Data Analysis in Geophysics
ESCI 7205

Bob Smalley
Room 103 in 3892 (long building), x-4929

Tu/Th - 13:00-14:30
CERI MAC (or STUDENT) LAB

Lab – 12, 10/3/13

Quotes for the day:

"Software stands between the user and the machine" - Harlan D. Mills

Software can help the user in their daily endeavors or stand in the way.

OS's at CERI

- Mac OS X (Darwin/UNIX)
  UNIX plus Mac GUI

- 10 Macs in Student Comptuer Lab in Long Building
  - many faculty offices.

OS's at CERI

- Solaris 9 UNIX

Misc SUN computers (not sure how many workstations left), many faculty offices

CERI compute servers (enigma, ??: all headless – no screen, have to ssh into them.)

OS's at CERI

- Solaris 9 UNIX

2 Graphical Desktop Environment options:

- Common Desktop Environment (traditional)
- - GNOME 2.0 (more PC-like)

OS's at CERI

- Various flavors of Linux

(dialects)

Popular, open source version of UNIX (often described as "UNIX-like", but is UNIX).

Found on a number of machines at CERI, but not officially supported at CERI.

Why learn Unix/Linux?

Designed to be

- multi-user (from the dark ages when all computers were shared),
- interactive (as opposed to "batch"), and
- multi-tasking (sharing again).

Why learn Unix/Linux?

- Invented by and for computer scientists/ system programmers

(not users or scientific programmers, unfortunately).

UNIX was designed to run on one of these!

It was amazing.

Unfortunately it has not evolved much.

Why learn Unix/Linux?

- Powerful, flexible, and small
- Hardware independent (myth!)

(these two points are much more important to manufacturers and designers than general users, i.e. us)

Why learn Unix/Linux?

- "Free" (this is why is it still around) from Bell Labs and Berkley.
- Open source – "free" – applications, including compilers.
  - Most common free applications designed as part of the GNU Project (GNU's Not Unix)

The real reason why to learn Unix/Linux?

- Because you have no choice

("Resistance is futile", The Borg, Star Trek).

- It is what is running in most geophysics (both university and corporate) departments.
- Most geophysics tools (SAC, GMT, GAMIT/GLOBK, etc.) only run on Unix (although there is a Windows version of GMT).

(~89% of the worlds computers run some form of Windows, ~10% run some form of the Mac OS, and ~1% run some flavor of Unix.)

Why learn Unix/Linux?

- "Free" in the sense you don't buy it from AT&T or Berkley
- But there is no such thing as a "Free Lunch".

Not "Free" in the sense that you must hire a system programmer/manager otherwise known as a UNIX Wizard or Guru.

(another UNIX myth shot down)

A bit of history

- Originally developed at AT&T in the late 60s/early 70s.
  - Freely given to universities in the 70s.
- Berkeley scientists continued to develop the OS as BSD Unix in parallel with AT&T (AT&T eventually licensed it for commercial use).
- Much development, branching, and combining has led to the most common variants of Unix ("flavors" or "distributions" in Unix speak).

- See http://www.bell-labs.com/history/unix/

Common flavors

- Solaris 9 Unix
  - Distributed by Sun Microsystems, runs on Sun Hardware, PC hardware.
  - Derived from Unix System V release (AT&T) on a Unix kernel.
- Mac OSX - Darwin
  - Distributed by Apple, runs on Mac Hardware.
  - Derived from BSD Unix OS on a Mach kernel - Darwin.
- Linux – lots of 'em
  - Free* and commercial# versions available built on a Linux kernel.
  - Flavors most likely to hear about are RedHat#, Ubuntu*, Fedora*, Debian*, Suse*,....

Does this matter?

- No, the differences between the various flavors of the Unix operating system should not severely affect your work in this class or even much of your research at CERI.

BUT

Does this matter?

- Yes, you need to be aware of OS differences
  - When file sharing with others (this is more of a hardware, rather than an OS, issue).
  - When compiling source code (the executable file is married to hardware).
    - If sharing programs, shell scripts, etc. with others.
    - Or if moving between the different systems at CERI.

Relation to Windows

None.

Windows XP

Built on MS-DOS (which is not really an operating system, it is a file system), which has nothing to do with Unix and everything to do with Microsoft.

Cygwin – unix/linux like environment for windows. Have to build everything from source.

Relation to Windows

The differences between the Unix Philosophy and the Windows Philosophy … can be boiled down into a question of smarts … .

Unix and Windows store the smarts in different places.

Unix stores the smarts in the user.
Windows stores the smarts in the OS.

Learning curves

Enter the concept of the “Learning Curve”. ...

A "steep" learning curve generally refers to something that requires a lot of initial learning to do anything, even something very simple.

A "shallow" learning curve is exactly the opposite; can do simple stuff easily immediately.

Learning curves

Armed with those definitions, it's fairly simple to then go ahead and say that Unix has an inherently steep learning curve, and Windows has a very shallow one.

Windows

Our Microsoft brethren have taken the approach of making the shallowest possible learning curve.

Windows

To take a cue from the fast food industry, Windows is the "under-3" toy of the OS world.

The ultimate goal is to flat-out destroy any barrier to entry by removing any requirement for initial knowledge or learning of how and why, and of making the system simplistic enough that it can be used without any understanding of how it works.

Unix

The Unix crowd has taken the opposite approach.

Unix

Unix has a steep learning curve; it doesn't shield the user from complexity; rather, it revels in the complexity.

It recognizes that a general-purpose computer is a fiendishly complicated device capable of doing an unbelievable assortment of things.

Unix

It recognizes that the computer is a tool of the user, and so takes a tool-building philosophy.

Make a lot of tools, and make each tool specific, and let the user select the tool they think appropriate, and let the user combine the tools however they want.

It's not aimed at making things easy; it's aimed at making things possible.

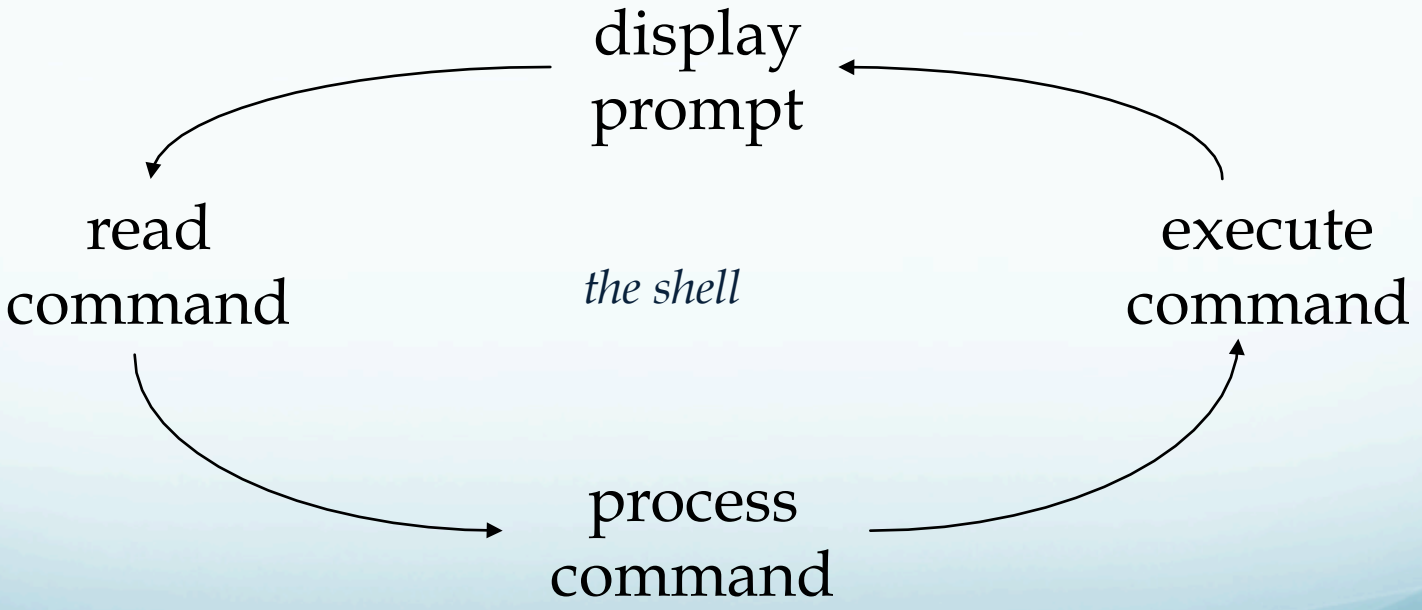
UNIX Philosophy

(Mac) (Unix)

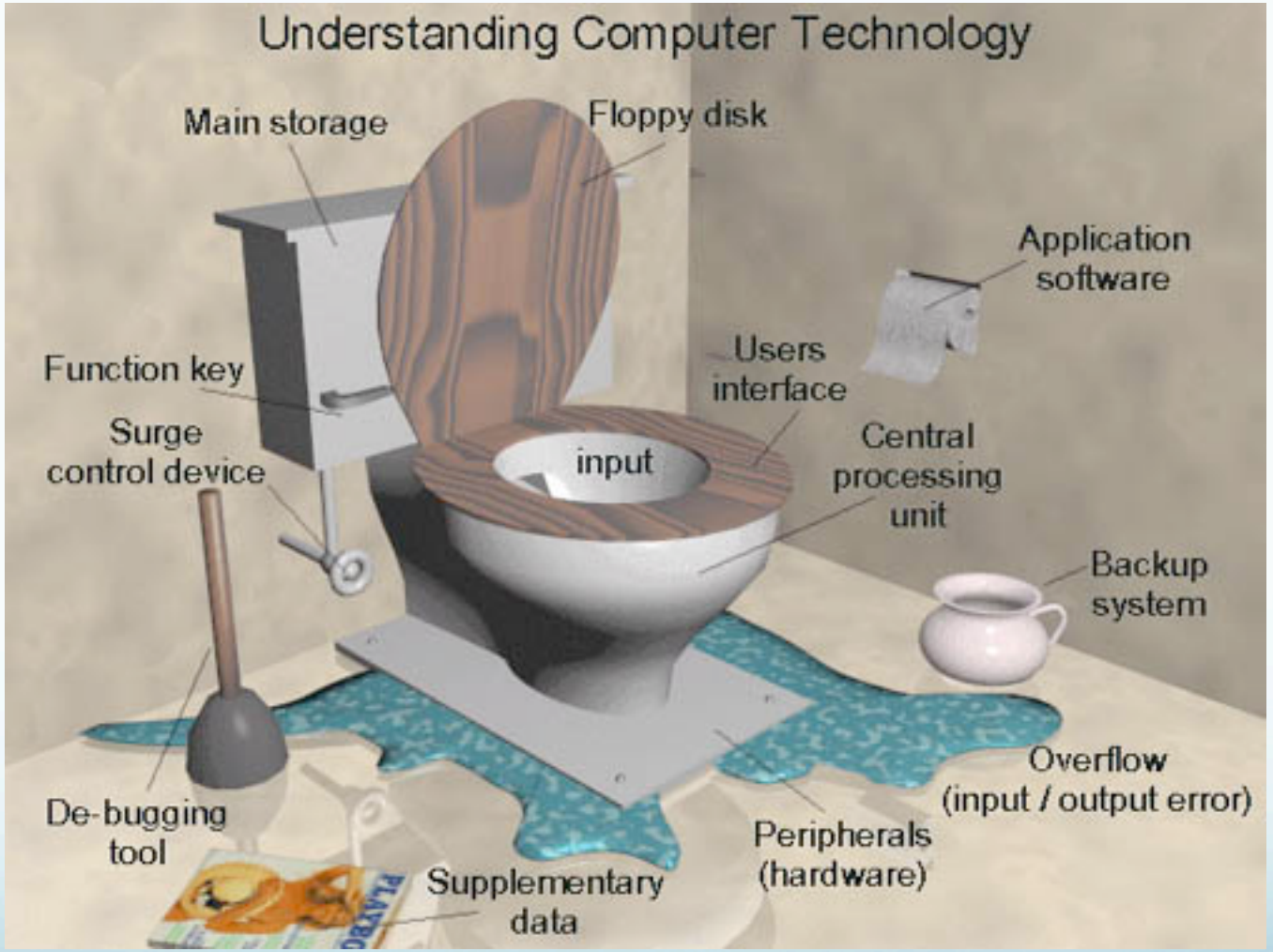
"Dilbert" by, Scott Adams, Sep 30, 1994.

The Shell

- The UNIX user interface is called the *shell*.
  - The shell does 4 jobs repeatedly:



Final Model


Understanding Computer Technology
Main storage
Floppy disk
Application software
Function key
Users interface
Surge control device
input
Central processing unit
Backup system
De-bugging tool
Overflow (input / output error)
Peripherals (hardware)
Supplementary data

We will now take a short detour to examine the Unix philosophy.

It will keep returning to haunt us, but if you understand it, it will make the process less painful.

What is the "Unix Philosophy"?

(can computer operating systems have a "philosophy"?)

According to Doug McIlroy

(i) Make each program do one thing well.

So, to do a new job, build afresh rather than complicate old programs by adding new features (otherwise known as "bells and whistles").

What is "Unix Philosophy"?

Machine shop vs. appliance

(gives you the tools and you to make appliance)

What is "Unix Philosophy"?

Advantage

- POWERFUL

What is "Unix Philosophy"?

Disadvantages

- Lots of reinventing the wheel
- Requires a more educated user
- Requires more work from the user rather than the developer

What is "Unix Philosophy"?

Typical question: can UNIX do this?

Typical answer: NO, but YOU can write a program!

Unix enthusiasts think this is the answer the average user wants to hear!

Caricature of UNIX vs Windows

If you need a washing machine

Windows gives you a simple washing machine (only one 1 setting, you shouldn't wash your cashmere sweater, but there are no operating instructions [its intiutive] so you probably don't know that and ruined it.)

UNIX gives you a machine shop [you better know 1) how wash clothes and 2) how to design and build a machine to do it].

"UNIX Philosophy"

(ii) Expect the output of every program to become the input to another, as yet unknown, program.

- Don't clutter the output with extraneous information useful to the user, but not needed by the input for next program.

"UNIX Philosophy"

Unfortunately this may make things confusing for the uninitiated user.

The output is for "next program" (in a "pipe"), not the user.

"UNIX Philosophy"

Idea of "filter" –

Every program takes its input from Standard-IN (originally a teletype, now a keyboard),

does something to it ("filters" it) and

sends it to Standard-OUT (originally a teletype, now a screen)

(notice that the "user" is not part of this model).

"UNIX Philosophy"

It is pretty easy to see these are not good assumptions (Stnd-IN, Stnd-OUT) for many tasks and we need a way around it
(in addition, many Unix commands break this convention).

"UNIX Philosophy"

Idea/use of – redirection ("<", "<<" and ">", ">>")

- Take input from a file rather than Standard-IN

- Send output to a file rather than Standard-OUT

(Unix treats everything like a "file", even hardware)

"UNIX Philosophy"

What happens when you ask for a listing of files in an empty directory?

```
$ ls<CR>
$
```

Returns to prompt without any other output. (there are no files to list, so Unix outputs nothing [and a new prompt], is that reasonable?). (Works differently in shell script)

"UNIX Philosophy"

What happens when you enter

```
$ echo<CR>
```

The command echo, "echoes" what you type. Should do nothing! (what about a new prompt on same line?)

(i.e. it should just sit there, with the "cursor", which was invisible on a teletype after the <CR>, waiting for input)

"UNIX Philosophy"

What happens when you enter

```
$ echo<CR>
```

Usually goes to next line and prints the prompt on the screen as in the previous example (break with philosophy because philosophy too confusing)

```
$ echo<CR>
$
```

But does what expected, nothing (it follows philosophy), in a shell script.

"UNIX Philosophy"

This brings up another issue – commands sometimes behave differently in shell scripts than they do "interactively"

Typically more "chatty" when interactive.

This kind of stuff can make for confusion when debugging. Works from screen, does not work in shell script.

"UNIX Philosophy"

Idea/use of – pipes (" | ")

Sends output to the next program (instead of "standard out" or a file)

And

Takes input from the previous program (instead of "standard in" or a file)

"UNIX Philosophy"

Example: we have two files with a name and student ID on each line.

There are some duplicates (i.e. exact same line, character for character, in both files).

We want one file, in alphabetical order, with duplicates removed.

```
cat file1 file2 | sort –u > file3
```

(cat does not require input file redirection, it will take a list, redirection does not even work with more than one file)

Write programs to handle text streams, because that is a universal interface.

(fine if you're a system programmer, not always so useful for scientific data crunching.

Good example of a real problem that does not follow this model is earthquake location.

You typically have one static text file for station locations, another static text file for the velocity model, and

a final text file with station names and arrival times for an earthquake.

This does not fit the serial, filter model.

Another example, binary seismic, topo, etc. data.)

"UNIX Philosophy" Continued

Avoid stringently columnar or binary input formats.

(Avoid, but sometimes necessary. Not closely followed by many programs. Stupid restriction – especially given UNIX "you can write a program to do that" philosophy)

Don't insist on interactive input.

([User} Does not fit in with use of pipes.)

Instead, control is implemented by use of "command line switches"

"UNIX Philosophy"

Put lots of (simple, easy to write) single minded programs in a row (with pipes) to do what you need.

(Don't use temporary/intermediate files – use a pipe).

"UNIX Philosophy"

New concept

use of – command substitution (`...`)
(uses "backwards" or French grave accent)

Use the output of a command as 'some sort of input' to another command.

command substitution example.

```
echo `pwd`
```

Executes the command **pwd** and then uses the output as the input to **echo**.

(this is not how you would actaully do this – it is a ginned up example. If you wanted to know this you would just use **pwd**.)

(you cannot nest them - `` `echo `pwd` `` - does not work)

REVIEW

Write programs that do one thing and do it well.

(lean and mean)

Write programs to work together.

(pipes)

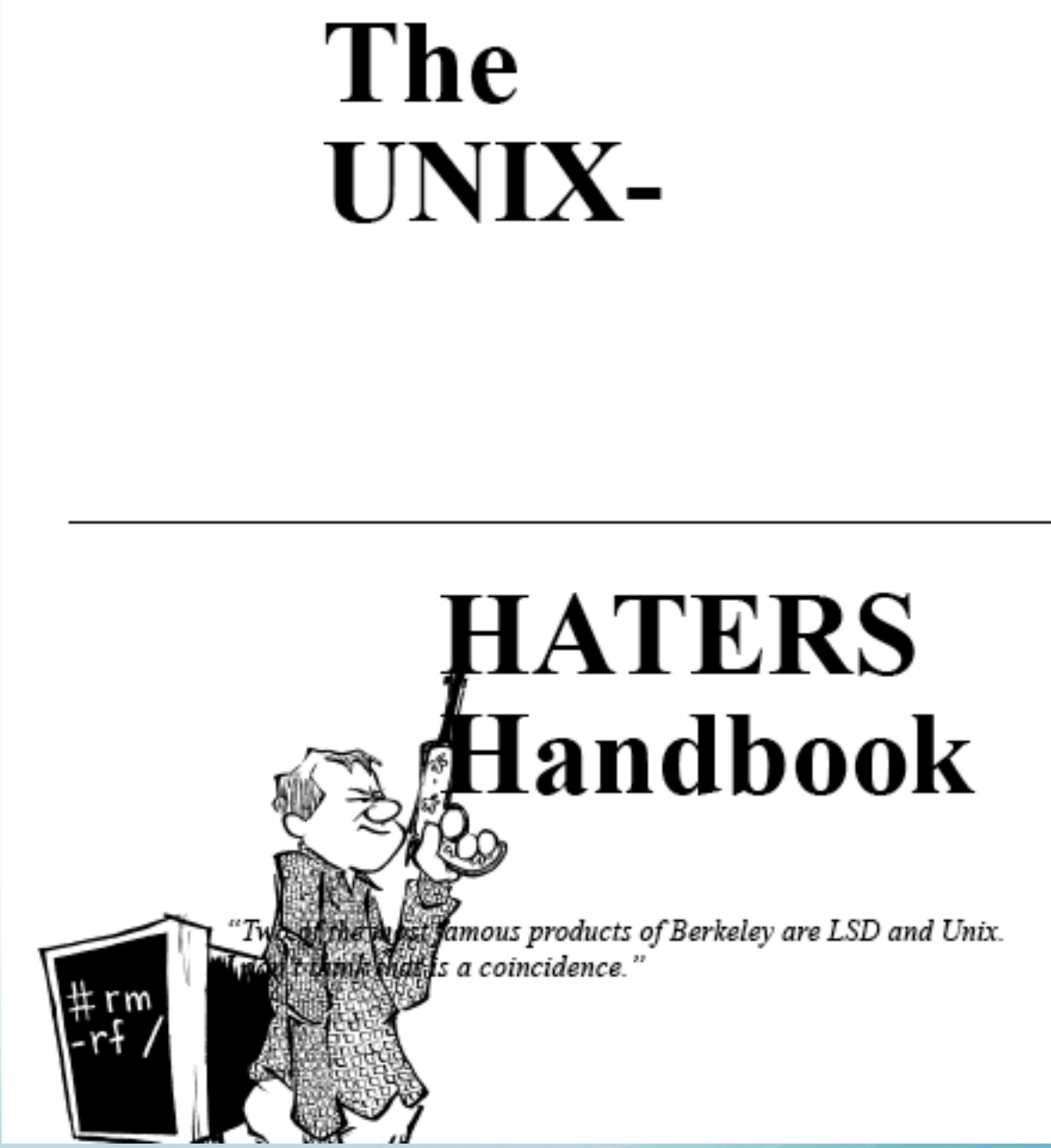
"the UNIX operating system, a unique computer operating system in the category of help, rather than hindrance."
Introducing the UNIX System
McGilton and Morgan, 1983.

or

The trouble with UNIX: The user interface is horrid
Norman, D. A. *Datamation*, 27, No. 12, 139-150.

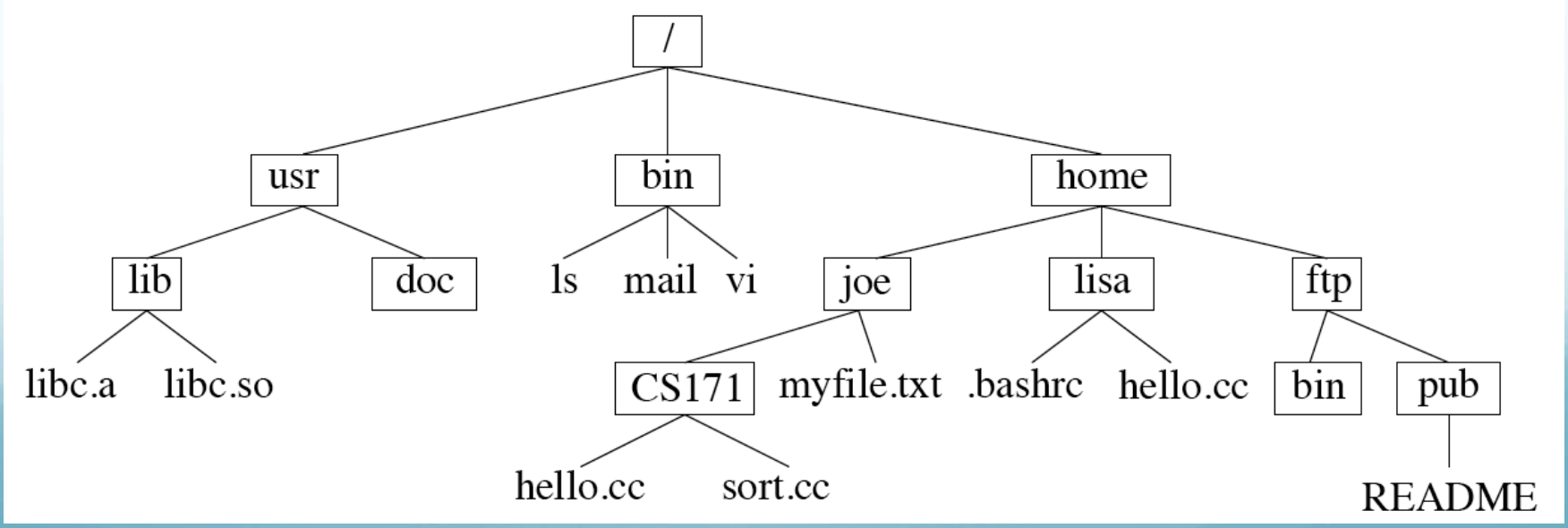
"Two of the most famous products of Berkeley are LSD and Unix. I don't think that this is a coincidence."

Anonymous



Before looking at more Unix commands, we will first look at the FILE STRUCTURE (how files [called documents on Mac and Windows] are stored/organized).

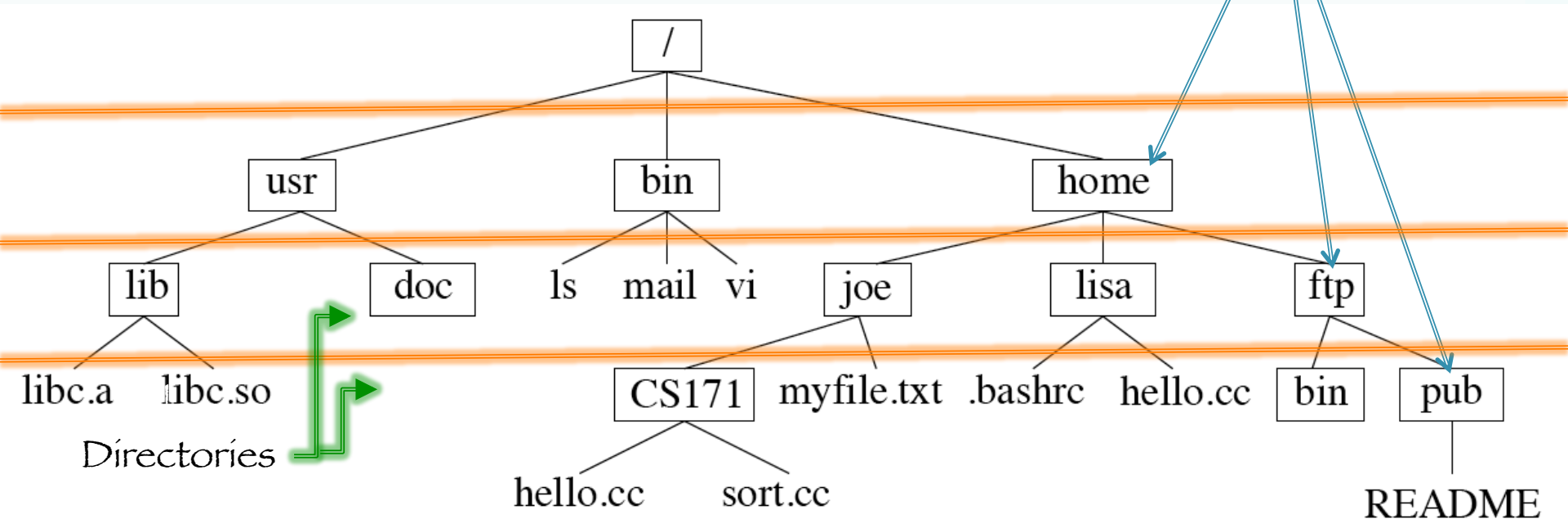
Unix uses a hierarchical file system (as does Mac and Windows/DOS).



Looks like an upside down tree.

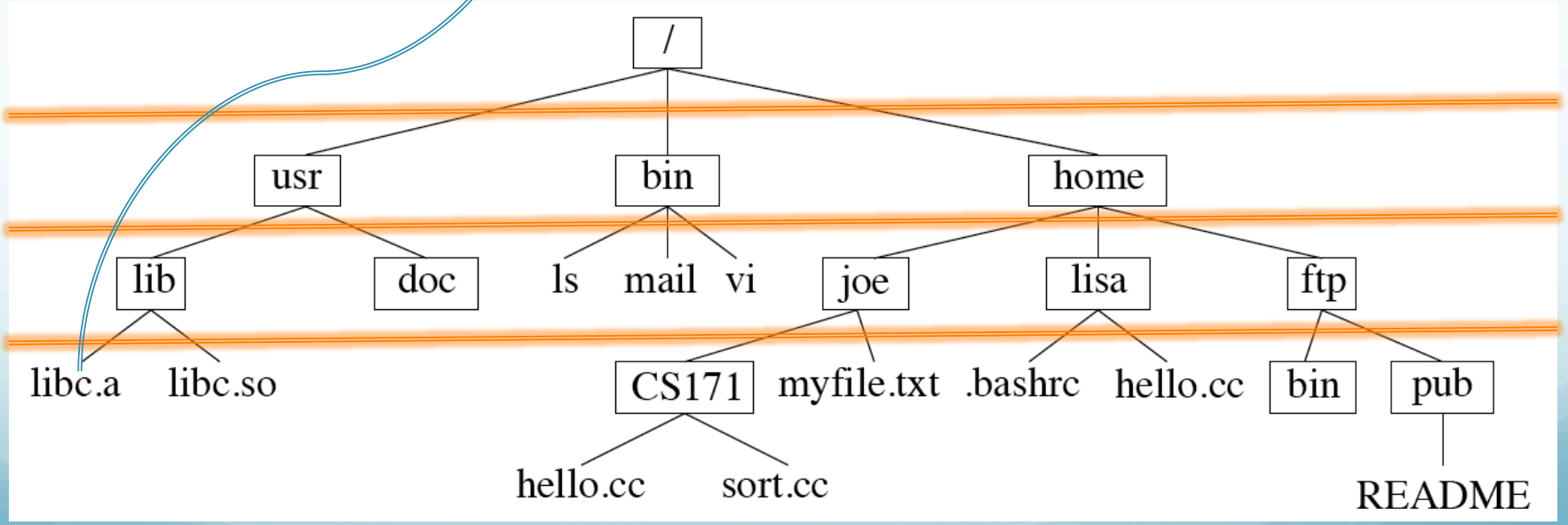
Starts at top with "/", called "root".

Unix uses the "/" to separate directories (known as folders on Mac or Windows)



File names – the "separator is "/".
root (first slash) then path and filename

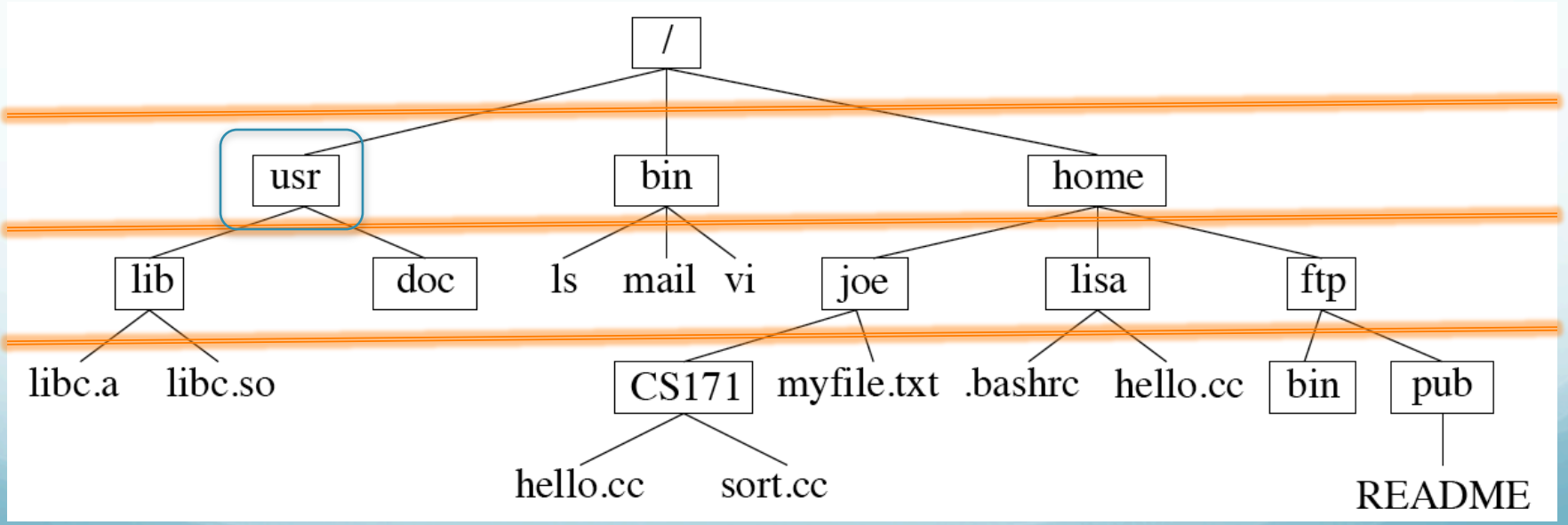
/usr/lib/libc.a



This is the full name from root (works from anywhere – i.e. any directory), if you were in the directory usr, you only need

`lib/libc.a`

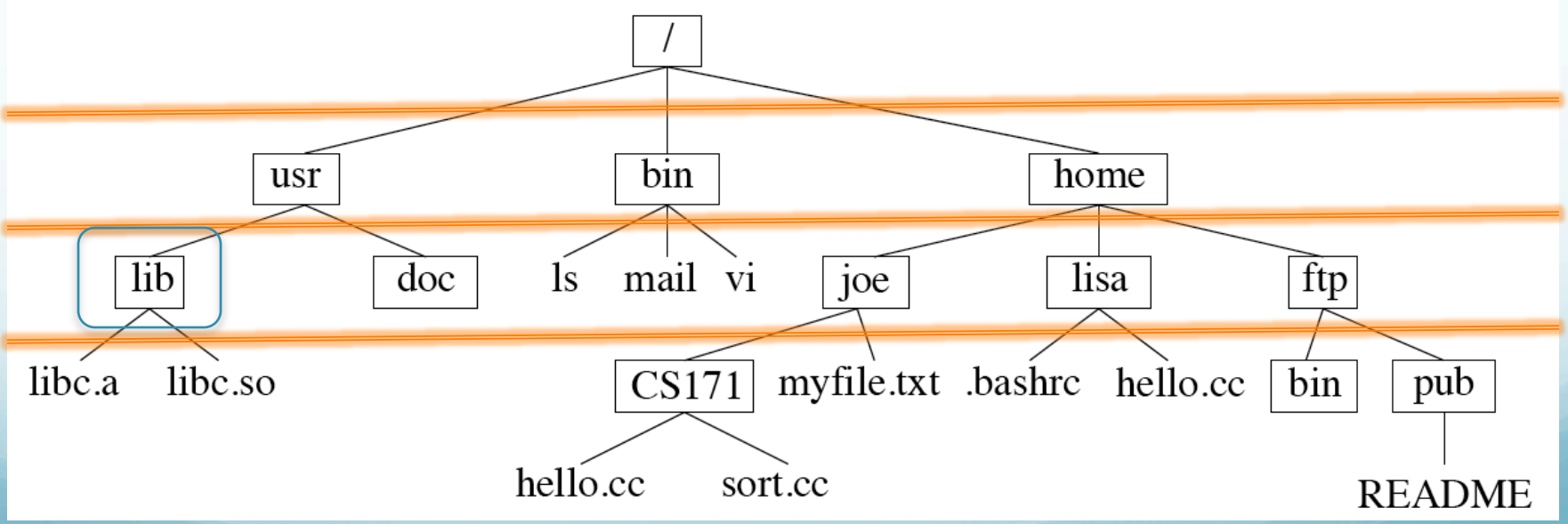
(no leading slash)



And if you were in the directory lib, you only need

`libc.a`

(no leading slash)

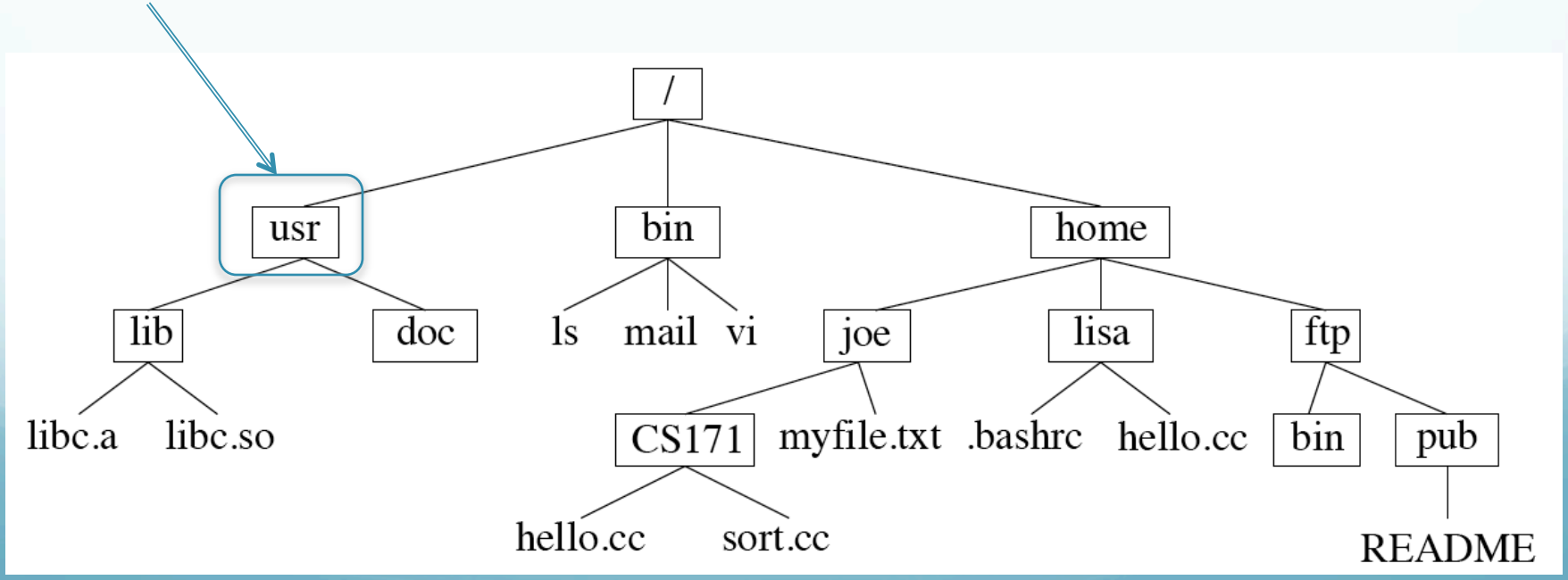


The "/" (slash or forward slash) in Unix is roughly equivalent to the "\" (backslash) in Windows/ DOS.

Some commands:

`pwd` – print working directory – tells us where we are in the directory tree.

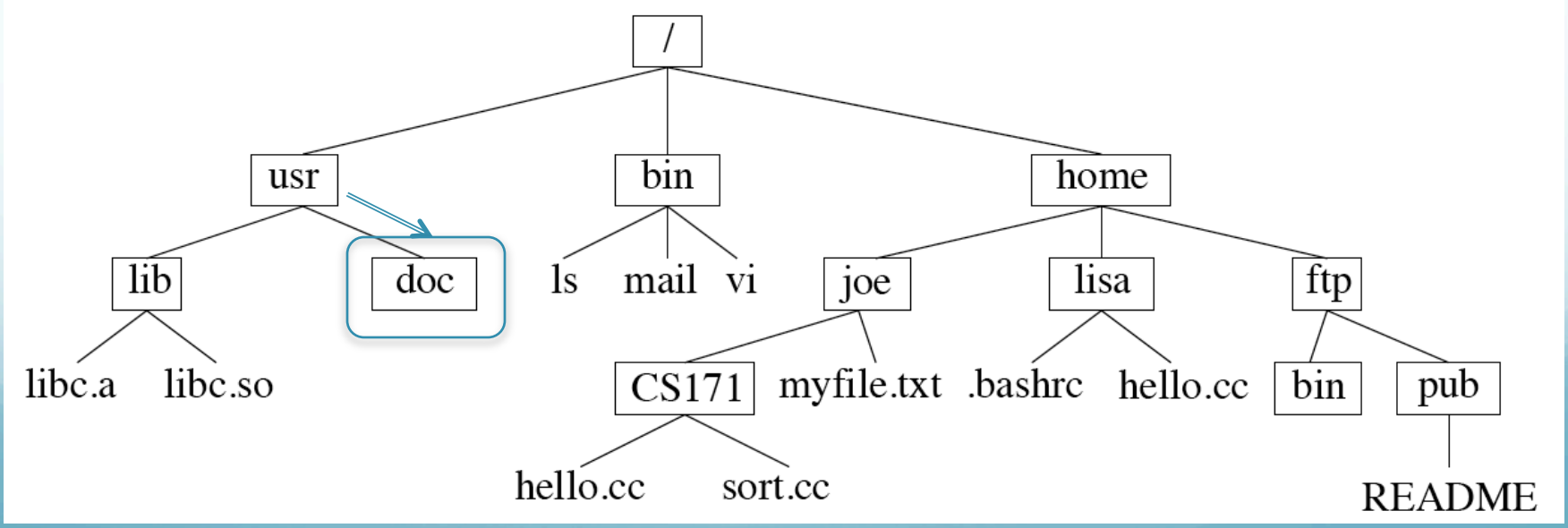
```
$ pwd<CR>
/usr
$
```



How to move between directories – going up and down the directory tree–

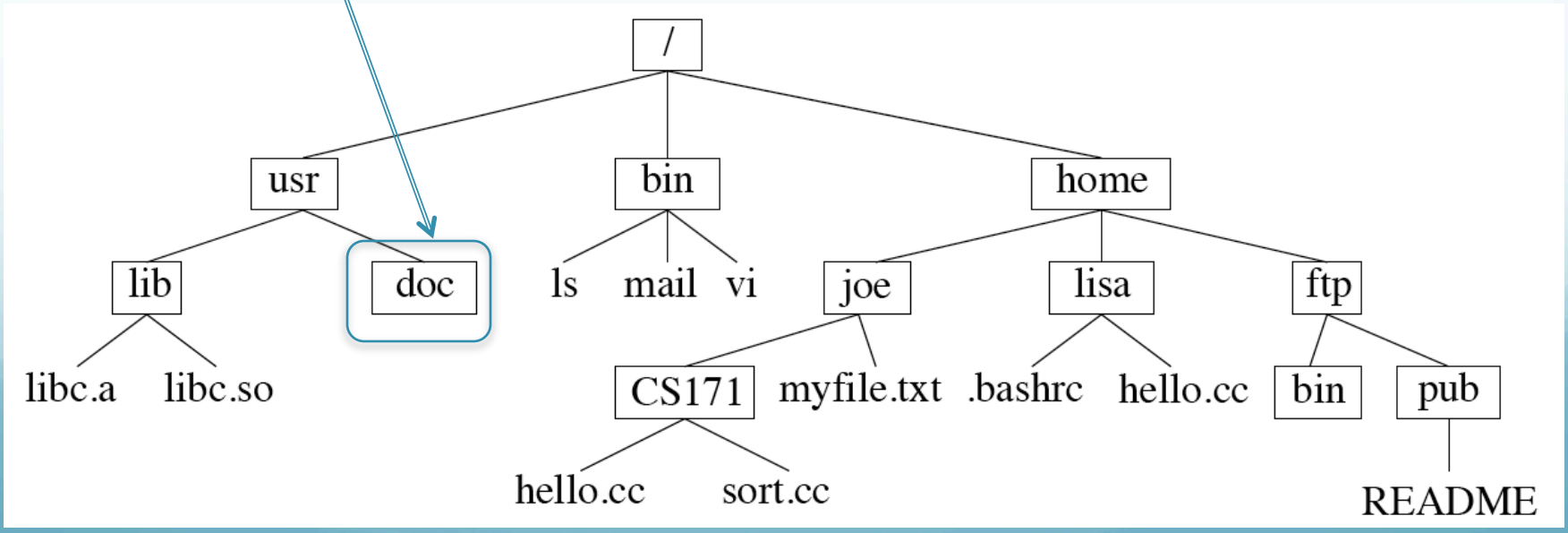
To go down to the directory doc we use the "change directory" = "**cd**" command

```
$ cd doc<CR>
$
```



Now

```
$ pwd<CR>
/usr/doc
$
```



Some details of the prompt

you can control all this - the prompt has been programmed to tell us a bunch of stuff! (power of unix.)

Machine
Directory name (in this case without full path)
User name
Text string

```
smalleys-imac-2:usr smalley$
```

Aside:

Unix sub philosophy –

Minimize typing (on teletype) – so use short (2, in extreme cases 3 character) command names constructed from description of the command.

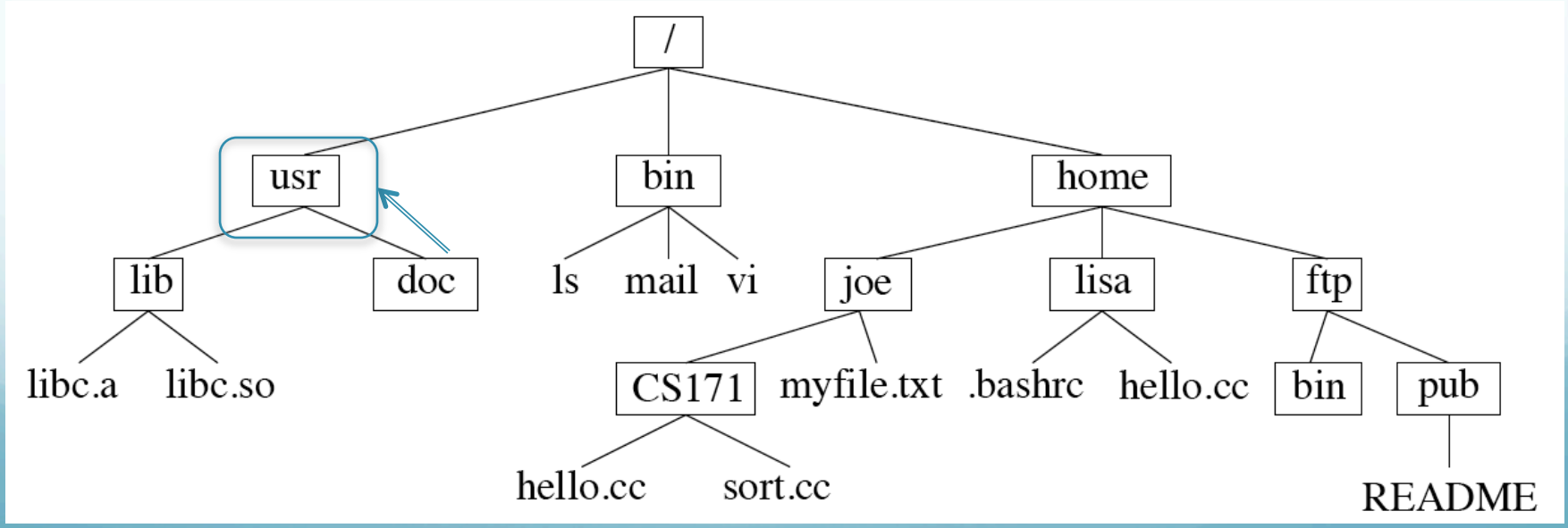
e.g. "cd" for "change directory"

(Unix fans claim this is a "feature" of Unix, compared to other O/Ses)

We can also go up the directory structure.

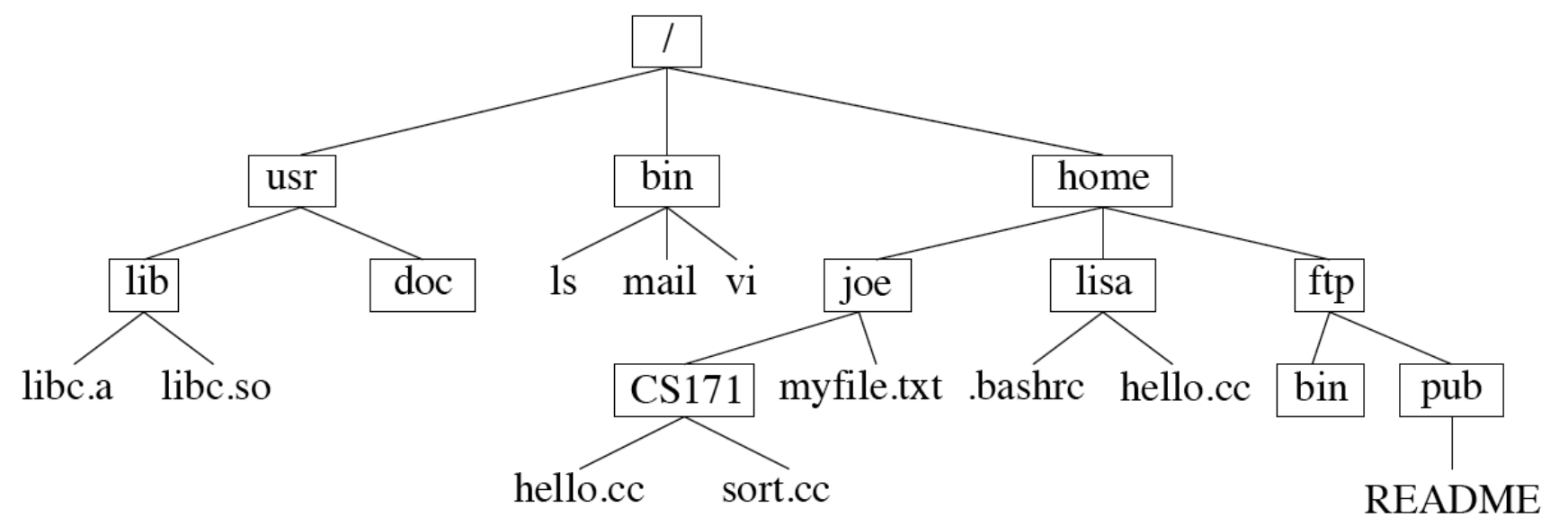
To return to usr from doc.

```
doc$ cd ..<CR>
usr$
```



This is a little strange ---

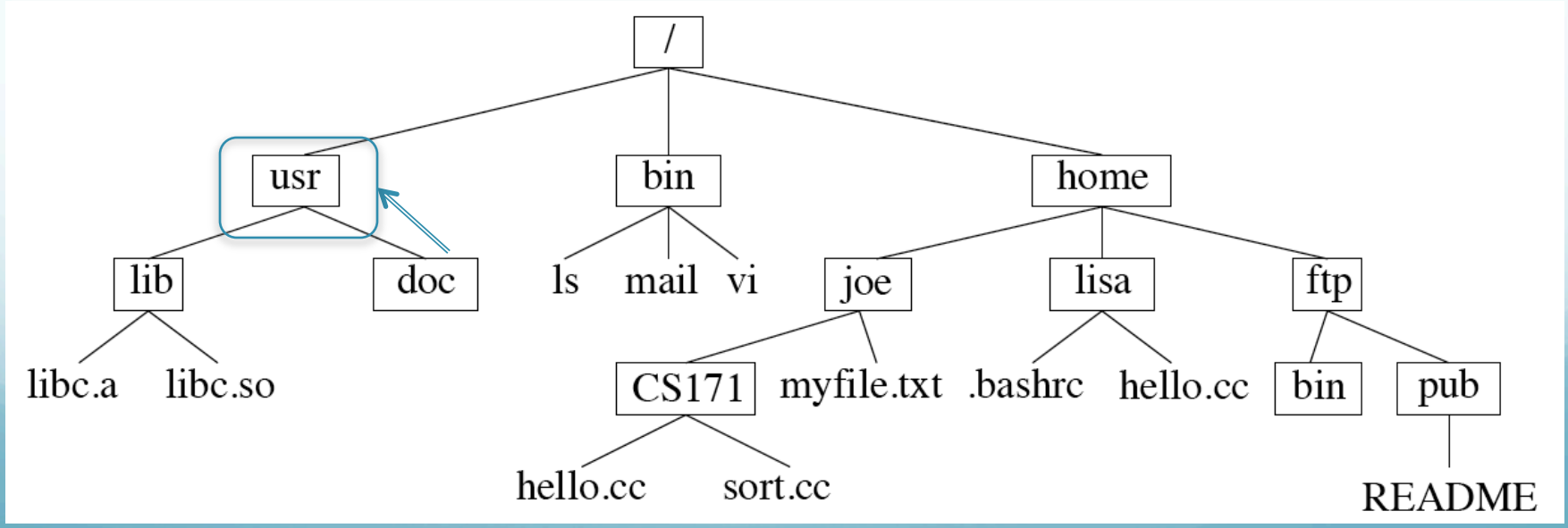
The double dot ("..") signifies the directory directly above you (up) in the directory structure (tree).



We can also go directly to anywhere in the directory structure using the full path.

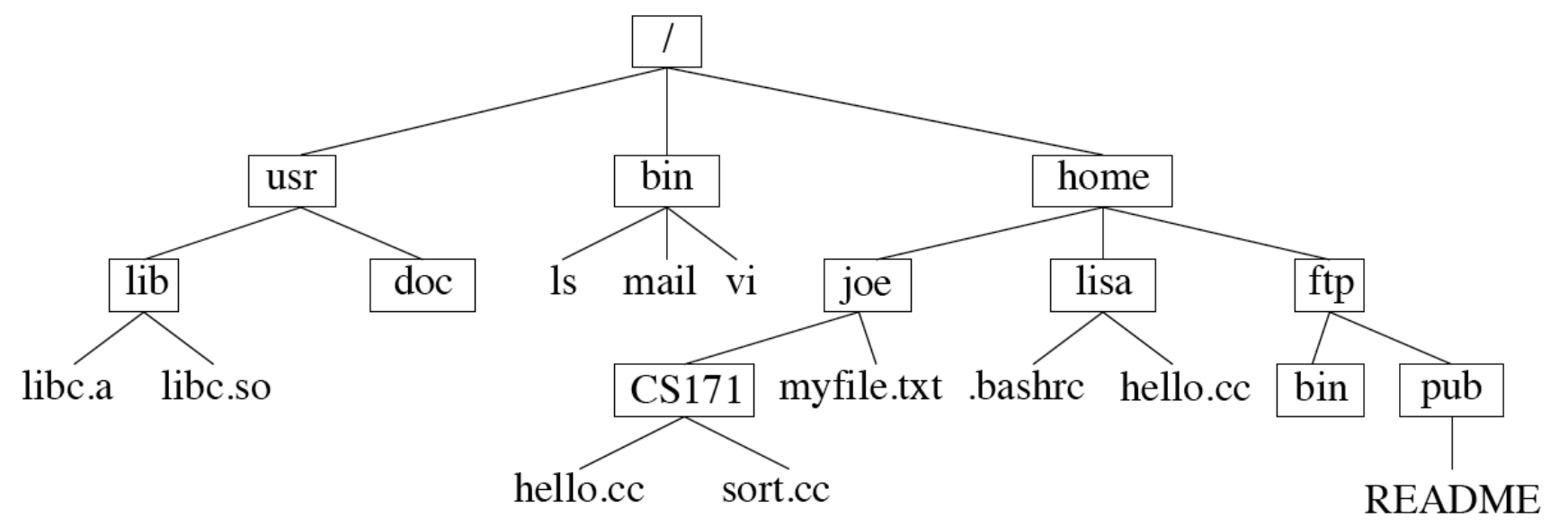
To go to usr (from doc or anywhere, such as pub)

```
doc$ cd /usr<CR>
usr$
```



Notice that you have to know where you are in the tree and what subdirectories are contained there to navigate down.

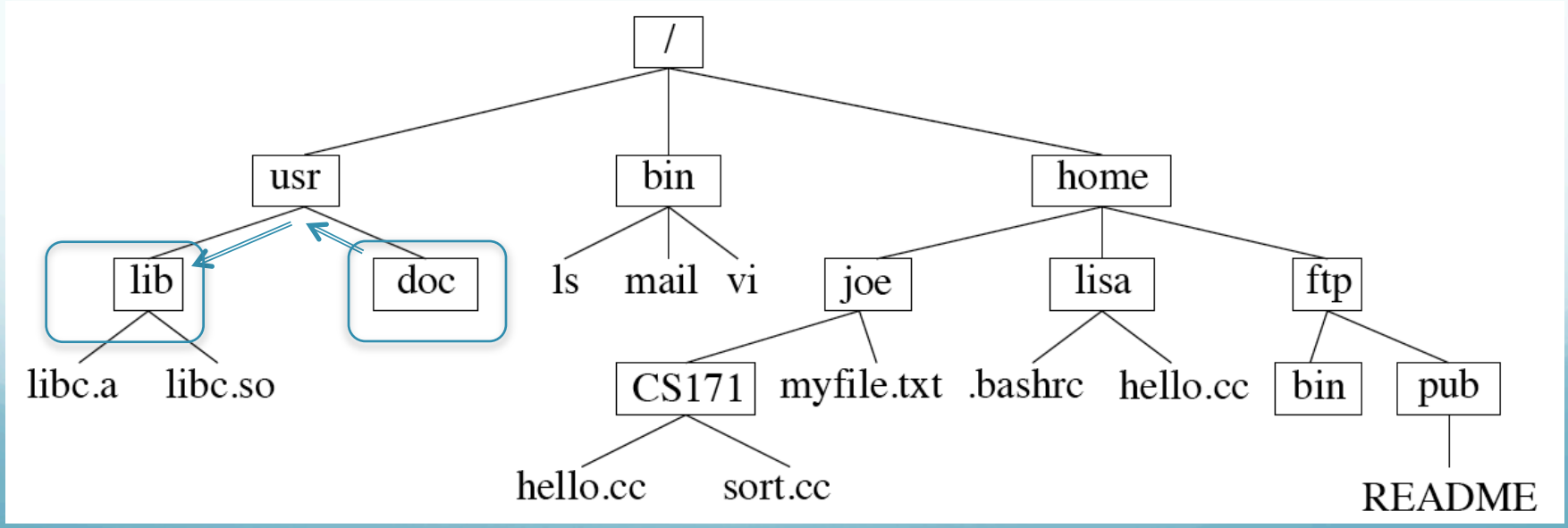
Unix does not provide a display of the picture below. You need to have it in your head.



How do we go from doc to lib?

We could do this using the full path.

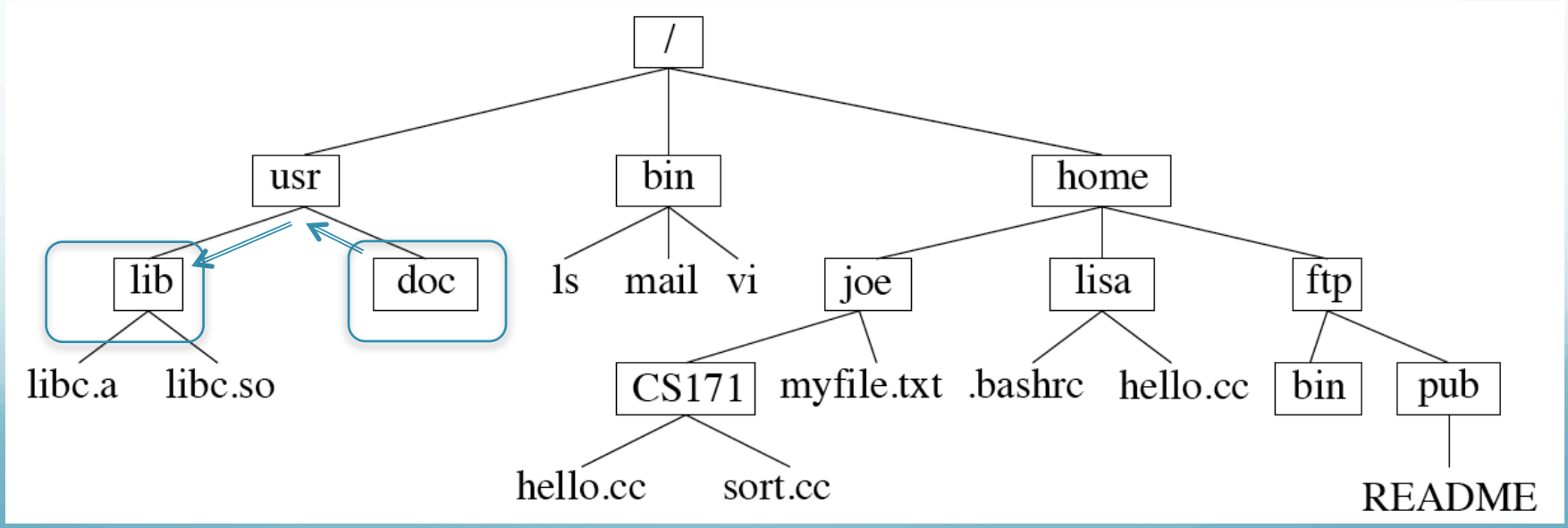
```
doc$ cd /usr/lib<CR>
lib$
```



How do we go from doc to lib?

But here's an easier (?) way – we have to go up one level; then down one level. This can be done with the command.

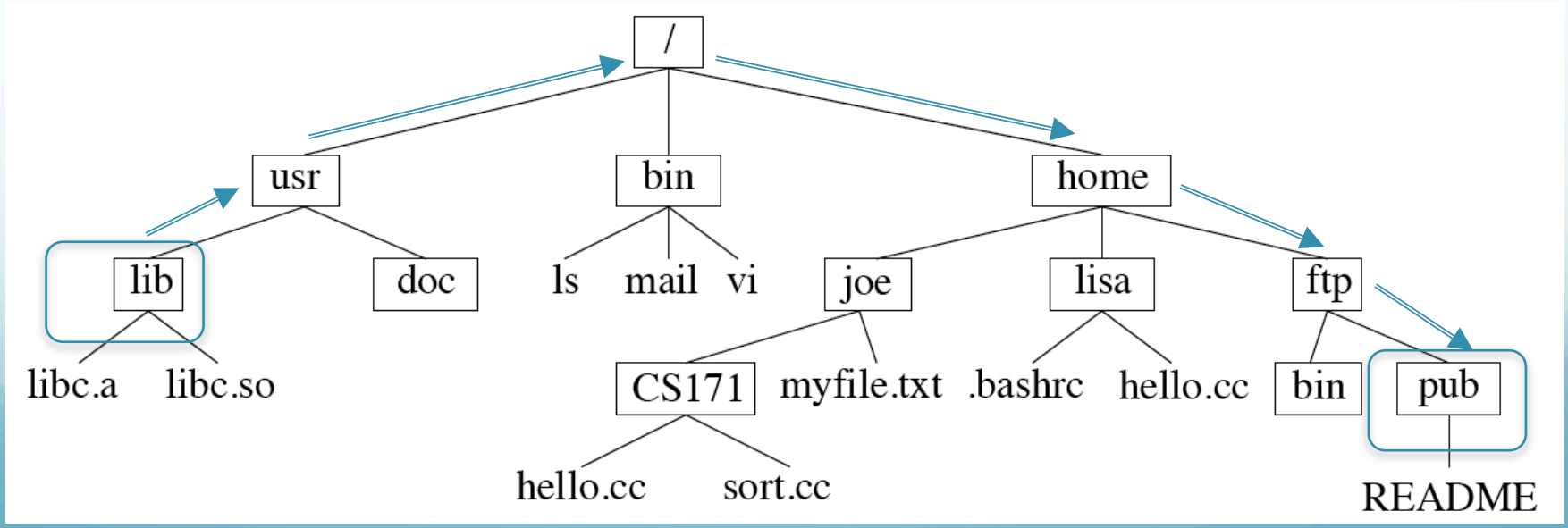
```
doc$ cd ../lib<CR>
lib$
```



Say we want to go to "pub"

```
lib$ cd ../../home/ftp/pub<CR>
```

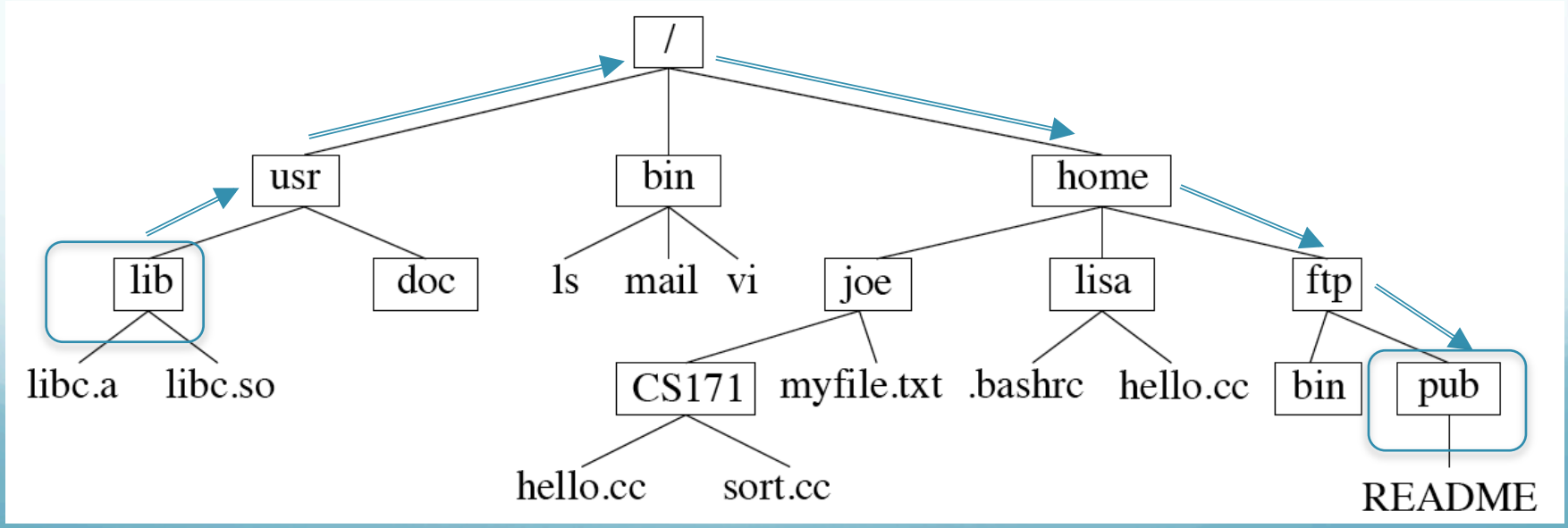
We went up two, then down three.



Say we want to go to "pub"

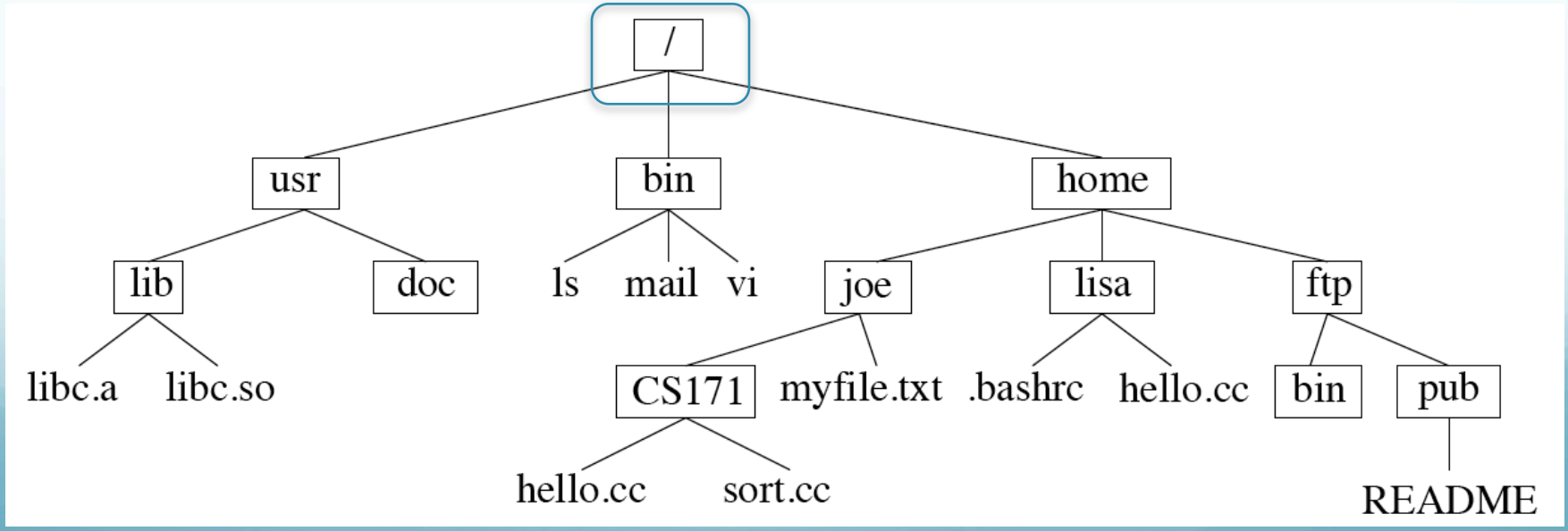
We could also have done (and is simpler) this with the full path.

```
pub$ cd /home/ftp/pub<CR>
```



Go directly to "root" directory ("/")

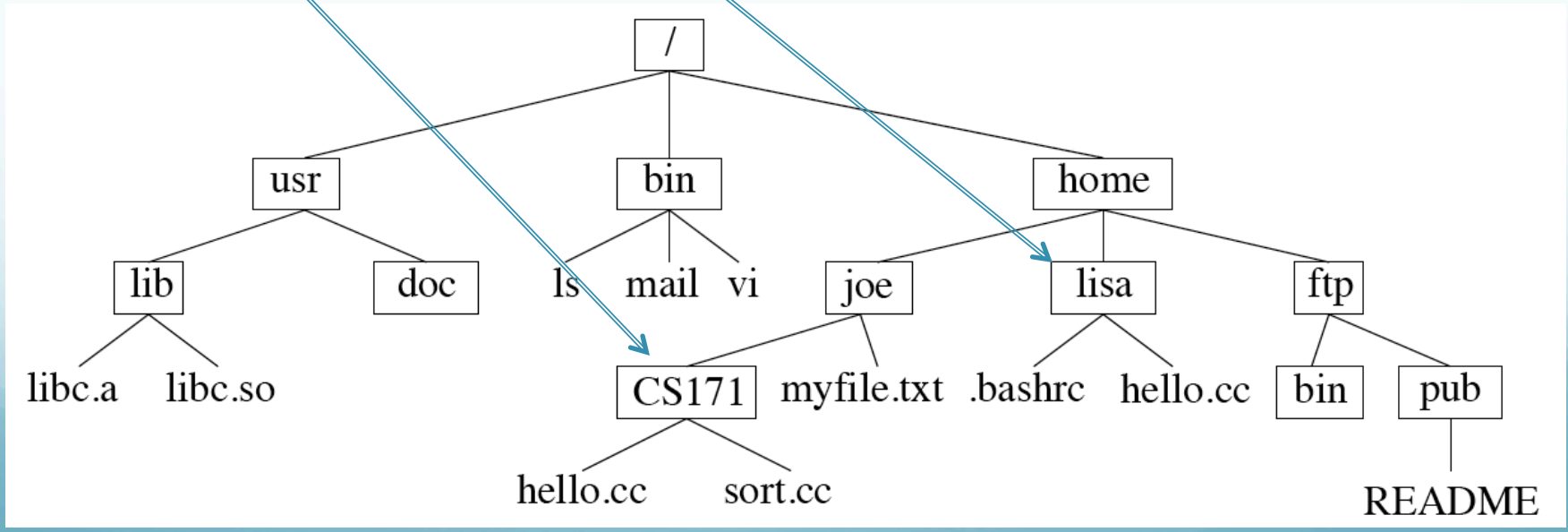
```
lib$ cd /<CR>
/$
```



Go from anywhere directly to your "home" directory (assume I'm "lisa").

```
CS171$ cd ~<CR>
lisa$
```

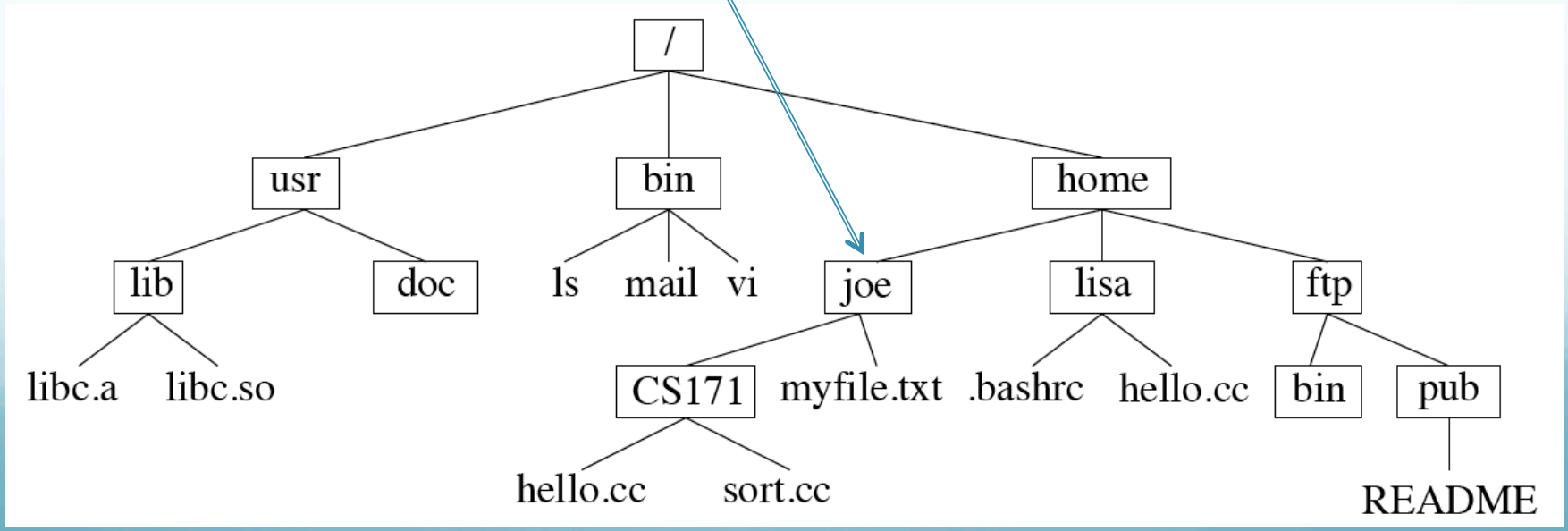
uses tilde "~"



Go from anywhere directly to someone else's home directory (assume I'm lisa)

```
lisa lisa$ cd ~joe<CR>
/home/joe lisa$
```

also uses tilde "~"



The tilde character "~"

~refers to your home directory when by itself,

~ or that of another user when used with their home directory name (the same as their user name).

(The shell expands the "~" into the appropriate character string for the full path - "`/home/joe`" or "`/home/lisa`")

The single dot " . "

-refers to the current directory

```
ls ./somefile
```

Review - specifying file names

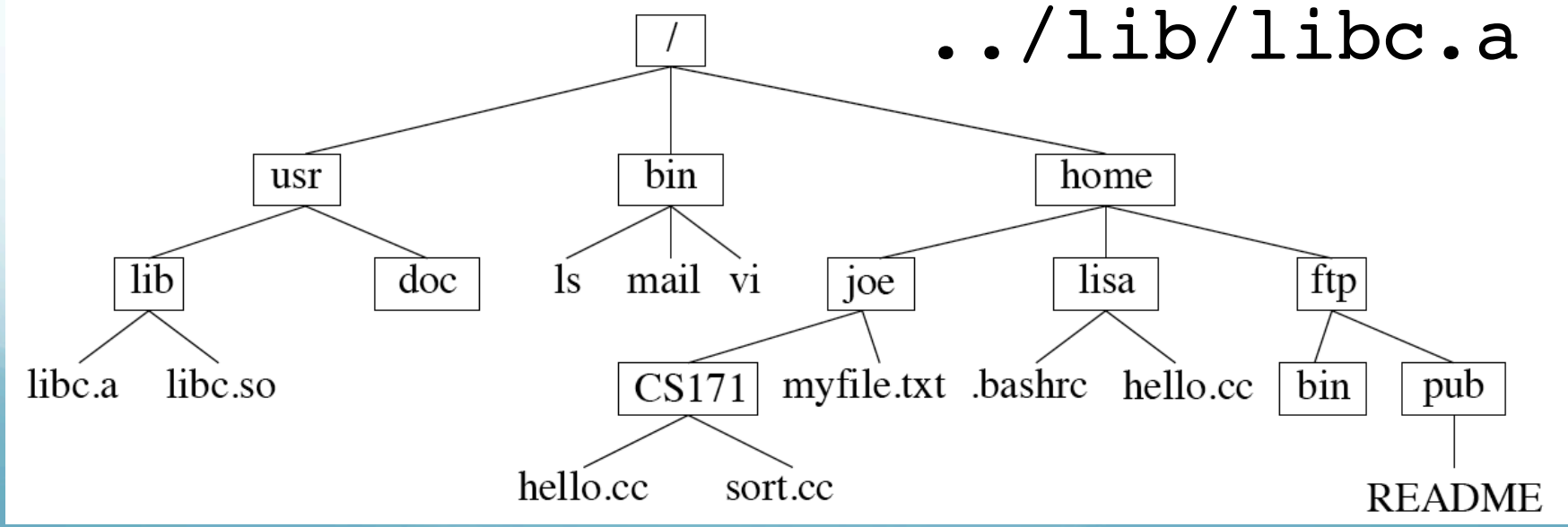
full path

`/usr/lib/libc.a`

relative path

(if in directory lib) `libc.a`

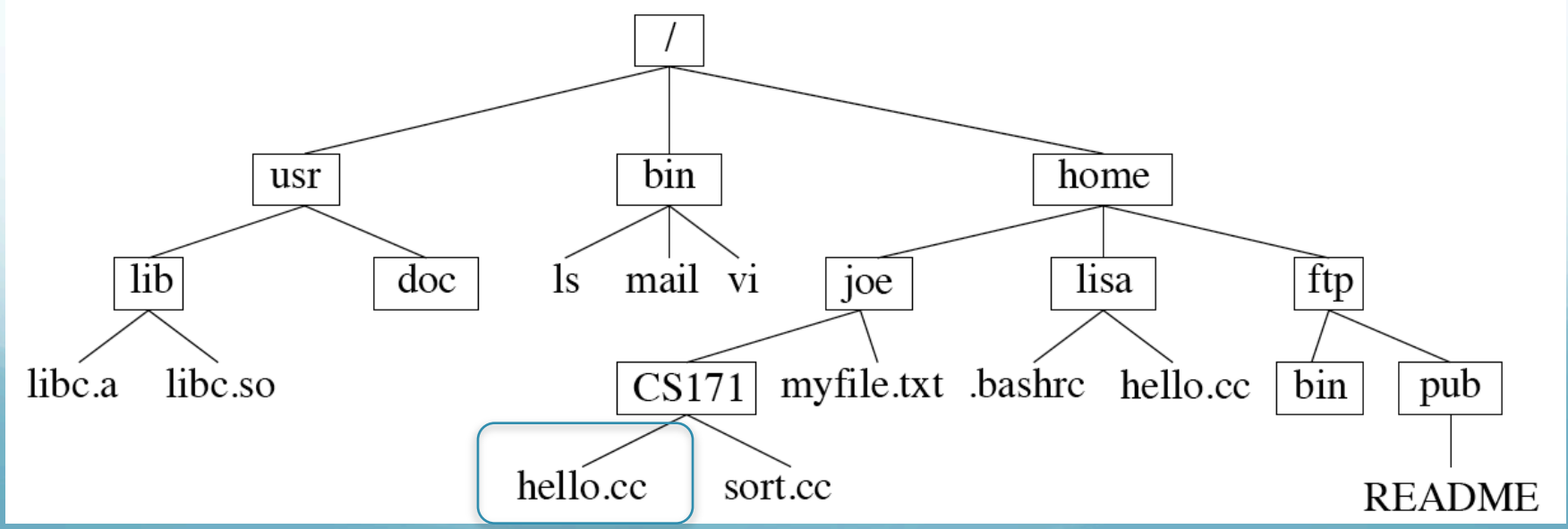
(if in another directory next to it, e.g. doc `../lib/libc.a`



Review - specifying file names abbreviations

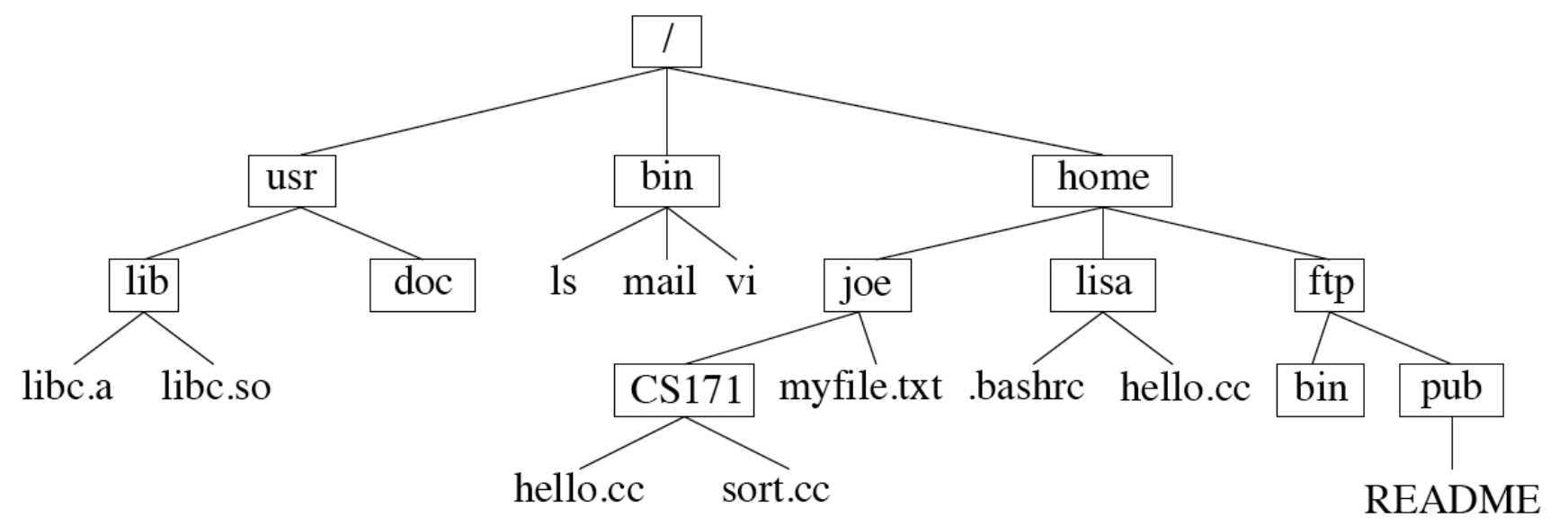
(if I am joe) `~/CS171/hello.cc`

(if I am not joe) `~joe/CS171/hello.cc`



You have to keep track of the file structure in your head

or have a way to find out what files are in the working directory.



What files are in working directory?

Use the "list" command, which is actually "`ls`".

(Compare this to VAX-VMS, a professional O/S with 100 man-years of development, which uses "directory" – much longer)
(but Unix supporters forget to tell you that it can be shortened, using "smart abbreviating", by dropping letters off the back, to

"director", "directo", "direct", "direc", "dire", or "dir"

at which point continued shortening stops as "di" is non-unique as another command (differences) also begins with the letters "di".
This means you can write "com" files – same as shell scripts or batch files – to be readable using "directory" or cryptically using "dir".)

Listing working directory (where we are) contents with "ls" command.

```
$ ls<CR>
Adobe SVG 3.0 Installer    vel.dat
Desktop                    heflen_web.dat
Documents                  isc0463.dat
Downloads                  nuvel-1a.dat
gpsplot.dat
$
```

Note the file

`Adobe SVG 3.0 Installer`

has spaces in the name.

On Unix this is somewhat of a problem.

Spaces are allowed in filenames in Unix (all characters but the "/", which we have seen means something special in a filename, are allowed in filenames!) , but spaces, and special characters

```
-!@#$%^&*()_+|?><`[]{}\'":;
```

are not handled nicely as most of them also mean something special (not related to file name) to the shell.

The problem with spaces is that the command interpreter of the shell parses (breaks) the command line up into tokens (individual items) based on the spaces.

So our file name gets broken into 4 small distinct character strings ("Adobe", "SVG", "3.0", and "Installer") which causes confusion since there are no files by those names.

So we have to "protect" the spaces from the interpreter.

This is done with quotes.

We refer to this file using

" Adobe SVG 3.0 Installer "

or

' Adobe SVG 3.0 Installer '

(We will see the difference between single, ' , and double, " , quotes later.)

`ls`: lists files and subdirectories of the specified path.

```
%ls /gaia/home/rsmalley<CR>
bin  src usr world.dat

%ls<CR>
```

lists everything in the current directory

```
%ls ~/bin<CR>
```

lists everything in <u>your</u> bin directory (not the system bin directory `/bin`).

`ls`: getting more information than just file name.

Use a "flag" to give the "`ls`" command control inputs.

Use "`-F`" to obtain kind of file – list directories with '/' and executables with '*' following them.

```
$ ls -F<CR>
Adobe SVG 3.0 Installer  vel.dat
Desktop/                 heflen_web.dat
Documents  /             isc0463.dat
mymap.sh*                a.out*
Downloads/               nnr-nuvel-1a.dat
gpsplot.dat
$
```

This example introduces the switch, or flag, "`-F`", which modifies the output.

The output now identifies if the file is a

"regular file" (nothing appended), a

"directory" (slash appended), or an

"executable file" (asterisk appended, = program, application).

More switches

list entries beginning with the character dot, '.', which are normally hidden or invisible, using the '–a' flag, and show the listing in long format using the –l flag (plus the –F).

```
$ls -alF<CR>
drwxr-xr-x+ 92 rsmalley  staff     3128 Aug 31 12:48 .
drwxr-xr-x  5 root       admin      170 May 25 14:14 ..
-rwx------  1 rsmalley rsmalley      1201 Jul 10 15:03 .cshrc*
drwx------  1 rsmalley rsmalley     16384 Aug  1 13:50 bin/
-rw-------  1 rsmalley rsmalley  186668405 Jul 31  2007 world.dat
```

In this case can combine flags as above (`-alF`) or put individually (`-a –l –F`).

```
-rwx------  1 rsmalley rsmalley       1201 Jul 10 15:03 .cshrc*
drwx------  1 rsmalley rsmalley      16384 Aug  1 13:50  bin/
-rw-------  1 rsmalley rsmalley  186668405 Jul 31  2007  world.dat
```

What is the extra information

First character, "d" for directory, "-" for regular file, plus about 10 other things for other types of files.

The next 9 characters show read/write/execute privileges for owner, group, and all (or world or other).

```
-rwx------  1 rsmalley rsmalley       1201 Jul 10 15:03 .cshrc*
drwx------  1 rsmalley rsmalley      16384 Aug  1 13:50 bin/
-rw-------  1 rsmalley rsmalley  186668405 Jul 31  2007 world.dat
```

If have read, write or execute privileges has "`r`", "`w`", or "`x`" respectively. If not, has a "`-`".

So the owner has read and write privileges on all the files or directories, and execute privileges on the executable file (indicated by the "`*`"), .cshrc, and the directory bin (although one cannot execute a directory – if a directory is not executable other users can't cd or see into it).

Group and world or other have no privileges.

```
-rwx------  1 rsmalley rsmalley       1201 Jul 10 15:03 .cshrc*
drwx------  1 rsmalley rsmalley      16384 Aug  1 13:50 bin/
-rw-------  1 rsmalley rsmalley  186668405 Jul 31  2007 world.dat
```

Privileges can also be specified or displayed in OCTAL (base 8) with each bit of the octal value representing the permission/privilege.

```
rwx=111=7
rw-=110=6
r--=100=4
--x=001=1
```

etc. for owner, group, world.

```
-700  1 rsmalley rsmalley       1201 Jul 10 15:03    .cshrc*
d700  1 rsmalley rsmalley      16384 Aug  1 13:50    bin/
-600  1 rsmalle yrsmalley  186668405 Jul 31  2007    world.dat
```

This is “much better” (on a teletype) as it uses fewer characters (and requires being “in the know” to understand).

```
-rwx------  1 rsmalle yrsmalley        1201   Jul 10 15:03   .cshrc*
drwx------  1 rsmalley rsmalley       16384   Aug  1 13:50   bin/
-rw-------  1 rsmalley rsmalley   186668405   Jul 31  2007   world.dat
```

Temporarily skipping the next 3 columns, we then have the file size in bytes, the date the file was last modified, and the file name.

Switches/flags and manual pages:

Most Unix commands have switches/flags that can be specified to modify the default behavior of the command.

How do we find what switches are available and what they do?

The developers of Unix (being so smart) thought of this and provided documentation through the manual command – "man". To read the man page for the list command.

```
160:> man ls
Reformatting page.  Please Wait... done

User Commands                                                ls(1)

NAME
ls - list contents of directory

SYNOPSIS
     /usr/bin/ls [-aAbcCdfFghilLmnopqrRstux1@] [file...]

     /usr/xpg4/bin/ls [-aAbcCdfFghilLmnopqrRstux1@] [file...]

DESCRIPTION
     For each file that is a directory, ls lists the contents  of
     the  directory.  For  each file that is an ordinary file, ls
     repeats its name and any other  information  requested.  The
```

This goes on for quite a while. Note the --More-- (9%) at the bottom – says we are 9% done (oh joy on a teletype!)

output is sorted alphabetically by default. When no argument is given, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but file arguments appear before directories and their contents.

There are three major listing formats. The default format for output directed to a terminal is multi-column with entries sorted down the columns. The -1 option allows single column output and -m enables stream output format. In order to determine output formats for the -C, -x, and -m options, ls uses an environment variable, COLUMNS, to determine the number of character positions available on one output line. If this variable is not set, the terminfo(4) database is used to determine the number of columns, based on the environment variable, TERM. If this information cannot be obtained, 80 columns are assumed.

The mode printed under the -l option consists of ten characters. The first character may be one of the following:

--More--(9%)

continuing

```
     d     The entry is a directory.

     D     The entry is a door.

     l     The entry is a symbolic link.

     b     The entry is a block special file.

     c     The entry is a character special file.

     p     The entry is a FIFO (or "named pipe") special file.

     s     The entry is an AF_UNIX address family socket.

     -     The entry is an ordinary file.

     The next 9 characters are interpreted as three sets of three
     bits  each. The first set refers to the owner's permissions;
     the next to permissions of others in the user-group  of  the

SunOS 5.9           Last change: 19 Nov 2001                1

User Commands                                           ls(1)

     file; and the last to all others. Within each set, the three
     characters indicate permission to read,  to  write,  and  to
     execute  the  file  as a program, respectively. For a direc-
     tory, ``execute'' permission is interpreted to mean  permis-
     sion to search the directory for a specified file. The char-
     acter after permissions is ACL indication. A  plus  sign  is
     displayed if there is an ACL associated with the file. Noth-
     ing is displayed if there are just permissions.

     ls -l (the long list) prints its output as follows  for  the
     POSIX locale:

--More--(16%)
```

This goes on for several pages.

Try the manual command on a number of commands
(including the man command with "`man man`").

Man pages are pretty opaque.

They follow a fixed format giving you the name of the command and the list of switches.

Most do not have examples
(like math books that don't use figures since figures can't truly represent the math).

Removing files and directories

`rm<CR>` : remove files or directories

A very straightforward and potentially dangerous command.

There is no trash can on a unix machine.

Once you hit the `<CR>` it is **GONE**.

Now for some more commands.

(from here on, will drop the <CR> at end).

Removing files

`rm`: remove files or directories

CERI accounts are set up so that `rm` is aliased to `rm -i` (more on aliases later), which means the computer will ask you if you really want to remove the file(s) one at a time

```
% which rm
rm:    aliased to /bin/rm -i
```

and another new command "`which`" that tells you if an executable exists and where it "lives".

Removing files

```
%rm f1
remove f1?
```

Valid answers.

`Yes, yes, Y, y` – to accept and erase.

`No, no, N, n` – to not erase.

`<CR>` - does not erase, default.

Removing files

`rm`: remove files or directories

```
%rm f1
remove f1? yes<CR>
%
```

and bye-bye file.

Removing files

Remember that Unix is lean and mean.

It is a multi-user system and once the disk space associated with your file is released, the system can write somebody else's file into it immediately.

There is NO RECOVERING removed files.

(*You have been told. Sufficient for Unix users.*)

Removing files

Without the `–i` option set – this is what we would get.

```
%rm f1
%
```

and bye-bye file.

So if you made a typo – tough.

Removing files

If the `-i` option was not set – you can get it by typing `-i` yourself (you can find this out on the man page)
(but sooner or later you will mess up on one if you reset it back to normal operation!).

```
%rm -i f1
remove f1? y
%
```

and bye-bye file.

So if you made a typo – tough.

Removing files

Say you are 100% sure and don't want to have to answer the question and the pesky system manager has set an alias to protect you from yourself (very non Unix philisophy). You can return to the original definition of rm using the "\".

```
%\rm f1
%
```

and bye-bye file without prompting. So if you made a typo – tough.

General Unix behavior.

The "\" before a command undoes an alias and gives you the default Unix version of the command.

Removing files

We will see more potential `rm` disasters when we get to wildcards.

(If you have sufficient privilages, it is possible to accidently erase the whole operating system!!!)

Making & removing directories

`mkdir`: make directory

```
% mkdir bin src Projects Classes<CR>
```

Makes 4 directories: bin, src, Projects, and Classes in the working directory.

Making & removing directories

`rmdir`: remove directory - only works with empty directories so is safe (very uncharacteristic of Unix).

```
% rmdir bin src Projects Classes
```

Removes the 4 directories bin, src, Projects, and Classes in the working directory -- IF they are EMPTY.

Making & removing files and directories

`rm –d`: to use rm to remove directories

`rm –r`: removes directories recursively (i.e. all subdirectories and files in them); implies –d

can be very dangerous… one typo could remove months of work (will probably also need the \)

```
% \rm -r  Classes
```

So if you made a typo – tough.

Making & removing files and directories

- `% rm -r  Classes`

With the CERI alias for `rm` to `rm –i`, this command will prompt you for each file!

Gets tedious – and makes you want to do

- `% \rm -r  Classes`

Which is VERY DANGEROUS (but I've told you, so I'm off the hook).

Manipulating files

`cat`: concatenate files, sends files or Standard-IN to Standard-OUT.

If you want the concatenated files in another file – you have to redirect the output from Standard-OUT to the file.

Manipulating files

`cat`: Since it dumps the entire file contents to the screen

– we can use it to "print out" or "type out" a file.

Manipulating files

Another Unix philosophy issue –
use of side effects.

We don't need another command to print or type the contents of a file to the screen as it is a side effect of the `cat` command and the Standard-OUT operation of commands.

So feature it!
There is no "print" command, it is un-needed (lean and mean, emphasis on ***mean***. The sooner you begin to think like this the sooner you will be able to use Unix.).

Manipulating files

`cat`: make one file out of file1, file2 and file3 and call it alltogether.

```
%cat file1 file2 file3 > alltogether
```

This command (does not need input redirection, exception to regular rule that input only comes from Standard IN – but it will also take input from Standard IN) takes files file1, file2, and file3 and puts them into file alltogether.

Manipulating files

OK, what does this do?

```
%cat > myfile
```

Manipulating files

OK, what does this do?

```
%cat > myfile
```

(Put on your Unix thinking cap)

Manipulating files

OK, what does this do?

```
%cat > myfile
```

(Put on your Unix thinking cap)

It takes Standard-IN (the keyboard) and puts it into the file myfile.

Looking at files

OK, what does this do?

```
%cat > myfile
```

How does one get it to stop?
i.e. how do you let it know you are done entering stuff?

Enter "^d" or "^z", where "^" is the control (ctrl) key and you hold it down and then press the d or z.

Notice the logic associated with the input, output, and use of the command.

This type of thinking, or (il)logic, permeates Unix.

When you cat a long file it flies by on the screen (and off the top).

On newer GUIs there are scroll bars and you can scroll up and down.

On the older interactive terminals the text disappeared off the top.

Not good.

Aside:

UUOC - "Useless Use of cat"

```
cat filename | command arg1 arg2 argn
```

But "better" to use redirection (common)

```
command arg1 arg2 argn < filename
```

Uncommon, but legal and works

```
<filename command arg1 arg2 argn
```

From the Jargon File (glossary of UNIX computer slang)

CAT

To spew an entire file to the screen or some other output sink without pause (syn. blast).

By extension, to dump large amounts of data at an unprepared target or with no intention of browsing it carefully.

Usage: considered silly. Rare outside Unix sites. See also dd, BLT.

Among Unix fans, cat(1) is considered an excellent example of user-interface design, because it delivers the file contents without such verbosity as spacing or headers between the files, and because it does not require the files to consist of lines of text, but works with any sort of data.

Among Unix critics, cat(1) is considered the canonical example of bad user-interface design, because of its woefully unobvious name.

It is far more often used to blast a single file to standard output than to concatenate two or more files. The name cat for the former operation is just as unintuitive as, say, LISP's cdr.

UUOC - "Useless Use of cat"

"The purpose of cat is to concatenate (or catenate) files.

If it is only one file, concatenating it with nothing at all is a waste of time, and costs you a process."

This is also referred to as "Cat abuse."

This problem was fixed by another Unix program that takes Standard IN and puts it to Standard OUT a screenful at a time. (has to know about screens=hardware, a sub-optimal situation).

(This way, following the Unix philosophy, the `cat` program could be lean and mean. It did not have to figure out if it was going to the screen, how big it was, etc., it just sends stuff to Standard OUT.)

So we pipe the output into another program that handles the screen display.

This program is called more.

```
%cat myfile | more
```

The program **more** puts up a screens worth of text and then waits for you to tell it to continue (using the **space bar** for a new page worth and **<CR>** for a new lines worth of the file. **^z** to quit **more**.)

So we pipe the output into another program that handles the screen display.

This program is called more.

```
%cat myfile | more
```

The program `more` has to know about the screen/hardware. But the unix command `cat` is standard.

Looking at files

`more` can also be used directly

```
% more myfile
```

Or

```
% more < myfile
```

(more was written outside the Unix club and borrowed by Unix, so it does not strictly follow Unix philosophy.)

Looking at files

`less`: same as `more` but allows forward and backward paging.

(in OSX, `more` is aliased to `less` because `less` is `more` with additional features.)

(We will discuss aliases later.)

UNIX is a four letter word

"Unix is user friendly –

It's just picky about who it's friends are..."

-- Unknown, seen in .sigs around the world

Manipulating files

`paste`:
concatenate files with each file a new column; (when used on a single file, it dumps the entire file contents to the screen).

(`cat` sticks the files together one after the other – sequentially – to Standard-OUT.

`paste` puts them together a line at a time to Standard-OUT.
Each line N of the output file from paste is made up of the lines N of the M input files.)

Looking at files

`head –nX`

`head –X`

prints the first X number of lines to the screen;
default is 10 lines if -n is not specified.

`tail –nX`

`tail –X`

prints the last X number of lines to the screen;
default is 10 lines if -n is not specified.

(don't need the n)

Piping and Redirect

Input and output on the command line are controlled by the `|`, `>`, `<`, and `!` Symbols.

`|` : pipe function; sends the output from command on left side as input to the command on the right side.

(We have seen these actions already.)

Piping and Redirect

Example pipe

```
% ls | head -5
29-sadvf1
29-sadvf2
2meas.sh.out.txt
3132.dat
31all32new.trk
%
```

Piping and Redirect
">" redirects standard output (screen) to a specific file*

```
% ls | head -5 > directory.list
% more directory.list
29-sadvf1
29-sadvf2
2meas.sh.out.txt
3132.dat
31all32new.trk
```

- In tcsh, this will not overwrite (clobber) a pre-existing file with the same name – it will warn you.
- In the bash shell, the > overwrites (clobbers) any pre-existing file with no warning!

Piping and Redirect

>! (csh) or >| (bash): redirects standard output (screen output) to a specific file and overwrites (clobber) the file if it already exists *

```
% ls | head –n5 >| directory.list
% more directory.list
29-sadvf1
29-sadvf2
2meas.sh.out.txt
3132.dat
31all32new.trk
```

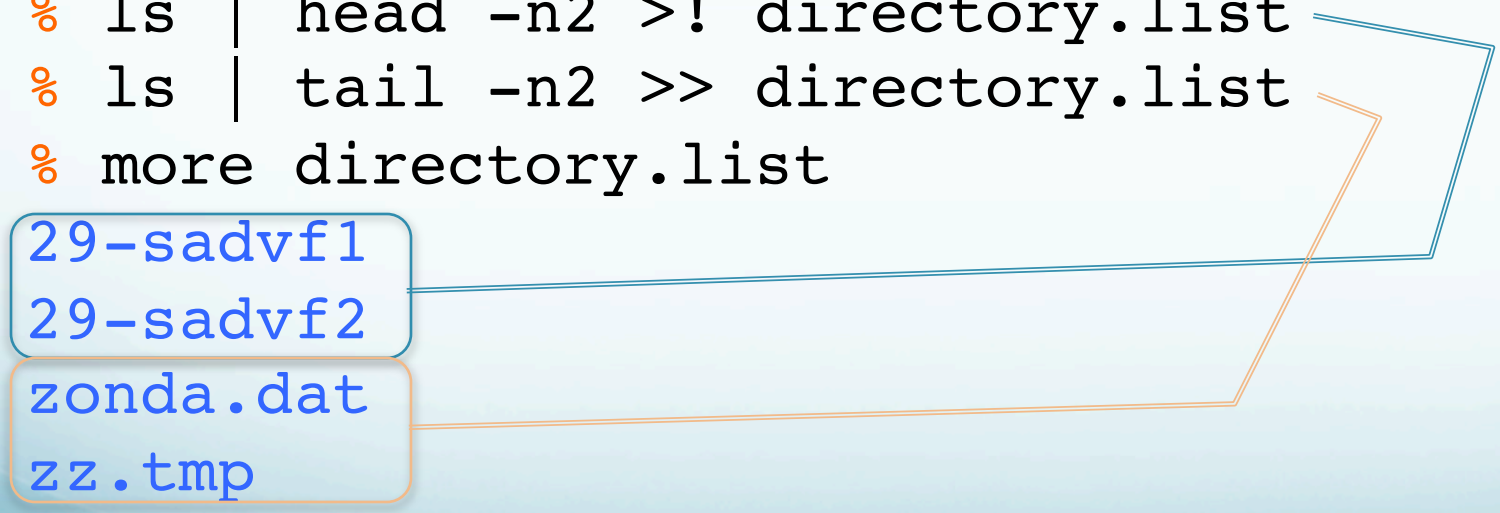
*make sure you use the correct one – else you will get files named "!" or "|"!

Use the command "set –o noclobber" in bash to turn noclobber on. To turn noclobber off use "set +o noclobber".

Piping and Redirect

>> : redirects and concatenates standard output (screen output) to the end of a specific (existing) file

```
% ls | head -n2 >! directory.list
% ls | tail -n2 >> directory.list
% more directory.list
29-sadvf1
29-sadvf2
zonda.dat
zz.tmp
```



Piping and Redirect

< : redirects input from Standard input to the file on right of the less-than sign to be used as input to command on the left

```
% head –n1 < suma1.hrdpicks
51995    31410273254     30870    958490
```

Copying files & directories

`cp`:

copy files

`cp -r`:

copy directory and all files & subdirectories within it (recursive copy)

Copying files & directories

```
% cp file1 ESCI7205/homework/HW1
```

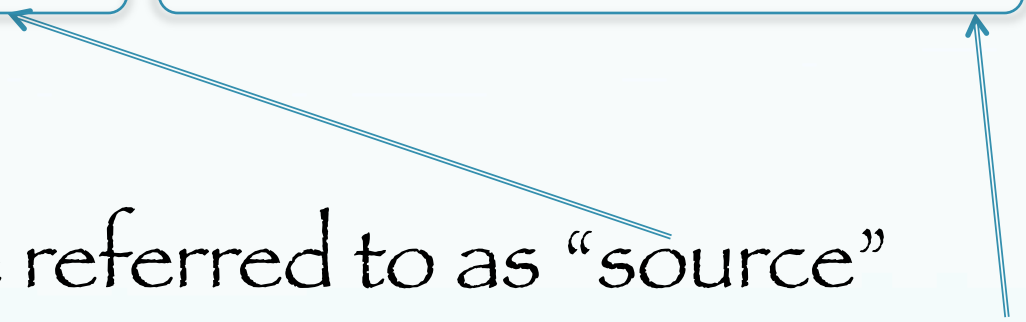
Makes a copy with a new name – "HW1" in the directory "ESCI7205/homework"

```
% cp file1 ESCI7205/homework/.
```

Makes a copy with the same name (file1), which is specified by the dot "." (period) to save typing, in the new directory.

Some jargon

```
% cp file1 ESCI7205/homework/.
```



Input file referred to as "source"

Output file referred to as "destination" or "target"

Moving files & directories
mv: move files or directories

- `% mv file1 file2 ESCI7205/HW/.`

Moves file1 and file2 to new directory (relative) ESCI7205/HW with same names (indicated by the ".").

Move differs from copy in that it removes the original file, you only have 1 copy of it when done.

Moving files & directories

`mv`: move files or directories

```
% mv file1 ESCI7205/HW/HW1
% mv file2 ESCI7205/HW/HW2
```

If you want to change the names when you move them, you have to specify each new file name (do them one at a time)

Renaming files & directories

(you should have been able to figure this out after the last two slides)

Uses a side-effect of move!!!

- `% mv file1 HW1`
- `% mv file2 HW2`

There is NO RENAME command.
(We consistently see this kind of inconsistent [il]logic in Unix.)

Linking files & directories

`ln -s`:

creates a symbolic link between two files.

This makes the file show up somewhere (the target, can be a new name in the same directory or the same name in another directory), but the file really only exists in the original place.

(equivalent to a file alias in OSX or shortcut in Windows).

Try reading the man page –

```
LN(1)                BSD General Commands Manual                LN(1)

NAME
     link, ln -- make links

SYNOPSIS
     ln [-Ffhinsv] source_file [target_file]
     ln [-Ffhinsv] source_file ... target_dir
     link source_file target_file

DESCRIPTION
     The ln utility creates a new directory entry (linked file)
which has the same modes as the original file.  It is useful for
maintaining multiple copies of a file in many places at once
without using up storage for the ``copies''; instead, a link
``points'' to the original copy.  There are two types of links;
hard links and symbolic links.  How a link ``points'' to a file
is one of the differences between a hard and symbolic link.
```

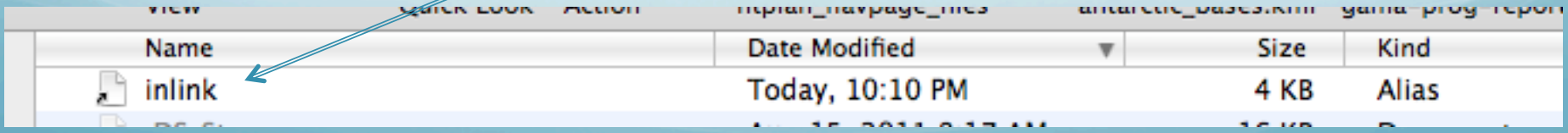
Linking files & directories

Two kinds of link - symbolic and hard. Only root can make hard links so don't worry about them.

```
% ln -s in inlink
```

"real"/actual file

linked file



Linking files & directories

Doing an ls command in the directory with the alias produces the following

```
$ ls -l in*
-rw-r--r--@ 1 smalley staff 69 Apr 26 2010 in
lrwxr-xr-x  1 smalley staff  2 Sep  2 22:10 inlink -> in
```

The leading "l" (the letter l, not the nubmer 1) in the long ls output says the file/filename in that line is a link.

It shows which file it is linked to.

Linking files & directories

This allows us to "have" the file in more than one place.

We can therefore access it locally from the directory where it is a symbolic link.

Introduction to wildcards.

Wildcards are essential when dealing with almost anything in terms of text processing. (Looking for/Managing files from the command line is text processing.)

They are a subset of regular expressions, an essential (i.e. powerful ≈ esoteric and difficult) Unix feature.

Wildcards

Wildcards allow you to match multiple instances of characters/numbers in file or directory names

They can be used in combination with almost all Unix commands

Wildcards are essential when dealing with large amounts of geophysical data

Introduction to wildcards.

Example

Say I want to find all the files in the working directory that begin with the letter "a".

(lower case only since Unix is case sensitive.)

Start out with the `ls` command

How do we specify we want all combinations of all characters following the "a"?
We use a wildcard.

```
% ls a*
```

The asterisk "*" wildcard means match a string with *any number of any character* (including none, so will match a file "a").

Try it ~~~

```
> ls a*
a.out                                antex.sh
antarctic sun panorama 3x.ai         atantest.f
antarctic sun panorama.125.jpg       awk
antarctic sun panorama.25.jpg        az_map
antarctic sun panorama.ai            az_map.ps
antarctic sun panorama.jpg

adelitst:
aadeli.ini        adelitst.sh     jessai          pessai
ADELI.MESSAGES    eessai          kcnusc.pal      PLOT1
ADELI.MINMAX      iessai          oessai          tempi

arc2gmtstuff:
arcgmt.README   arcgmt.tar      arcgmt_ai       arcgmt_av
>
```

Probably not what you wanted though – it lists files starting with "a" and then goes recursively through all directories that start w/ "a".

Try it ---

```
> ls –d a*
a.out                                   antex.sh
antarctic sun panorama 3x.ai            atantest.f
antarctic sun panorama.125.jpg          awk
antarctic sun panorama.25.jpg           az_map
antarctic sun panorama.ai               az_map.ps
antarctic sun panorama.jpg
>
```

Flag –d says do not go recursively through all directories (that start w/ "a").

Use man page to figure this out.

(As part of the regular expression feature of Unix) wildcards can be used in combination with almost all Unix commands.

Wildcards

"*" – asterisk - matches zero or more characters or numbers.

Combining/multiple use of wildcards.

Find all files in local subdirectory SEIS that begin with the letter "f" and also have the string ".BHZ." in their file name.

```
%ls SEIS/f*.BHZ.*
SEIS/filt.HIA.BHZ.SAC  SEIS/filt.WMQ.BHZ.SAC
```

"?" – question mark - matches a single character or number.

Find all files in local subdirectory SEIS that have the name "HIA.BH" plus some single letter (the ?) plus a "." and then plus anything (the *).

```
% ls  SEIS/HIA.BH?.*
SEIS/HIA.BHE.SAC        SEIS/HIA.BHN.SAC
SEIS/HIA.BHZ.SAC
```

Wildcards

"[ ]" – brackets - used to specify a set or range of characters or numbers rather than all possible characters or numbers.

Find all files in local subdirectory SEIS that have the name "**HIA.BH**" plus one of E, N or Z (the stuff in brackets) plus a "**.**" and then plus anything (the *).

```
% ls  SEIS/HIA.BH[E,N,Z].*
SEIS/HIA.BHE.SAC        SEIS/HIA.BHZ.SAC
SEIS/HIA.BHN.SAC
```

Wildcards

"[ ]" – brackets - used to specify a set or range of characters or numbers rather than all possible characters or numbers.

Find all files in local subdirectory SEIS that have the name "**HIA.BH**" plus one of E, N or Z (the stuff in brackets) plus a "**.**" and then plus anything (the *).

```
% ls  SEIS/HIA.BH[ENZ].*
SEIS/HIA.BHE.SAC        SEIS/HIA.BHZ.SAC
SEIS/HIA.BHN.SAC
```

Wildcards

Find all files in all local subdirectories (the first *) that have the string "HIA" in the name plus anything (the second *) plus the characters "198" plus a single character in the range 0-9 then plus anything (the third and last *).

```
% ls  */HIA*198[0-9]*
795/HIA.BHZ.D.1988.041:07.18.30
799/HIA.BHZ.D.1988:14:35:27.00
812/HIA.BHZ.D.1988:03:43:49.00
813/HIA.BHZ.D.1988.362:13.58.59
814/HIA.BHZ.D.1989.041:17.07.43
```

Some random stuff
Control-characters (CTRL-characters)

`ctrl-s` freezes the screen and stops any display on the screen from continuing (equivalent to a no-scroll key) (sometimes takes a moment to work)

`ctrl-q` un-freezes the screen and lets screen display

Some random stuff
Control-characters (CTRL-characters)

continue `ctrl-c` interrupts a running program

`ctrl-\` same as `ctrl-c` but stronger (used when terminal doesn't respond)

Some random stuff
Control-characters (CTRL-characters)

`ctrl-z` suspends a running program (use the fg command to continue the program)

`ctrl-h` deletes last character typed

`ctrl-w` deletes last word typed

`ctrl-u` deletes last line typed

Some random stuff
Control-characters (CTRL-characters)

`ctrl-r` redraws last line typed

`ctrl-d` ends text input for many UNIX programs, including mail and write.

(http://web.cecs.pdx.edu/~rootd/catdoc/guide/TheGuide_38.html)

Some random stuff

Everything "looks like" a file to UNIX.

Some random stuff

Everything "looks like" a file to UNIX.

What does this mean?

What we have seen so far

Commands

```
cd
pwd
ls
mkdir
rmdir
rm
more
less
cat
paste
head
tail
cp
mv
ln
echo
man
```

See this link for a list and description of many Unix commands

http://pcsplace.com/tech-list/ultimate-list-of-linux-and-unix-commands/

What we have seen so far

Redirection

Pipes

Switches

Some special characters (~ \ . ..)

Wildcards (* ?)