Data Analysis in Geophysics
ESCI 7205

Class 2

Bob Smalley

Basics of the Computer Environment

User ID and passwords

Basics of the Unix/Linux Environment

User ID and Passwords

User ID: usually a derivative of your name and the same as the beginning of your email.

Your CERI and UoM user ids are the same.

(User ID's on the UM system can only be a maximum of 8 characters due to the limitations of early computes. Unix is full of such anachronisms.

User ID's are formed using an algorithm. Take first initial (and maybe middle initial) and full last name, if that is more than 8 characters, start removing vowels from the back, if still longer than 8 characters, start removing consonants from the back.)

User ID and Passwords

Password: a (hopefully complicated, hard to guess [and therefore to remember]) combination of upper and lower case characters, symbols, and numbers that allows access to the account.

Your CERI Unix passwords for the Sun and Mac systems only work on those systems (two separate passwords, but you can use the same one).

Your UoM Outlook email, Spectrum, Tiger labs, and CERI PC lab password is the same for all these systems (because they all access your primary UoM/Microsoft account).

User ID and Passwords

If you need/use the non-Sun systems (Unix machines in the GPS lab, a faculty member's MAC, etc.) you will have to see the owner of the machine to get an account and will have a unique username/password for each of them.

Passwords

Do not tell anyone your password!!!!

Do not leave your password sitting around on a post-it note.

Do not email your password.

If you forget your password, you have to visit the system administrator and (humbly) ask for a new one. There is no way for anyone (except hackers) to figure it out.

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.

Back to basics

"UNIX Philosophy"

"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"

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

```
Robert-Smalleys-MacBook-Pro:untitled folder robertsmalley$ ls<CR>
Robert-Smalleys-MacBook-Pro:untitled folder robertsmalley$
```

Returns to prompt without any other output. (there are no files to list, so Unix just outputs a <CR> [and a new prompt], is that reasonable?). (Works differently in shell script, no <CR>)

"UNIX Philosophy"

What happens when you enter

```
Robert-Smalleys-MacBook-Pro:documents robertsmalley$ 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>, after the <CR> waiting for input)

"UNIX Philosophy"

What happens when you enter

```
Robert-Smalleys-MacBook-Pro:documents robertsmalley$ 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

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 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 a good assumptions (Stnd IN, Stnd OUT) for many tasks and many Unix commands break this convention.

"UNIX Philosophy"

redirection when input and output are not coming from or going to standard places ("<", "<<" 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"

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 stationary one 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.)

Don't insist on interactive input.

(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.

Suppose I want to print something and would like to control its orientation – landscape or portrait.

```
ORIENT=<CR>
```

Or

```
ORIENT=-P<CR>
```

Then

```
print `echo $ORIENT` < INFile<CR>
```

Puts in nothing, in case of `ORIENT=<CR>`, or "-P", in case of `ORIENT=-P<CR>`, into the command as if that is what you typed

(this is not how you would actaully do this – it is a ginned up example)

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.

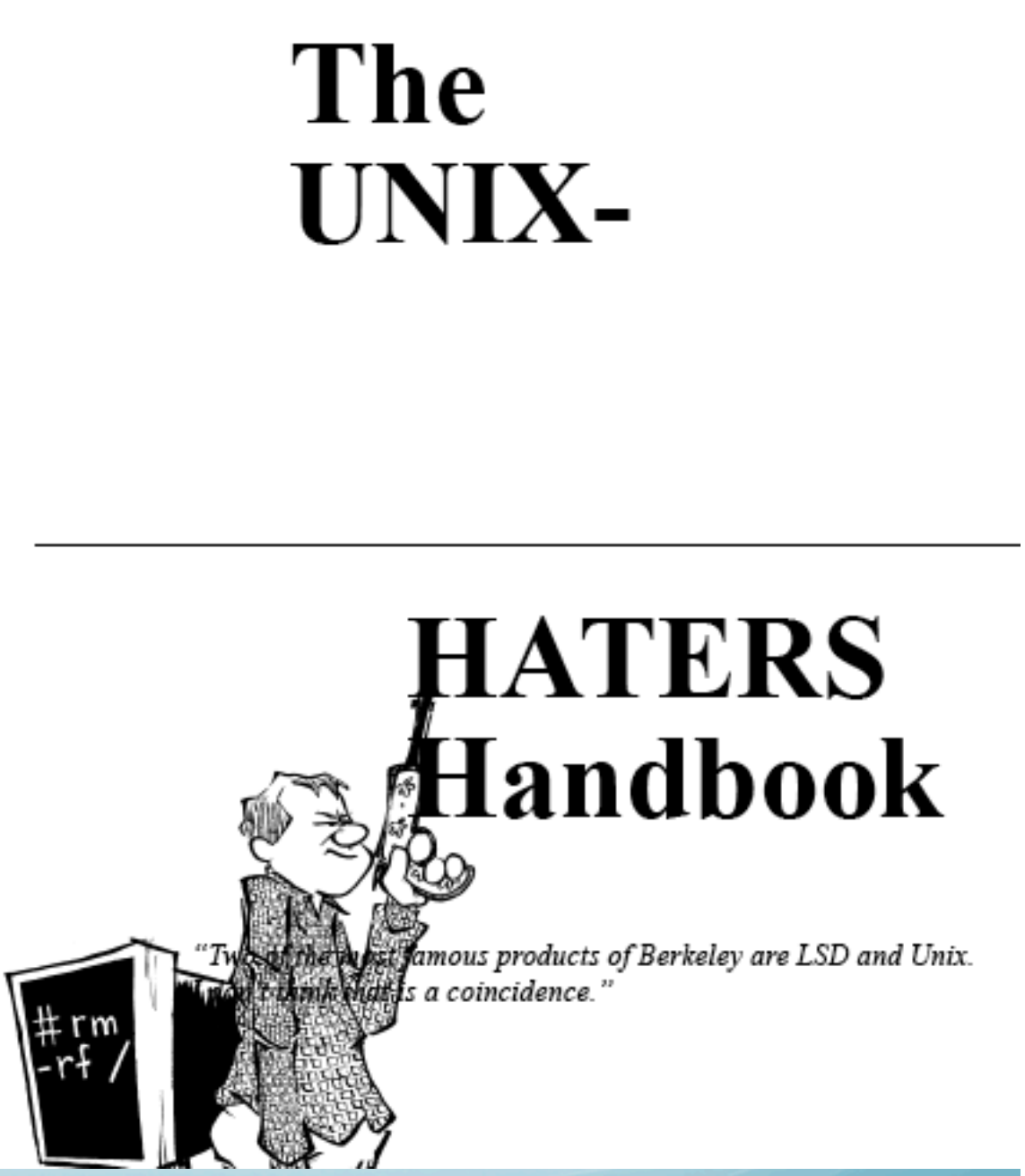
or

The trouble with UNIX: The user interface is horrid

Norman, D. A. *Datamation*, 27, No. 12, 139-150.

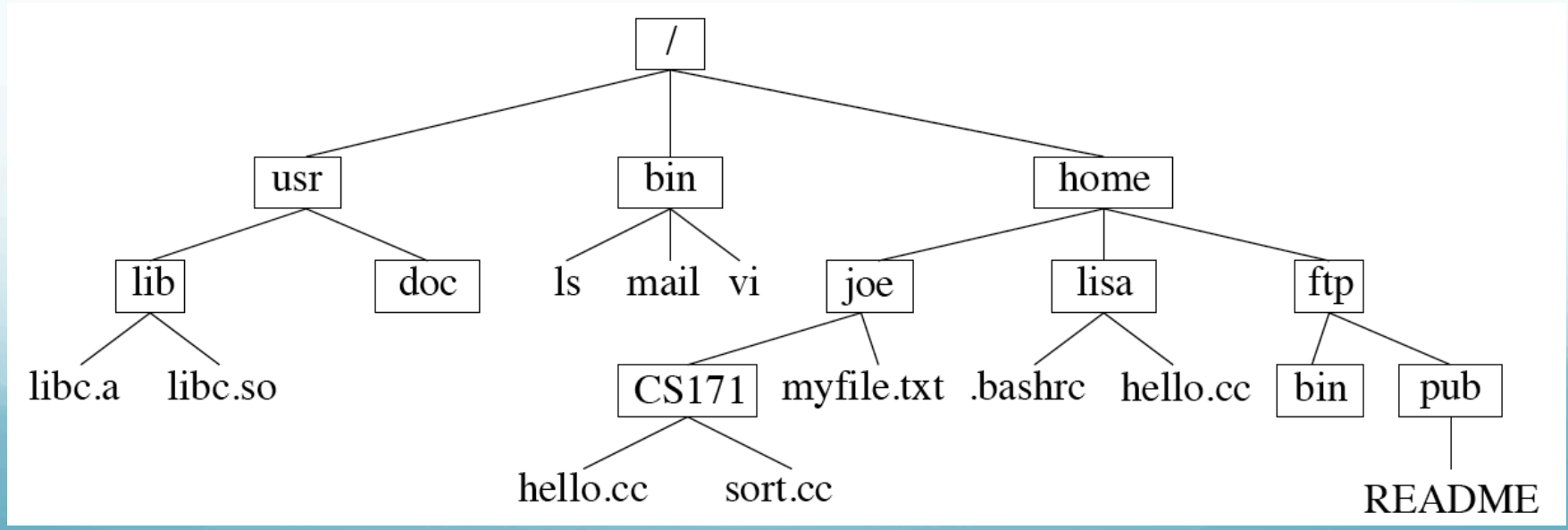
"Two of the most famous products of Berkeley are LSD and Unix. I don't think that this is a coincidence."

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).

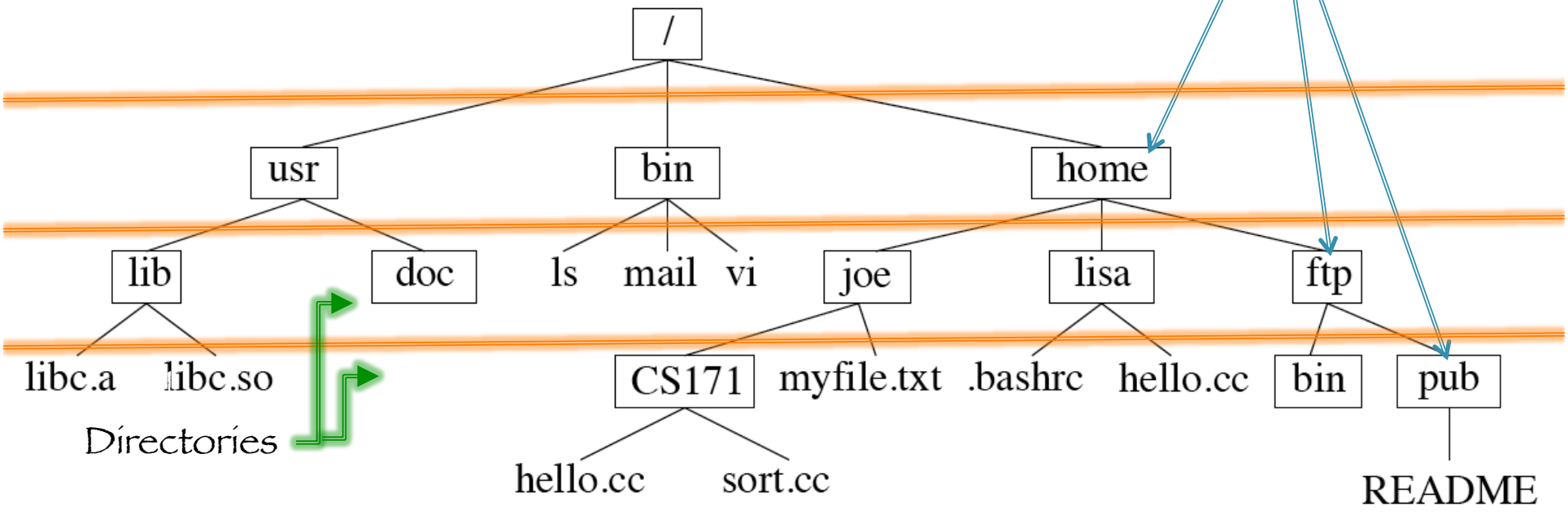
Unix uses a hierarchical file system (as does Mac and Windows/DOS).



Looks like an upside down tree.

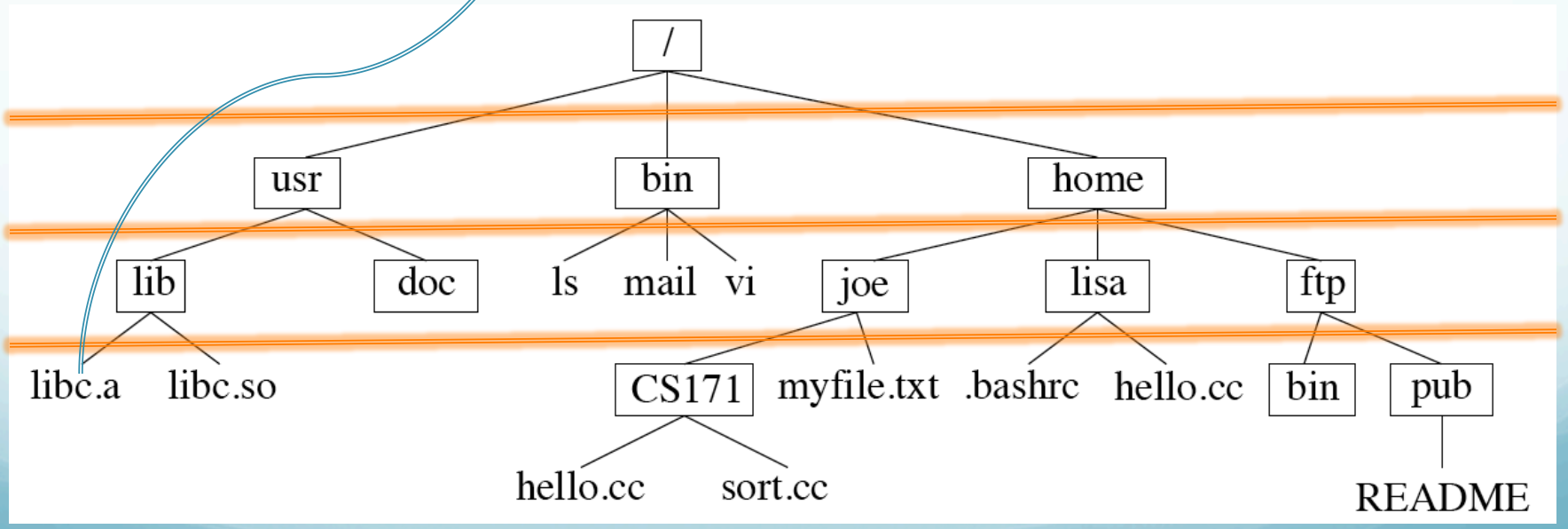
Starts at top with "/", called "root".

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

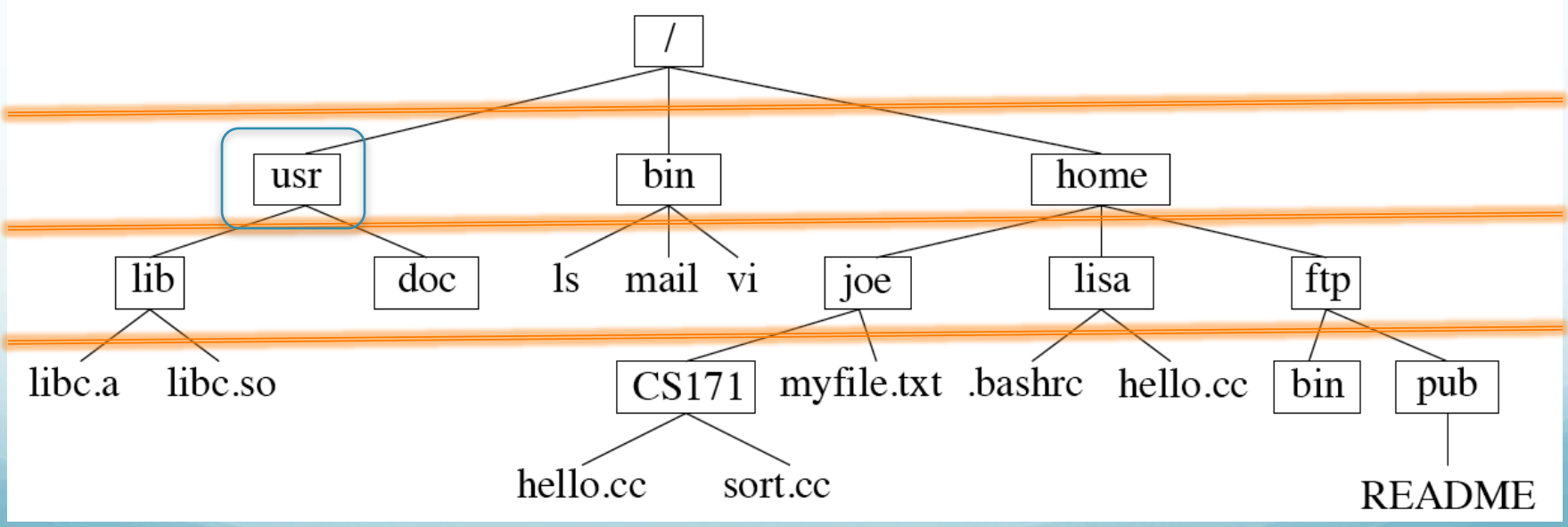
/usr/lib/libc.a



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`

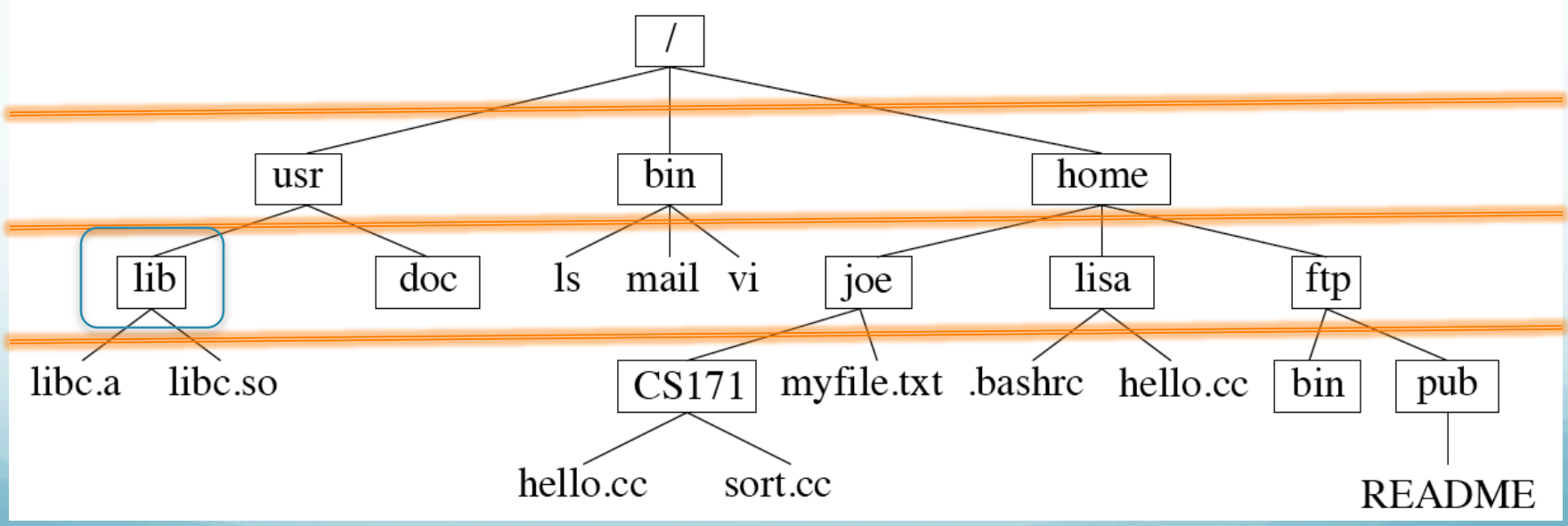
(no leading slash)



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

`libc.a`

(no leading slash)

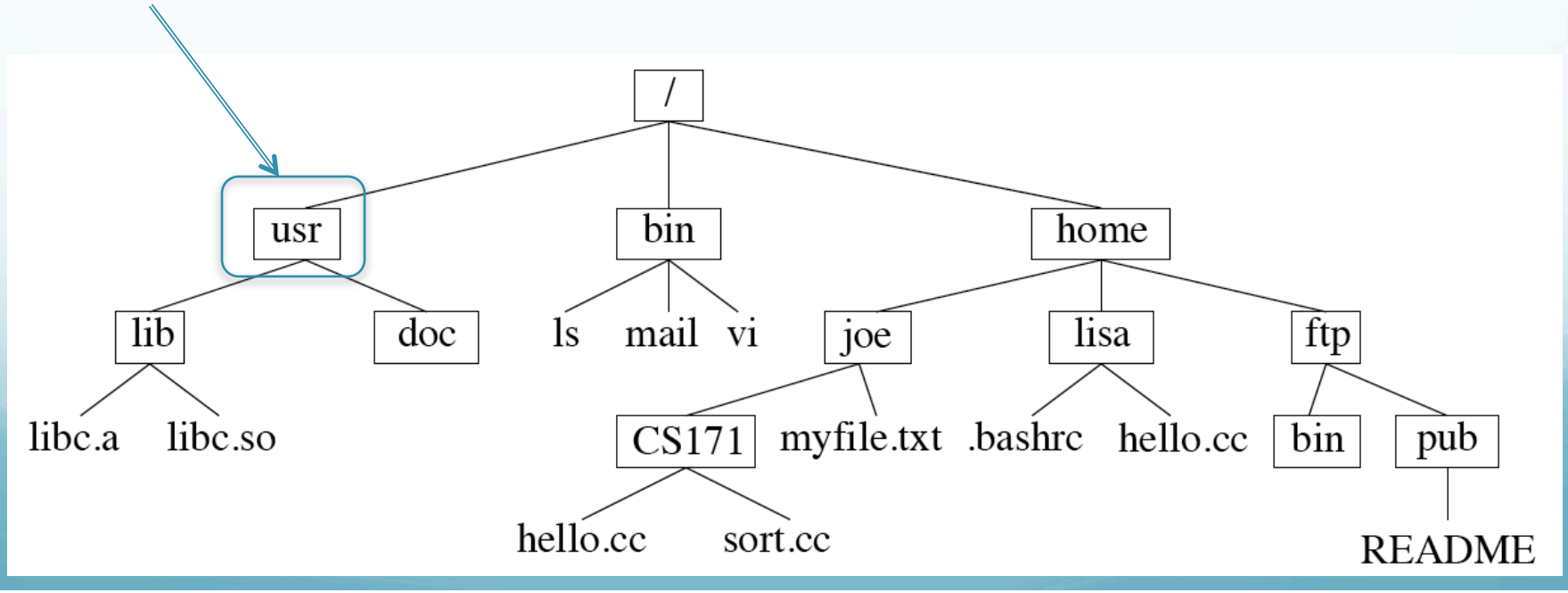


The "/" (slash or forward slash) in Unix is roughly equivalent to the "\" (backslash) in Windows/ DOS.

Some commands:

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

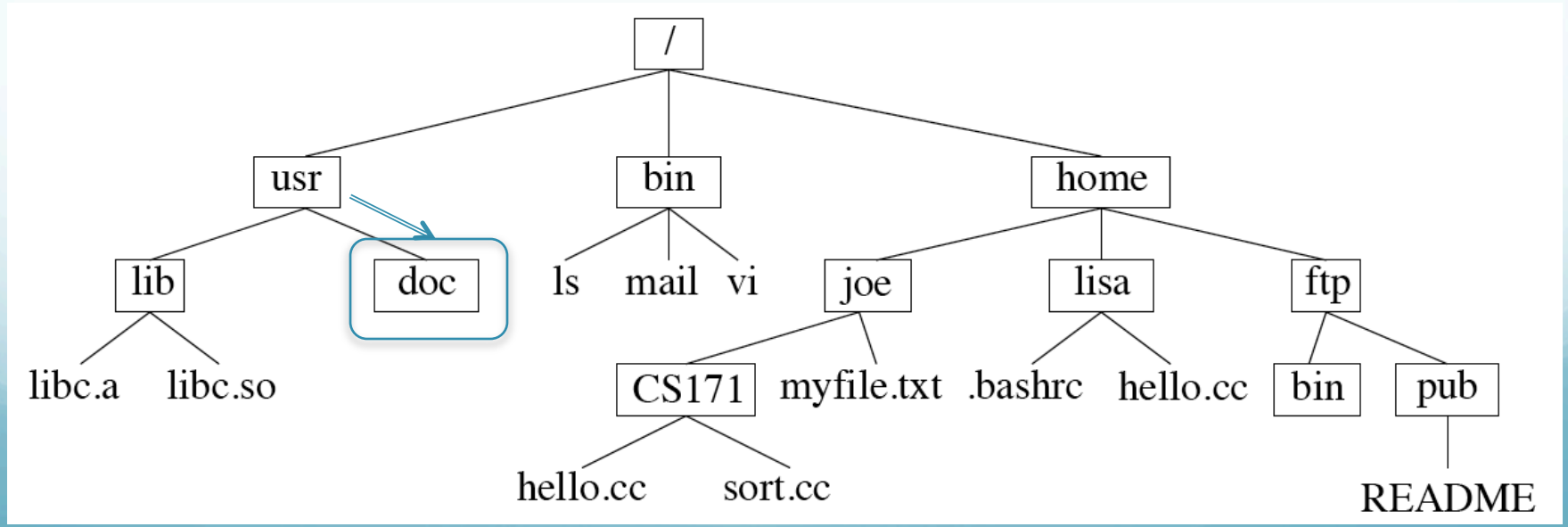
```
smalleys-imac-2:usr smalley$ pwd<CR>
/usr
smalleys-imac-2:usr  smalley$
```



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

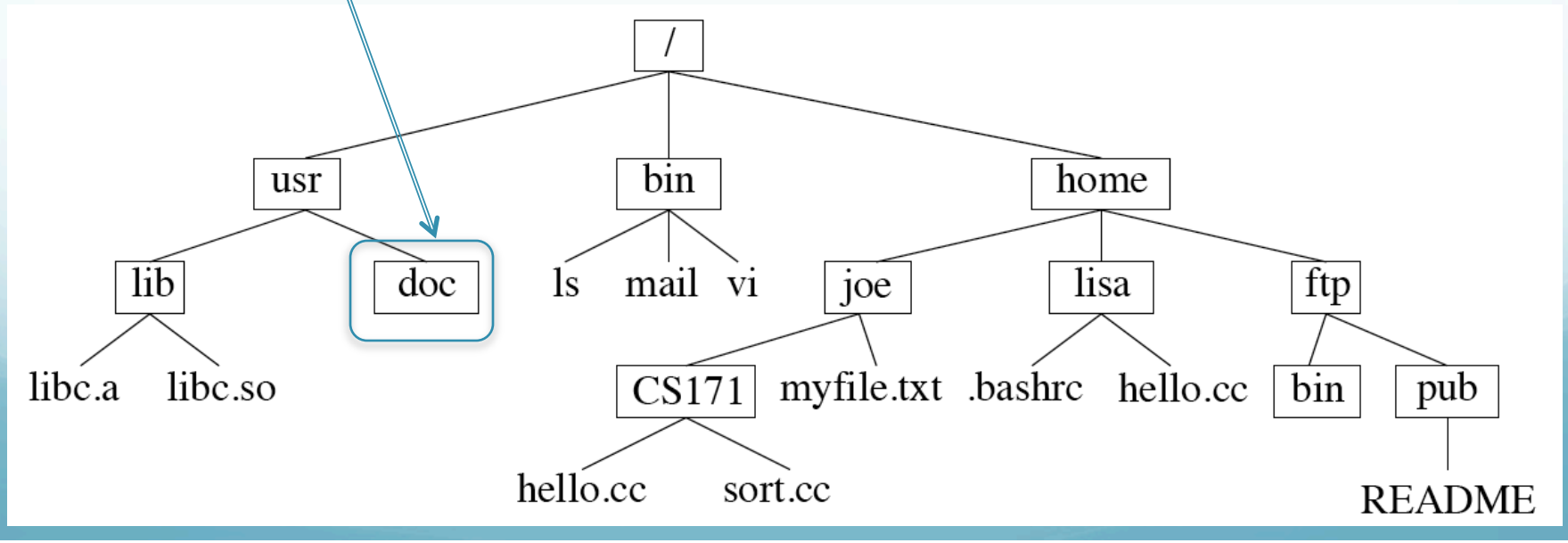
To go down to the directory doc we use the "change directory" = "**cd**" command

```
smalleys-imac-2:usr smalley$ cd doc<CR>
smalleys-imac-2:doc smalley$
```



Now

```
smalleys-imac-2:usr smalley$ pwd<CR>
/usr/doc
smalleys-imac-2:doc  smalley$
```



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.

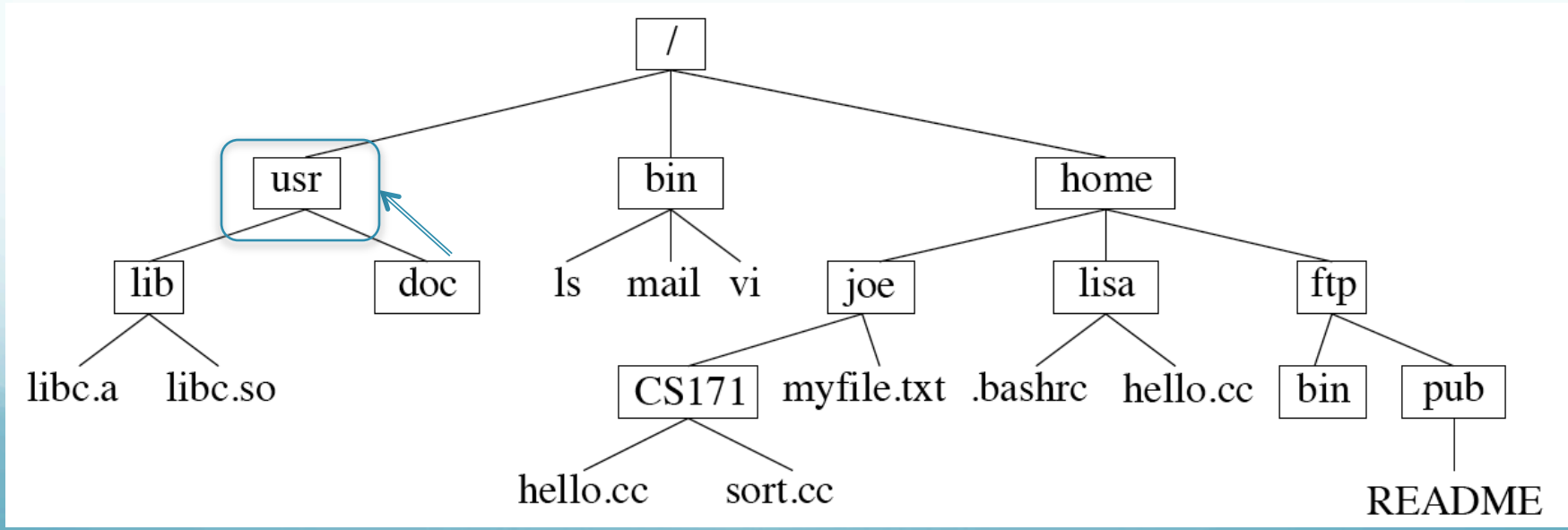
e.g. "cd" for "change directory"

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

We can also go up the directory structure.

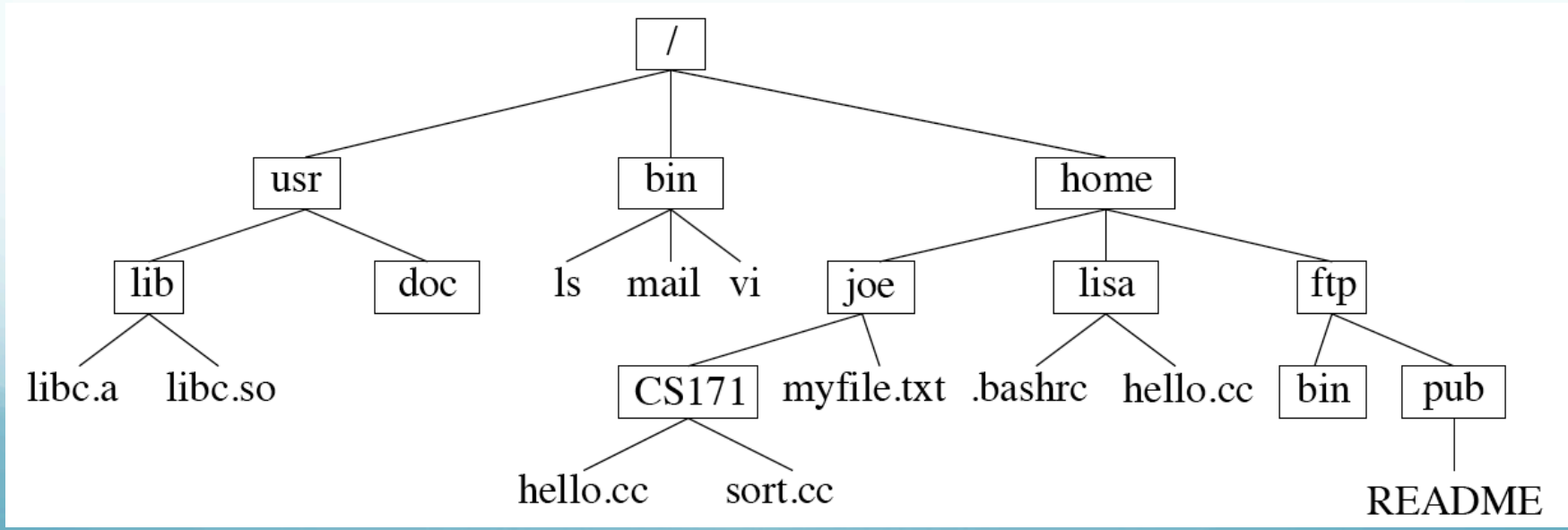
To return to usr from doc.

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



This is a little strange ---

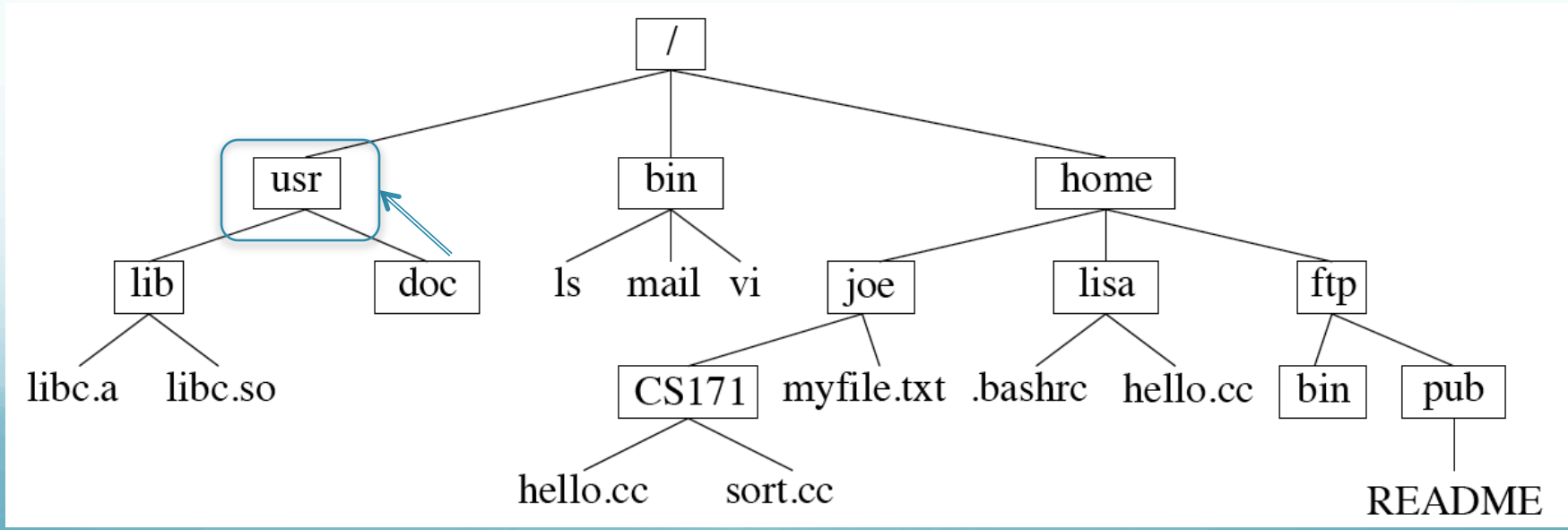
The double dot ("..") signifies the directory directly above you (up) in the directory structure (tree).



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

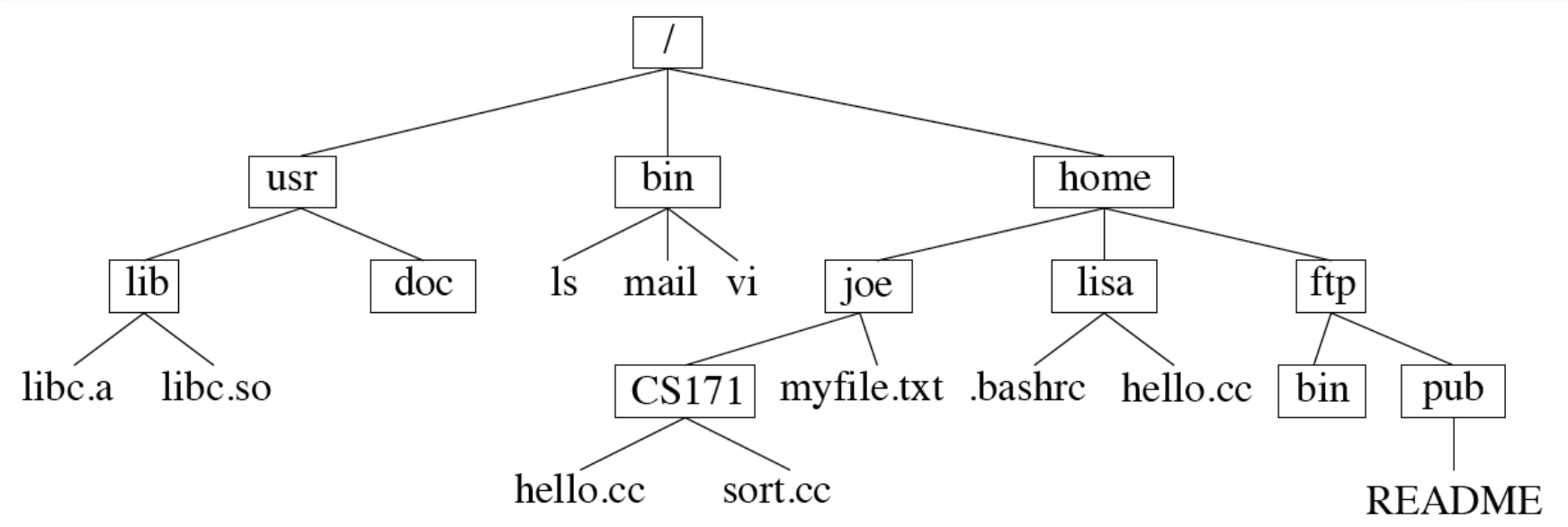
To go to usr (from doc or anywhere, such as pub)

```
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.

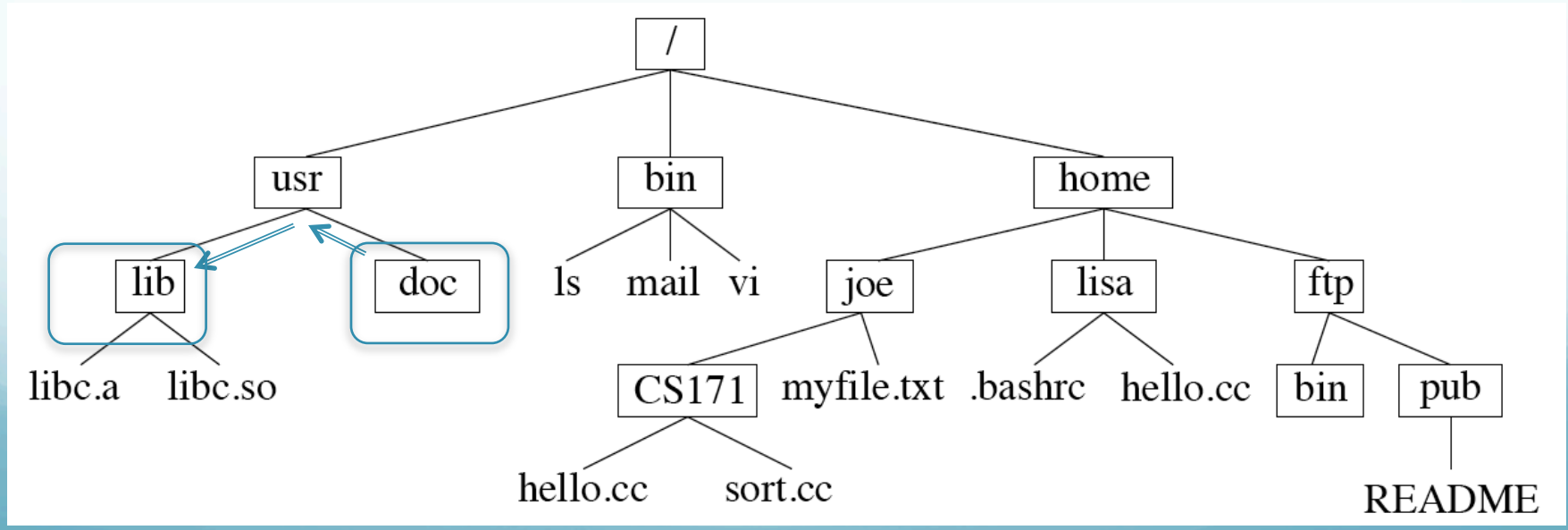
Unix does not provide a display of the picture below. You need to have it in your head.



How do we go from doc to lib?

We could do this using the full path.

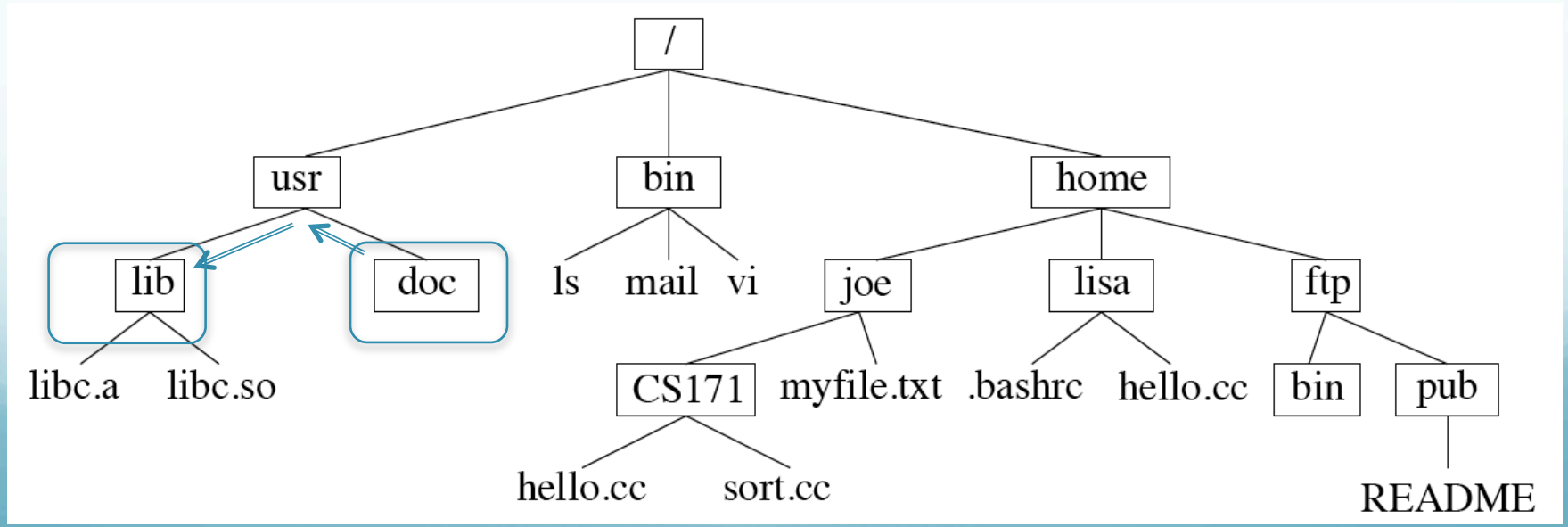
```
doc$ cd /usr/lib<CR>
lib$
```



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.

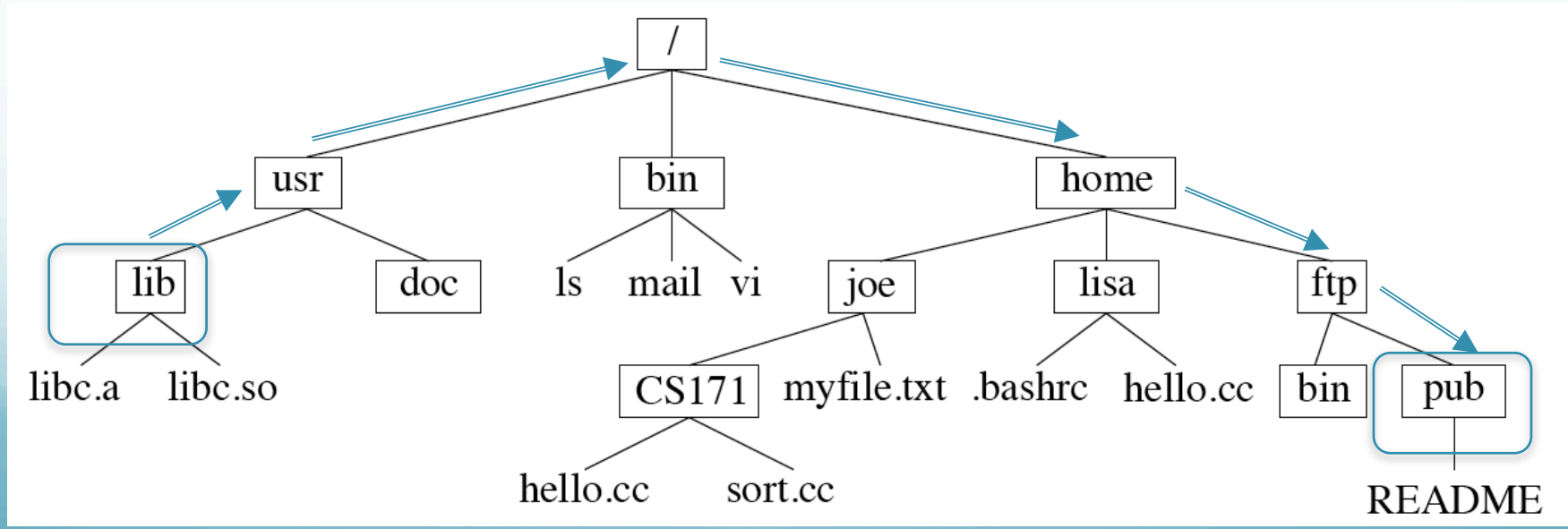
```
doc$ cd ../lib<CR>
lib$
```



Say we want to go to "pub"

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

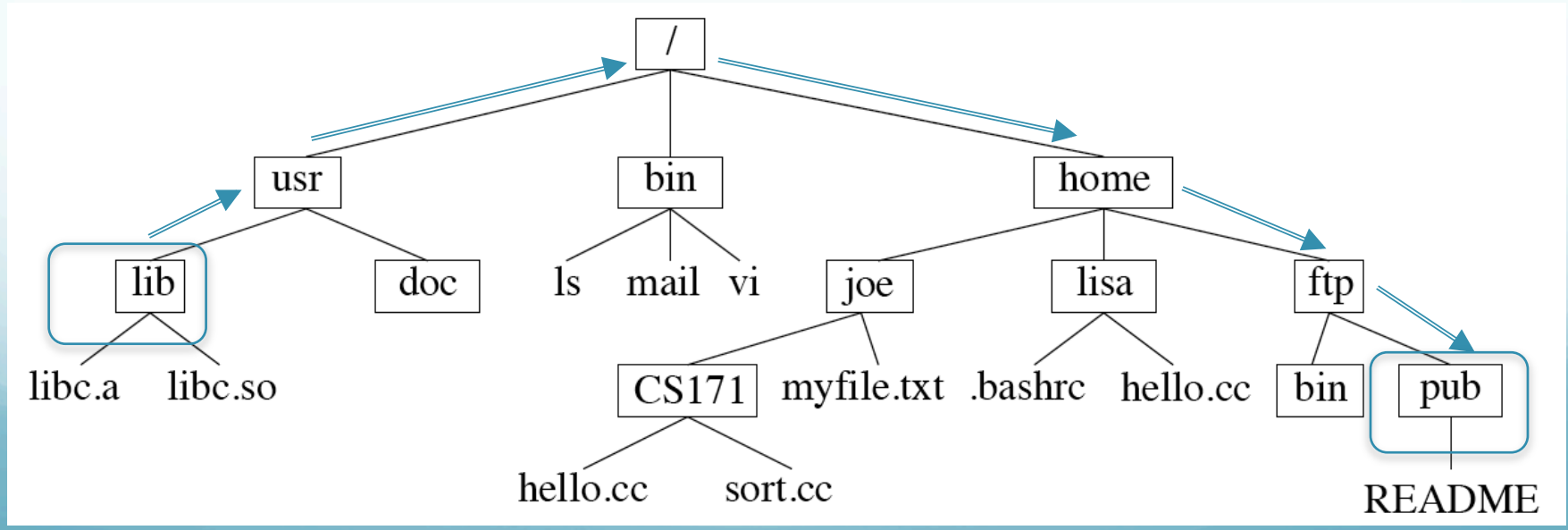
We went up two, then down three.



Say we want to go to "pub"

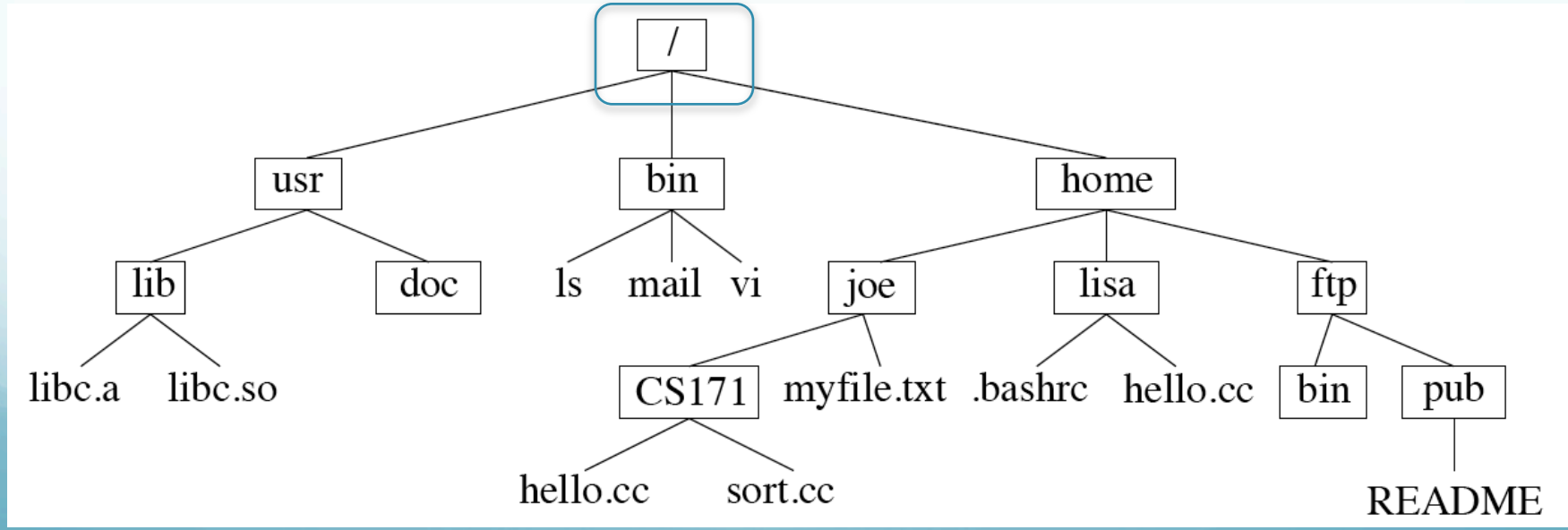
We could also have done (and is simpler) this with the full path.

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



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

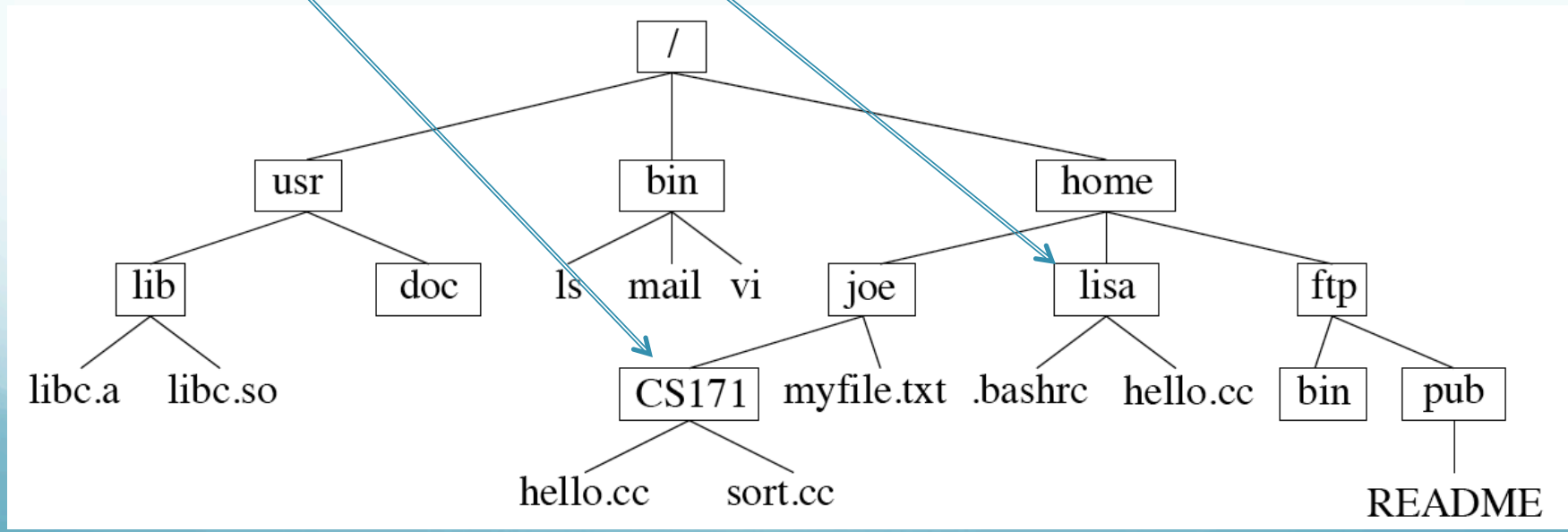
```
lib$ cd /<CR>
/$
```



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

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

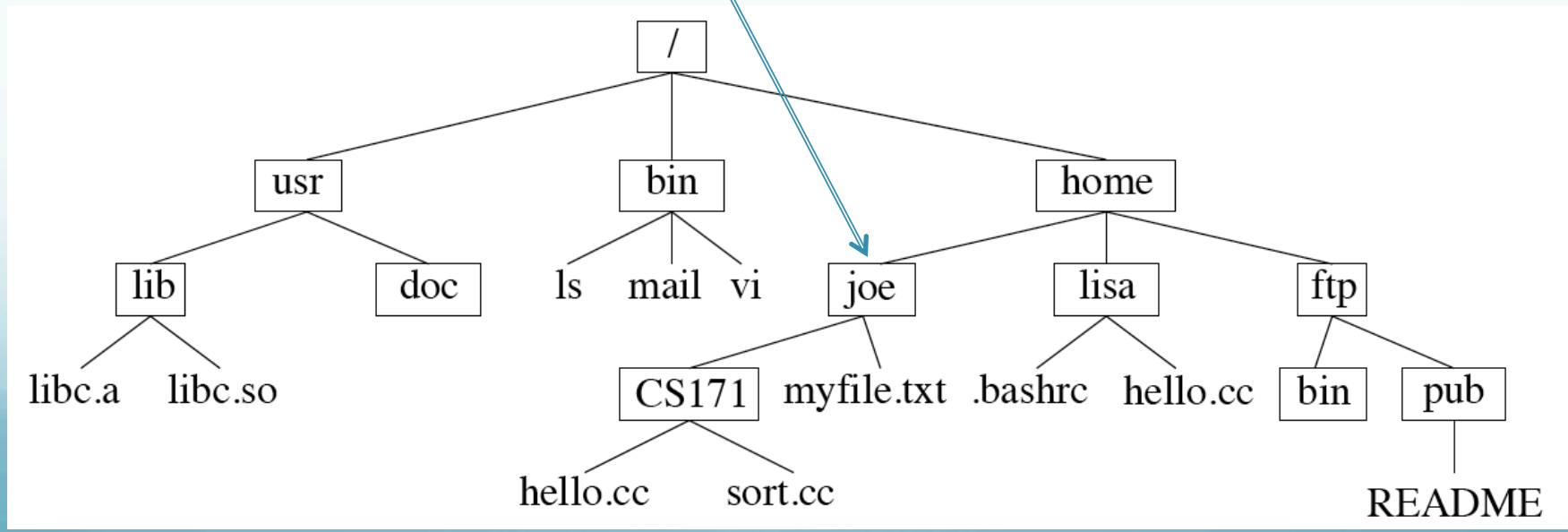
uses tilde "~"



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`")

Review - specifying file names

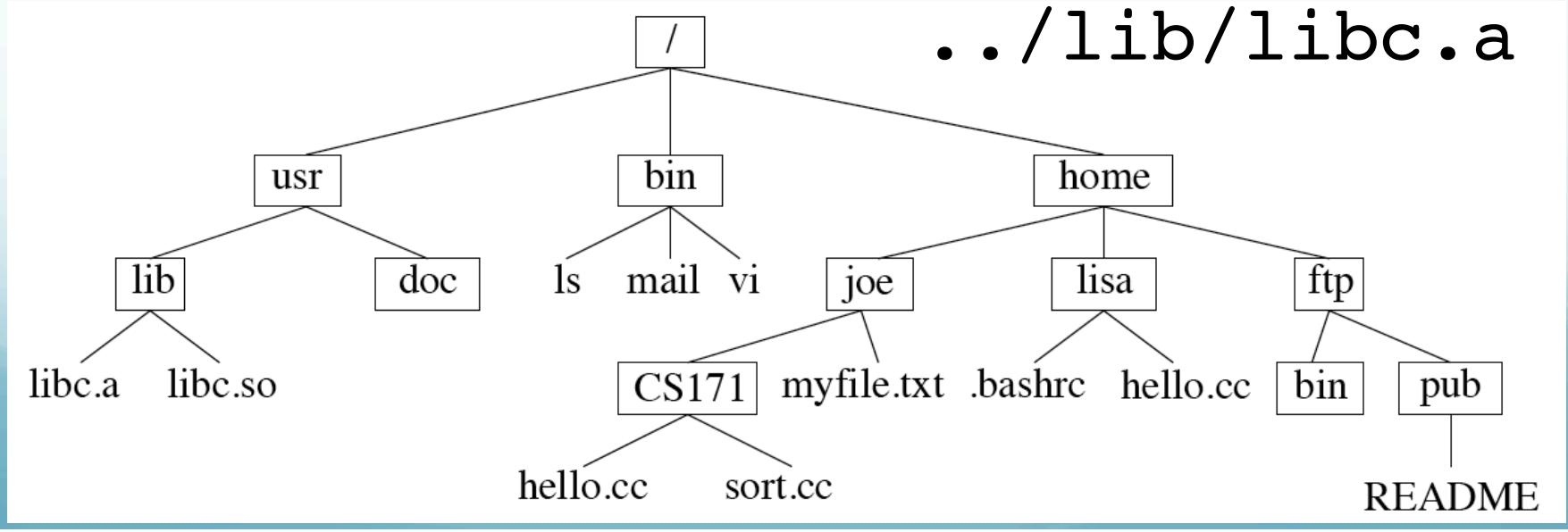
full path

`/usr/lib/libc.a`

relative path

(if in directory lib) `libc.a`

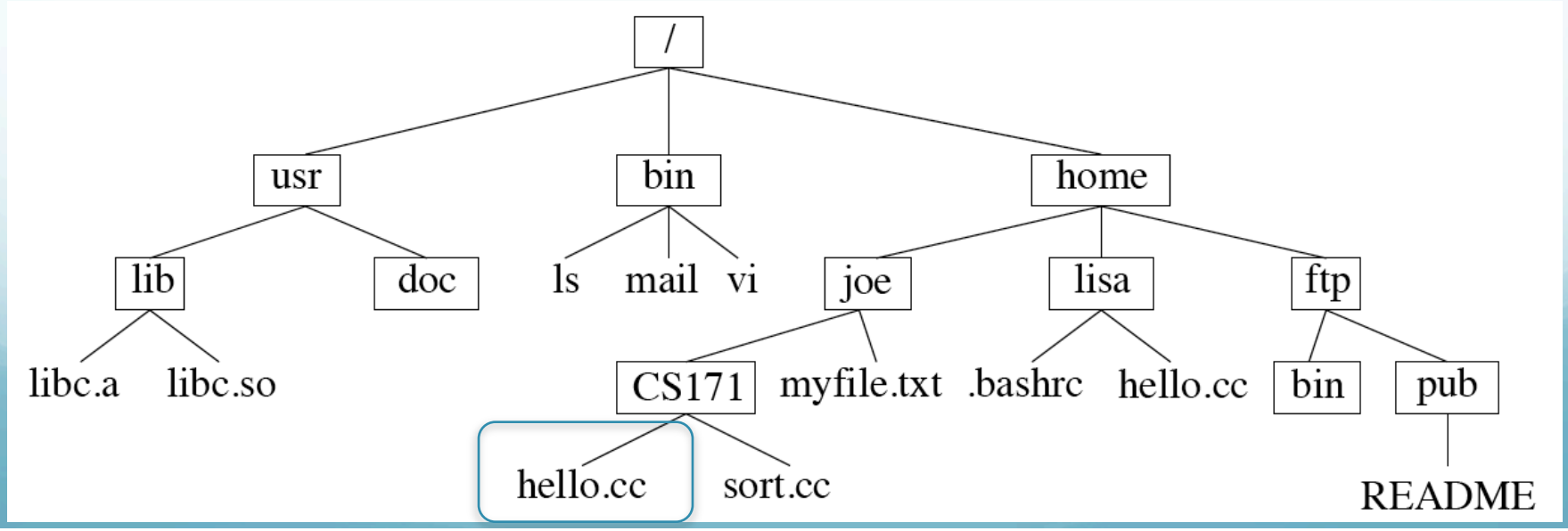
(if in another directory next to it, e.g. doc `../lib/libc.a`



Review - specifying file names abbreviations

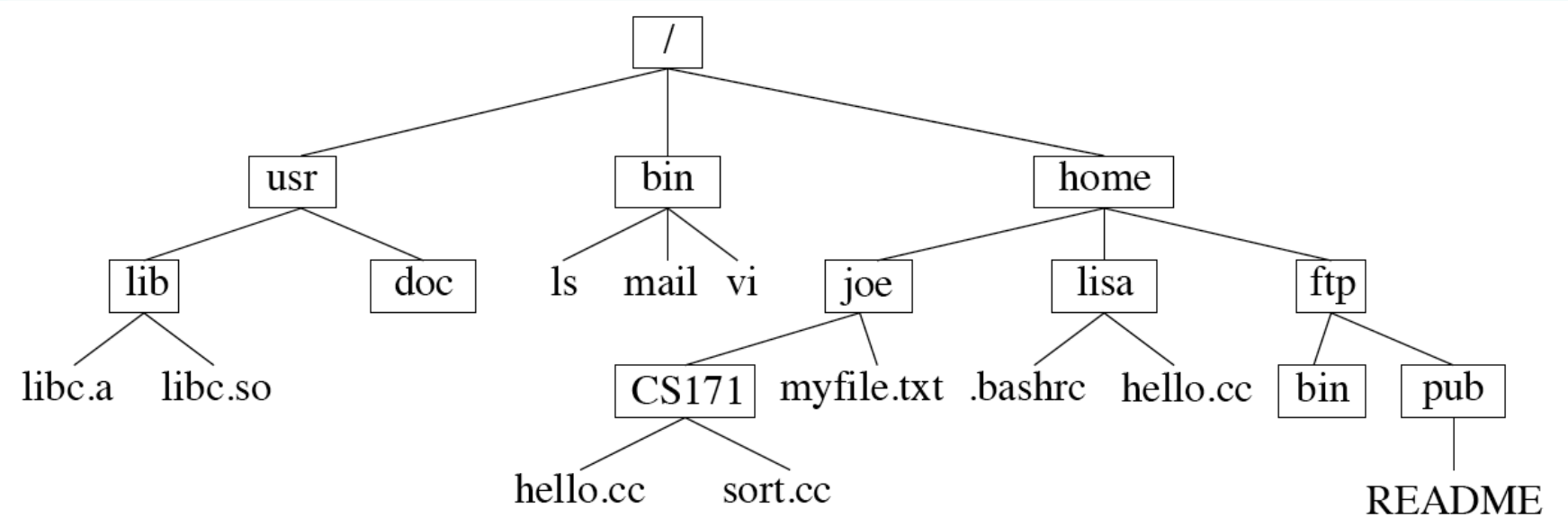
(if I am joe) `~/CS171/hello.cc`

(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.

```
smalleys-imac-2:~ smalley$ ls<CR>
Adobe SVG 3.0 Installer    vel.dat
Desktop                    heflen_web.dat
Documents                  isc0463.dat
Downloads                  nuvel-1a.dat
gpsplot.dat
smalleys-imac-2:~ smalley$
```

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 that name.

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 your 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 '*'

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

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).

```
smalleys-imac-2:~ smalley$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.

```
alpaca.ceri.memphis.edu/rsmalley 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)

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.

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).

(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, 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**.)

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.)