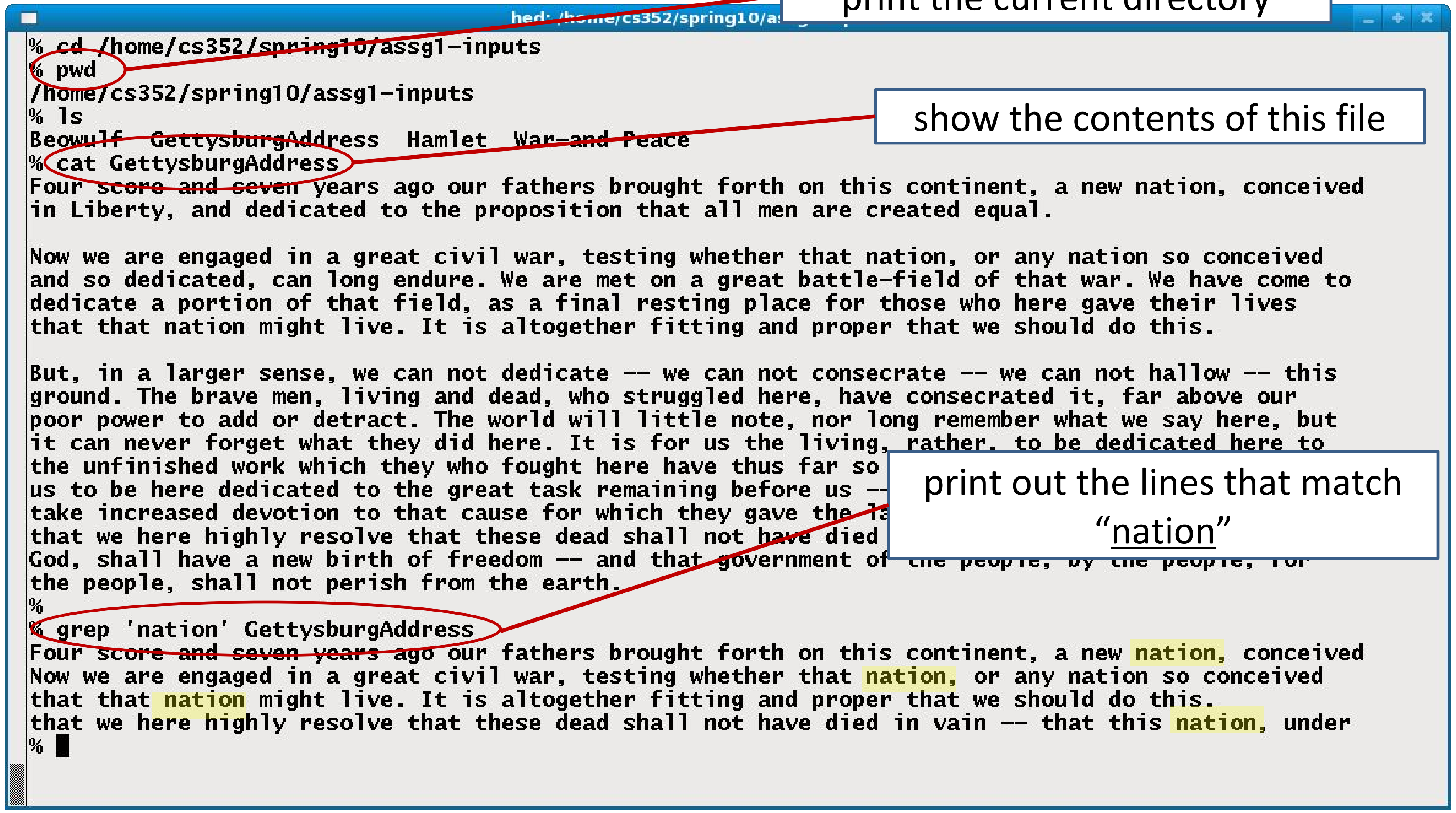
Pattern matching: grep



hed: /cs/www

```
GREP(1)                                                                GREP(1)

NAME
       grep, egrep, fgrep - print lines matching a pattern

SYNOPSIS
       grep [options] PATTERN [FILE...]
       grep [options] [-e PATTERN | -f FILE] [FILE...]

DESCRIPTION
       Grep  searches  the  named input FILEs (or standard input if no files are named, or the
       file name - is given) for lines containing a match to the given PATTERN.   By  default,
       grep prints the matching lines.

       In  addition, two variant programs egrep and fgrep are available.  Egrep is the same as
       grep -E.  Fgrep is the same as grep -F.

OPTIONS
       -A NUM, --after-context=NUM
              Print NUM lines of trailing context after matching lines.  Places  a  line  con-
              taining -- between contiguous groups of matches.

       -a, --text
              Process  a  binary  file as if it were text; this is equivalent to the --binary-
              files=text option.

       -B NUM, --before-context=NUM
              Print NUM lines of leading context before matching lines.  Places  a  line  con-
              taining -- between contiguous groups of matches.

       -C NUM, --context=NUM
              Print NUM lines of output context.  Places a line containing -- between contigu-
:
```

Pattern matching: grep… (1)



print the current directory

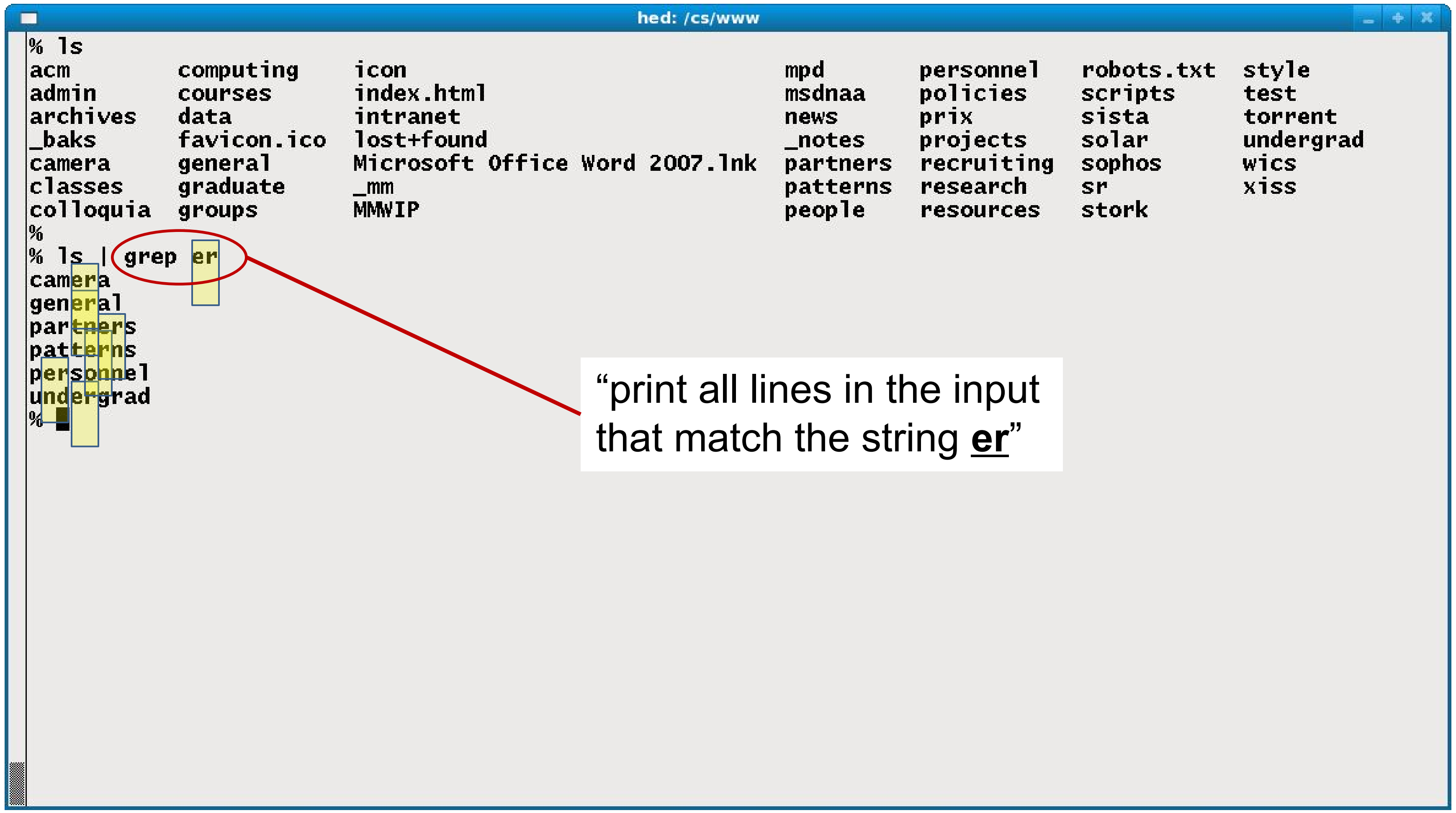
show the contents of this file

print out the lines that match "nation"

```
% cd /home/cs352/spring10/assg1-inputs
% pwd
/home/cs352/spring10/assg1-inputs
% ls
Beowulf  GettysburgAddress  Hamlet  War-and-Peace
% cat GettysburgAddress
Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived
in Liberty, and dedicated to the proposition that all men are created equal.

Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived
and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to
dedicate a portion of that field, as a final resting place for those who here gave their lives
that that nation might live. It is altogether fitting and proper that we should do this.

But, in a larger sense, we can not dedicate -- we can not consecrate -- we can not hallow -- this
ground. The brave men, living and dead, who struggled here, have consecrated it, far above our
poor power to add or detract. The world will little note, nor long remember what we say here, but
it can never forget what they did here. It is for us the living, rather, to be dedicated here to
the unfinished work which they who fought here have thus far so
us to be here dedicated to the great task remaining before us --
take increased devotion to that cause for which they gave the la
that we here highly resolve that these dead shall not have died
God, shall have a new birth of freedom -- and that government of
the people, shall not perish from the earth.
%
% grep 'nation' GettysburgAddress
Four score and seven years ago our fathers brought forth on this continent, a new nation, conceived
Now we are engaged in a great civil war, testing whether that nation, or any nation so conceived
that that nation might live. It is altogether fitting and proper that we should do this.
that we here highly resolve that these dead shall not have died in vain -- that this nation, under
%
```

Pattern matching: grep…

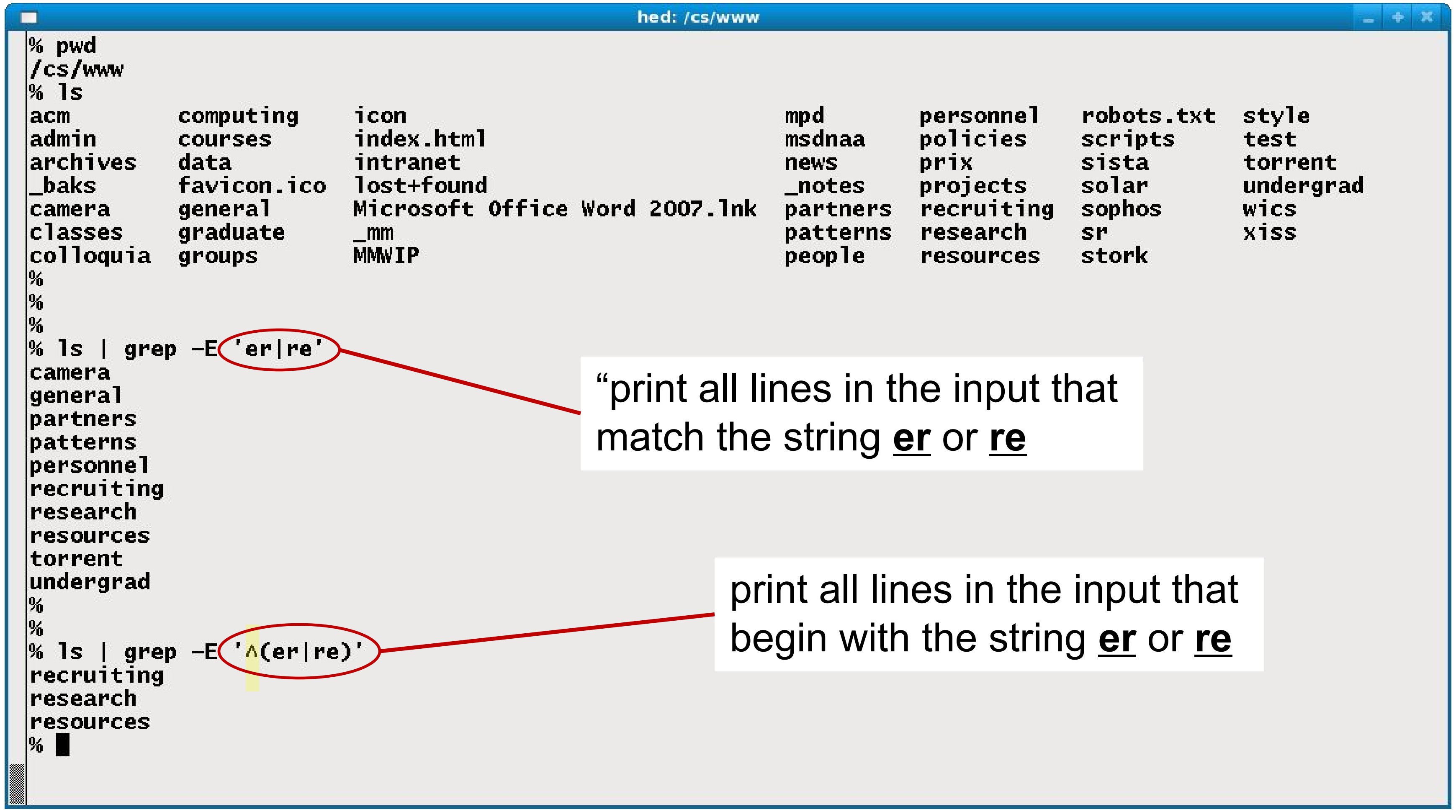
(2)



```
% ls
acm          computing   icon                               mpd       personnel   robots.txt  style
admin        courses     index.html                         msdnaa    policies    scripts     test
archives     data        intranet                           news      prix        sista       torrent
_baks        favicon.ico lost+found                         _notes    projects    solar       undergrad
camera       general     Microsoft Office Word 2007.lnk     partners  recruiting  sophos      wics
classes      graduate    _mm                                patterns  research    sr          xiss
colloquia    groups      MMWIP                              people    resources   stork
%
% ls | grep er
camera
general
partners
patterns
personnel
undergrad
%
```

"print all lines in the input that match the string **er**"

Pattern matching: grep… (3)


hed: /cs/www

```
% pwd
/cs/www
% ls
acm          computing    icon                             mpd         personnel    robots.txt   style
admin        courses      index.html                       msdnaa      policies     scripts      test
archives     data         intranet                         news        prix         sista        torrent
_baks        favicon.ico  lost+found                       _notes      projects     solar        undergrad
camera       general      Microsoft Office Word 2007.lnk   partners    recruiting   sophos       wics
classes      graduate     _mm                              patterns    research     sr           xiss
colloquia    groups       MMWIP                            people      resources    stork
%
%
%
% ls | grep -E 'er|re'
camera
general
partners
patterns
personnel
recruiting
research
resources
torrent
undergrad
%
%
% ls | grep -E '^(er|re)'
recruiting
research
resources
% █
```

"print all lines in the input that match the string **er** or **re**

print all lines in the input that begin with the string **er** or **re**

Setting defaults for your commands

- Create an “alias” for your command
  - syntax different for different shells
  - bash: **alias** *aliasName=“cmdName”*

    e.g.: alias rm=“rm –i”

However the alias will only last during this session of your shell

Good news and bad news

- Good news:
  - The behavior of bash can be customized by putting commands in the initialization files that bash reads.
- Bad news:
  - Several files and rules are involved.
  - The initialization files present and their contents vary from system to system and user to user.
- We'll first talk about the mechanics of bash's initialization files and then look at types of things we might add to initialization files.
- <u>Anything that's valid at the bash prompt can appear in an initialization file and vice-versa.</u> In other words, initialization files simply contain a sequence of bash commands.

The rules (simplified)

- If bash is specified as your shell in `/etc/passwd` and you login, the instance of bash that's started is said to be a *login shell*.

- <u>When bash is started as a login shell</u> it first reads `/etc/profile`. It then looks for three files in turn: `~/.bash_profile`, `~/.bash_login`, and `~/.profile`. Upon finding one, it executes the commands in that file and doesn't look any further.

The rules (simplified)

- Sometimes you'll want to start another instance of bash from the bash prompt:
  - `%` **`bash`**
  - `%`

- Such an instance of bash is an "interactive non-login shell". It reads `/etc/bash.bashrc` and `~/.bashrc`.

- Common practice:
  - (1) `~/.profile` loads `~/.bashrc`.
  - (2) All customizations are in `~/.bashrc`.

Shell variables

- bash supports variables that hold values of various types, including integers and arrays but we'll focus on string-valued variables, which are the default.

- bash variables have four common uses:
  - Specification of various bash behaviors
  - Access to information maintained by bash
  - Command line convenience variables
  - Use as a conventional variable for programming

- The variable `PS1` falls in the first category. Its value specifies the primary bash prompt. You may have noticed that the prompt looks different in the various shells I've shown, that's because this variable is set different.

- If we assign a value to `PS1`, the next prompt reflects it:

```
~ % PS1="What now, master? "
What now, master? PS1="C:>"
C:>
```

Variables that make information available

- bash makes a variety of current shell-centric information available as variables. Two simple ones are `PWD` and `OLDPWD`.

- Variables are accessed by prefixing their name with a dollar sign:
  - `% `**`echo $PWD`**
  - `/home/whm/352`

- Practical example with `OLDPWD`:
  - `% `**`pwd`**
  - `~/src/ruby/examples`
  - `% `**`cd ~/work`**
  - `% `**`cp $OLDPWD/fbscores.rb .`**

Command Line Variables

- You can also create your own vaiables.

```
% net="/cs/www/classes"
% echo $net
/cs/www/classes
% ls $net/cs210
fall17
```

The search path for commands (`PATH`)

- Your `PATH` variable specifies the directories that are to be searched when you run a command.

```
% echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/b
in:/sbin:/bin:/usr/games:/usr/local/rvm/bin:/ho
me/stdntwm/bin
```

```
% echo $PATH | tr : " " | wc -w
9
```

The search path for commands (`PATH`)

- When you type in a command each directory in the PATH is searched to see if an executable file of that name is there.

- The directories are searched in order from the first to the last.

- The command **which** will tell you where the command was first found (if it was)

Setting the path in `~/.bashrc`.

- Here's a possible line from `~/.bashrc`. What's it doing?

```
PATH=$PATH:~/bin
```

  - It's appendeing *"`:~/bin`" to whatever `PATH` already is.*

- When bash starts up, `PATH` gets set to a system-specific value.

- The net effect of `PATH=$PATH:~/bin` is "If you don't find a command in the standard directories on this system, look in my bin."

Dot danger?

- Some programmers have "dot" in their search path (`PATH=...:.`) so that a script `x` in the current directory can be run with "`x`" instead of "`./x`".

- "Dot in the path" can be convenient but imagine...
  - Instead of typing `ls` you accidentally typed `sl`.
  - You happened to be in a "world-writable" directory like `/tmp`.
  - A malicious student has put this in `/tmp/sl`:
    - `chmod o+rwx ~`
  - ...or maybe this:
    - `rm -rf ~/ &    #` *"&" runs a command "in the background"*

- In the first case, they've gained access to your home directory!

- In the second case, deletion of all your CS account files is in progress!

Dot danger, continued

- Should you include dot in your path?
- Like with any risk, you need to weigh risk vs. convenience.
- Pro: When developing scripts and programs, not having to type `./x` is convenient.
  - Your account on lectura has dot in the path by default
  - I have always have had dot in my path and never had a problem
  - On your own machine, the danger seems nearly nonexistent
- Con: People with a lot of experience think the danger is real and many recommend not including it.