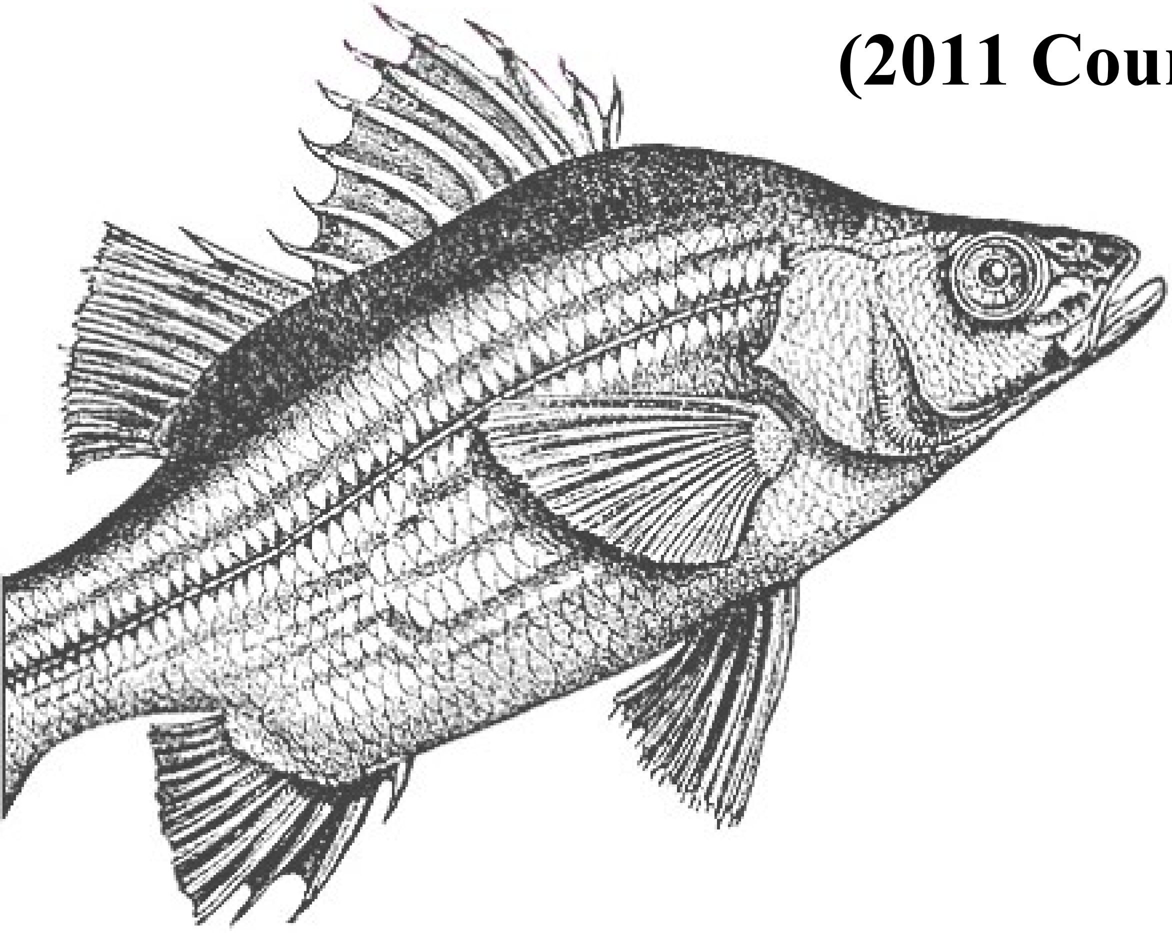
Introduction to Linux Extra Material (Part 1): Advanced Commands and Shell Usage

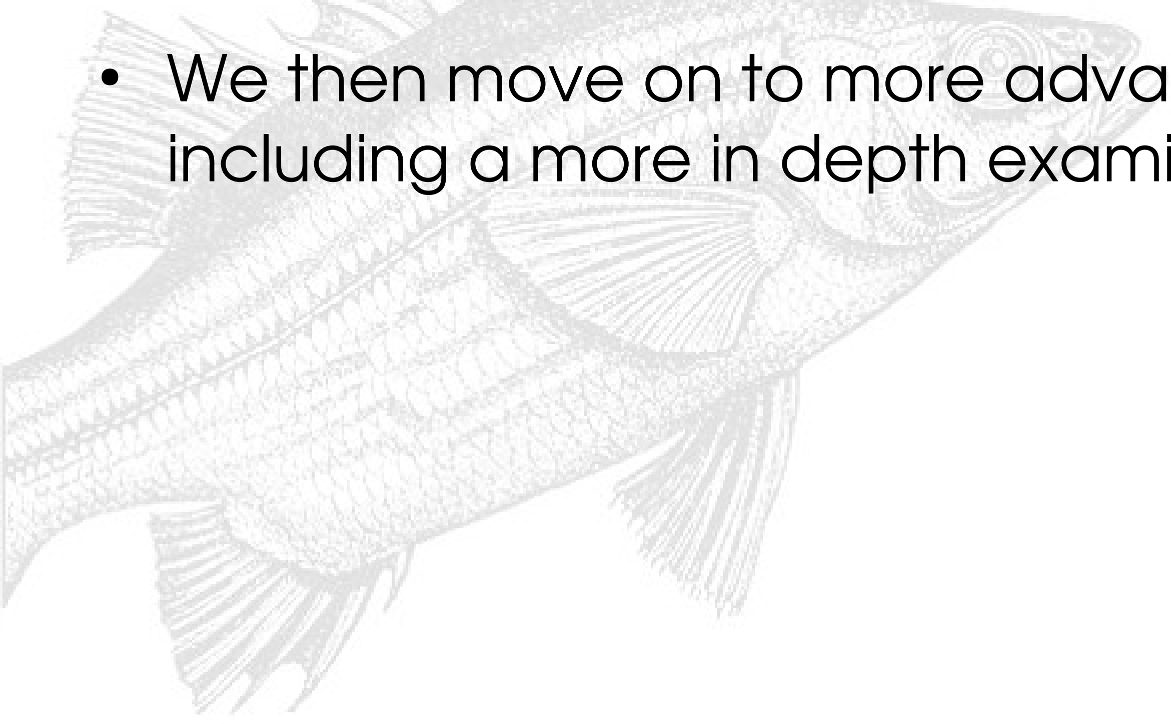
(2011 Course)



Tom Latham
Ben Morgan

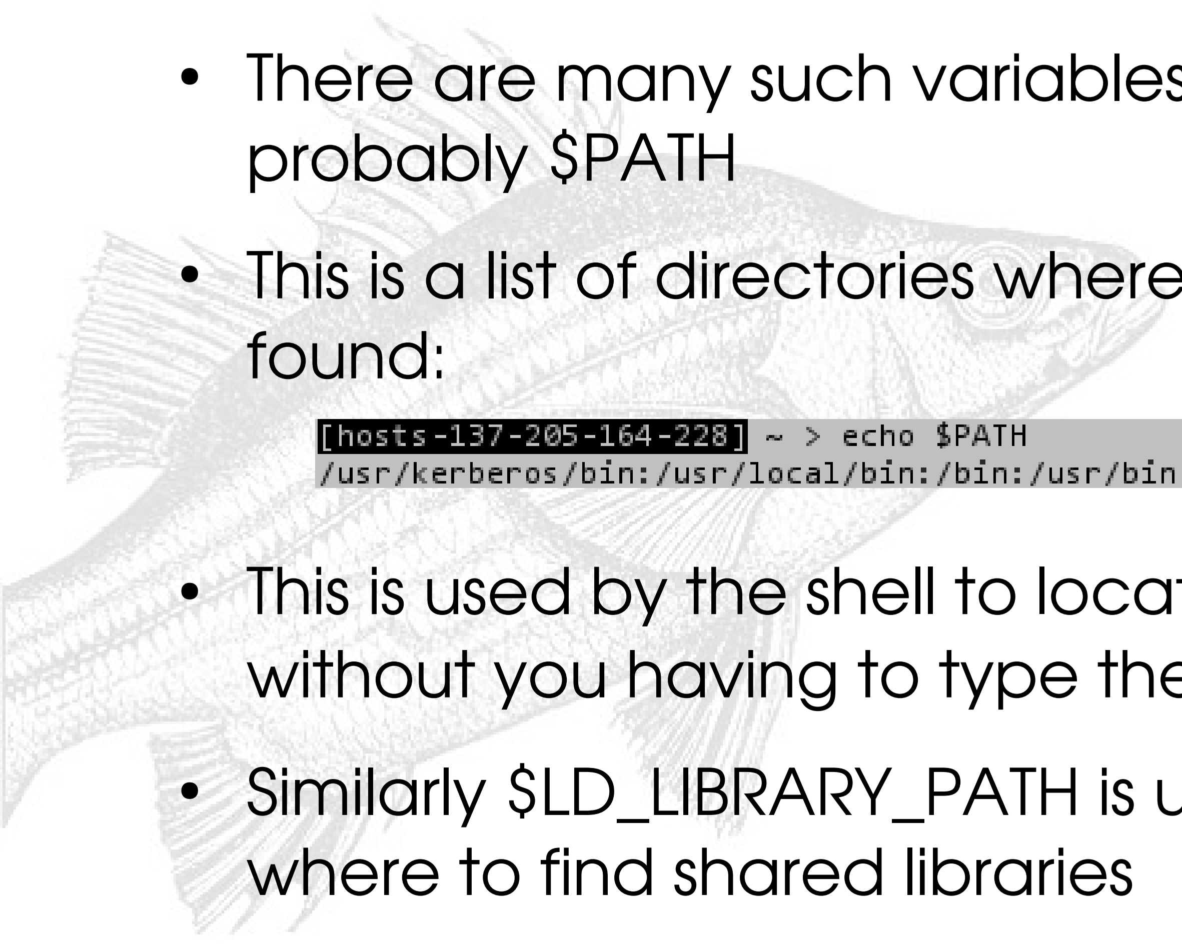
THE UNIVERSITY OF WARWICK

Overview

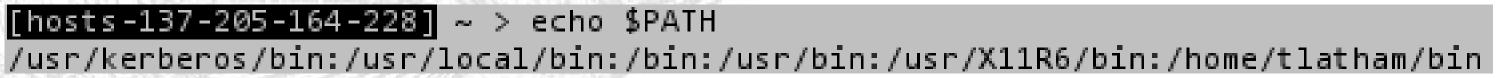


- In this booklet we provide a large amount of extra material
- We start with a few additional things about file permissions, monitoring disk space and basic image manipulation
- Then we go through how to use Secure Shell (SSH) to connect to remote machines, CVS for source control and PBS for submitting batch jobs
- We then move on to more advanced use of the shell, including a more in depth examination of I/O redirection

Environment Variables 1



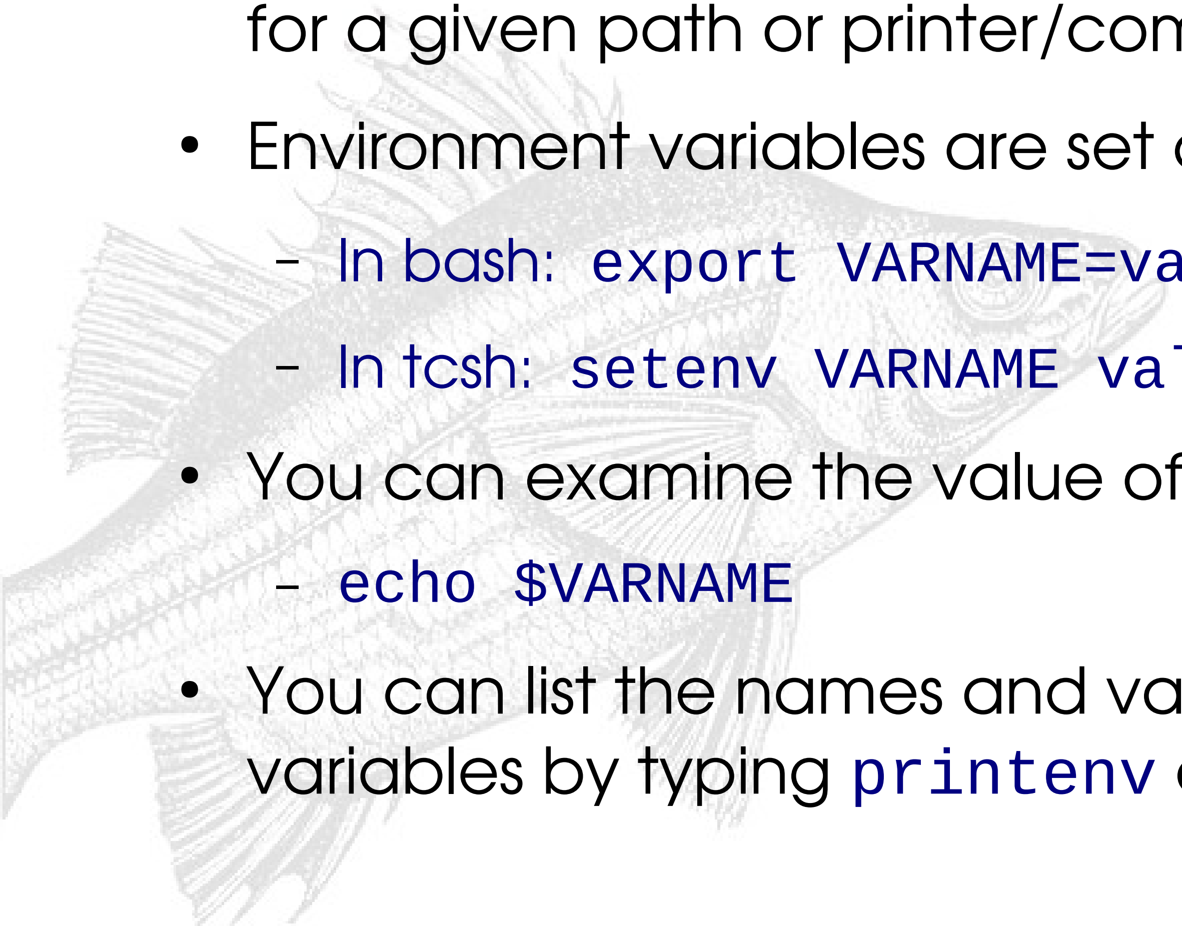
- You can store your editor preference in an "environment variable" called $EDITOR
- Some programs that need to call an editor will query this variable to discover which to use, the default is `vi`
- There are many such variables, the most important is probably $PATH
- This is a list of directories where programs can be found:



```
[hosts-137-205-164-228] ~ > echo $PATH
/usr/kerberos/bin:/usr/local/bin:/bin:/usr/bin:/usr/X11R6/bin:/home/tlatham/bin
```

- This is used by the shell to locate a program, e.g. `ls`, without you having to type the full path
- Similarly $LD_LIBRARY_PATH is used to tell the linker where to find shared libraries

Environment Variables 2



- In addition to the system environment variables there are some associated with particular programs that allow you to tweak their behaviour, e.g. $CVSROOT
- You can also set ones of your own, e.g. as a short cut for a given path or printer/computer name
- Environment variables are set as follows:
  - In bash: `export VARNAME=value`
  - In tcsh: `setenv VARNAME value`
- You can examine the value of an env var by doing:
  - `echo $VARNAME`
- You can list the names and values of all environment variables by typing `printenv` or `env`

Job Control 1

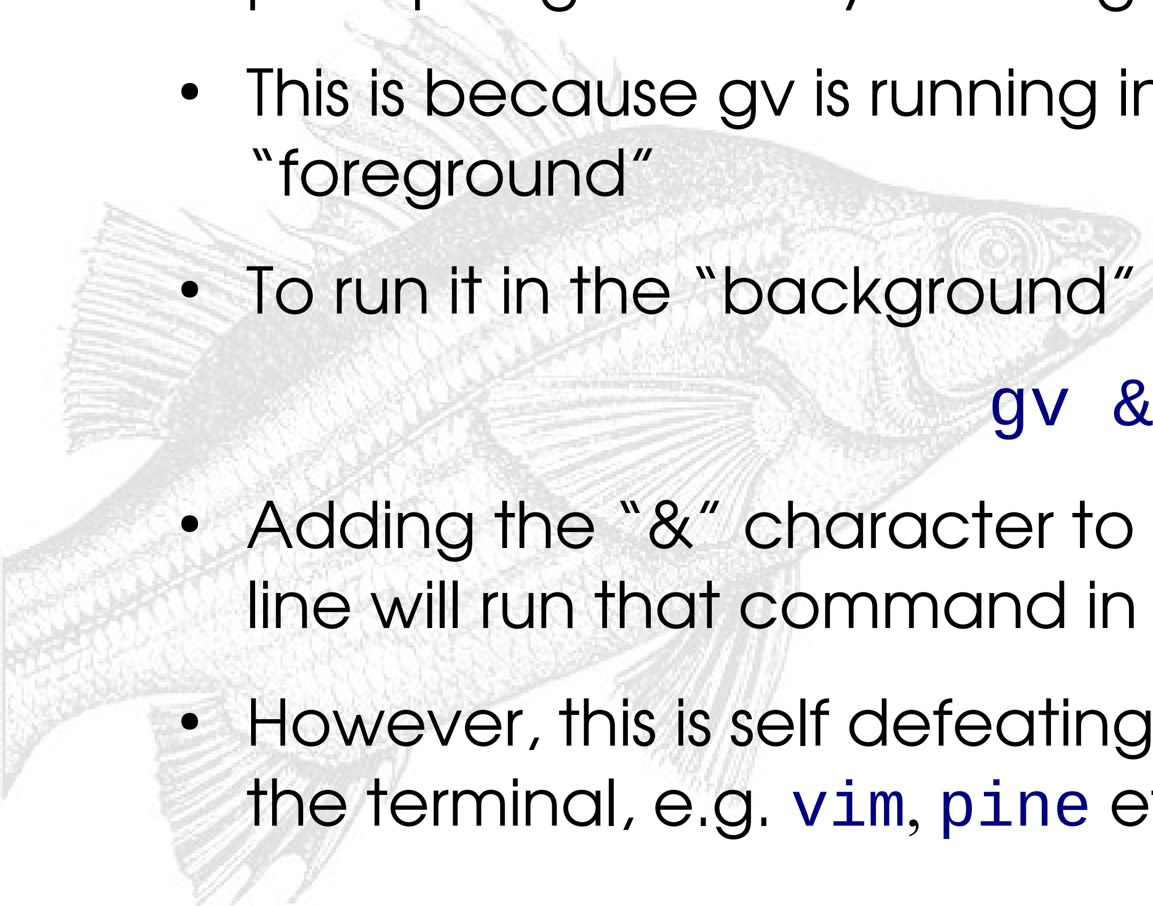
- Start the gv viewer by issuing the command:

```
gv
```

- You'll notice that in the terminal window you have no prompt again until you exit gv
- This is because gv is running in the terminal's "foreground"
- To run it in the "background" you do:

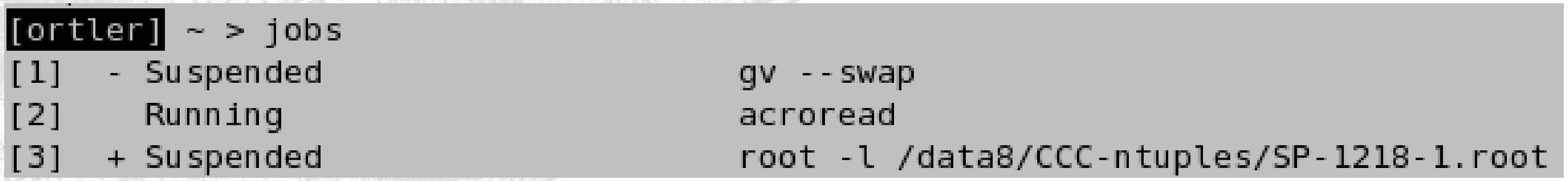
```
gv &
```

- Adding the "&" character to the end of any command line will run that command in the background.
- However, this is self defeating for programs that run in the terminal, e.g. `vim`, `pine` etc.



Job Control 2

- What if you forget to add the “&”?
- In the terminal you can type `Ctrl+z`
- This “suspends” the job
  - You'll see that you can't do anything in the gv window
  - But you get your command prompt back again
- If you type: `jobs` you'll get a list of the jobs running in the current shell and their status:



```
[ortler] ~ > jobs
[1]  - Suspended                      gv --swap
[2]    Running                        acroread
[3]  + Suspended                      root -l /data8/CCC-ntuples/SP-1218-1.root
```

- You can put a suspended job into the background using the `bg` command, e.g.

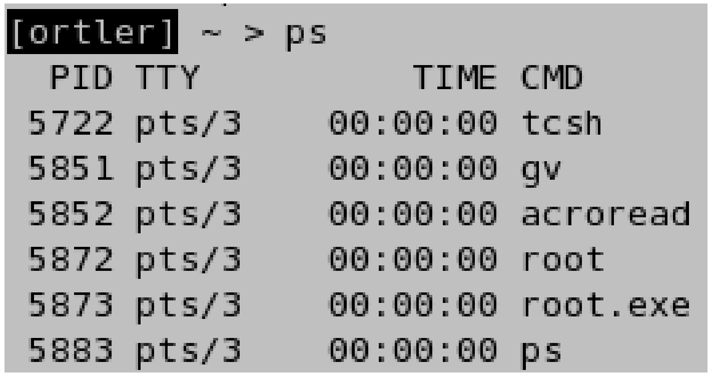
`bg %1`   or   `bg %3`

Job Control 3

- You can of course suspend any job using `Ctrl+z`
- Using the `fg` command you can then bring jobs back to the foreground
- You can kill foreground jobs using `Ctrl+c`
- Background or suspended jobs can be killed using:

  `kill %n` or `kill “id”`

  - where “id” is the unique process ID
- You can find this out by typing: `ps`
- This lists information about all processes running in the current terminal window



```
[ortler] ~ > ps
  PID TTY          TIME CMD
 5722 pts/3    00:00:00 tcsh
 5851 pts/3    00:00:00 gv
 5852 pts/3    00:00:00 acroread
 5872 pts/3    00:00:00 root
 5873 pts/3    00:00:00 root.exe
 5883 pts/3    00:00:00 ps
```

Job Control 4

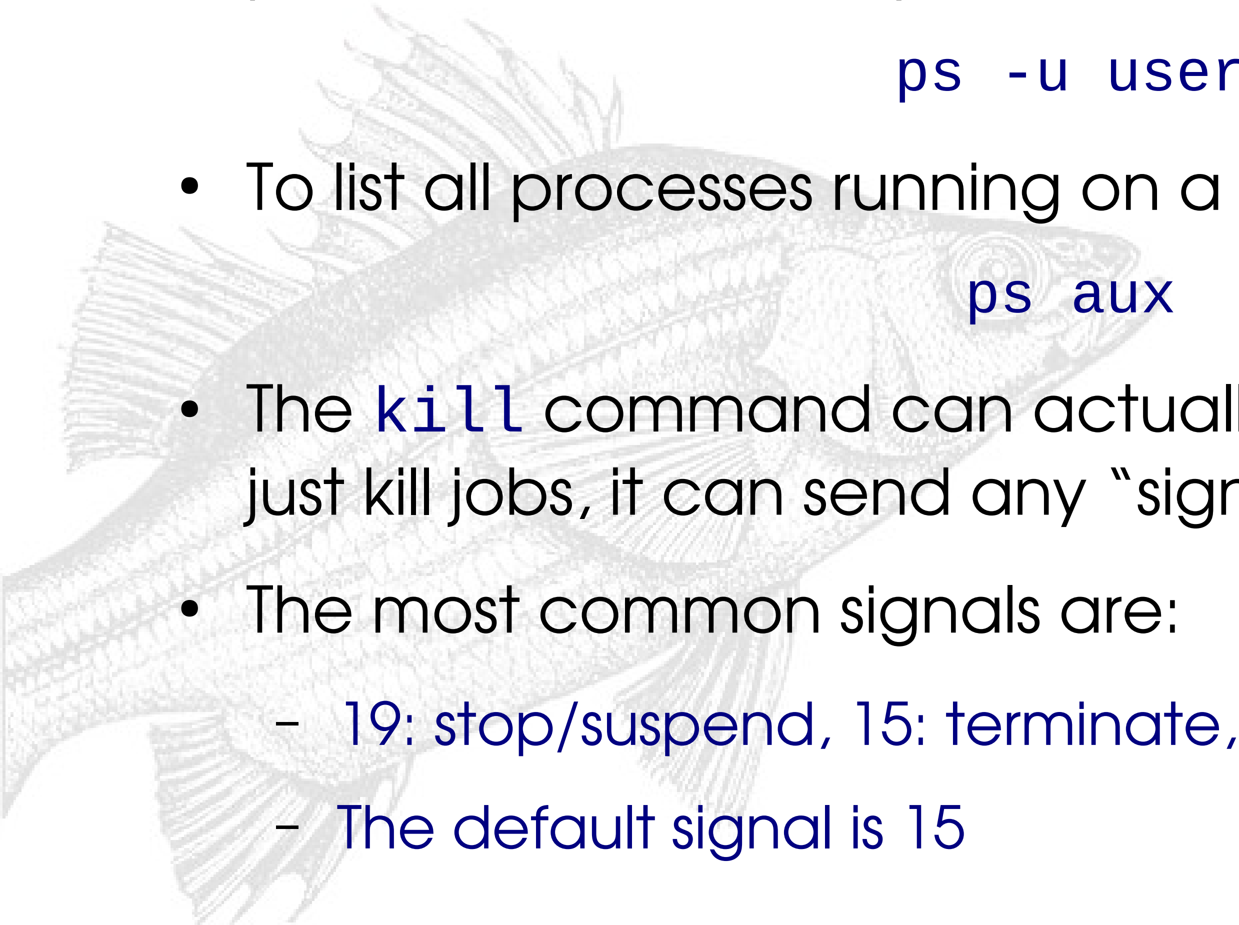
- The `ps` command has many, many options, here are a couple of the most common ones
- To list information on all processes running under a particular username you can do

```
ps -u user
```

- To list all processes running on a machine do

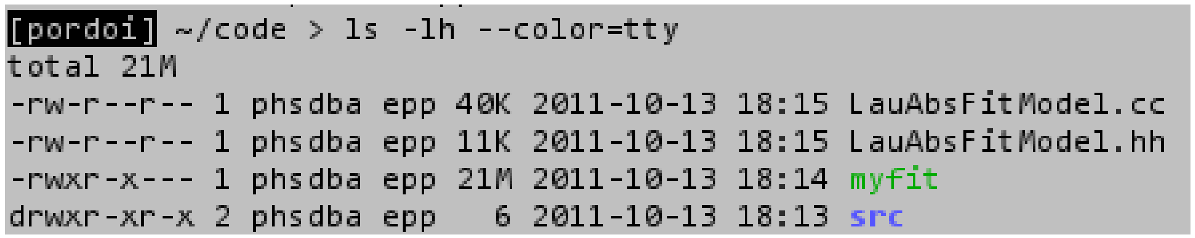
```
ps aux
```

- The `kill` command can actually do a lot more than just kill jobs, it can send any "signal" to a given process
- The most common signals are:
  - 19: stop/suspend, 15: terminate, 9: kill
  - The default signal is 15



File Permissions 1

- File permissions tell you who is allowed to do what to a given file or directory
- There are 3 basic permission types:
  - read, write and execute
- There are 3 sets of people to be given permissions on a file:
  - the owner of the file, users in the file's group and users not in the file's group
- The group allows great flexibility, since group membership is continuously configurable



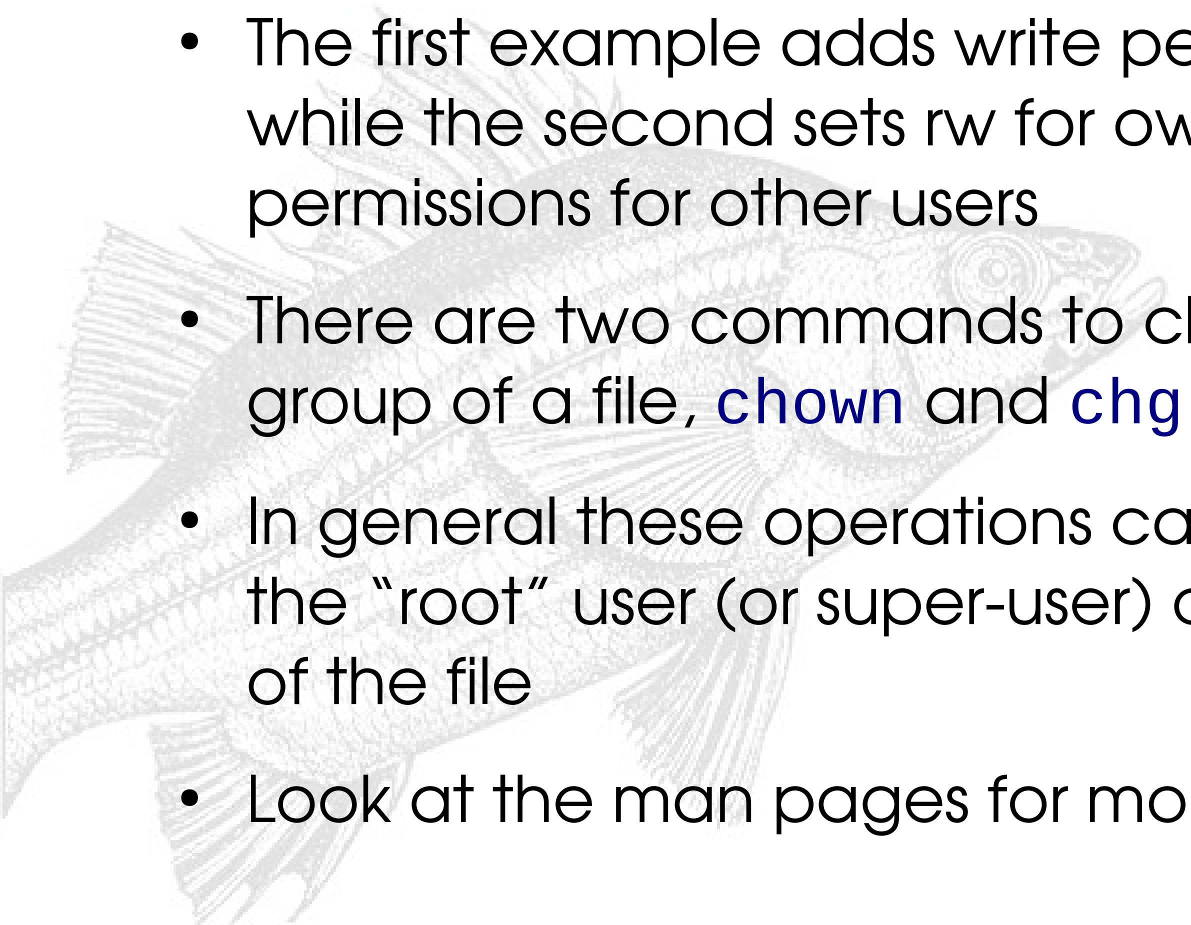
```
[pordoi] ~/code > ls -lh --color=tty
total 21M
-rw-r--r-- 1 phsdba epp 40K 2011-10-13 18:15 LauAbsFitModel.cc
-rw-r--r-- 1 phsdba epp 11K 2011-10-13 18:15 LauAbsFitModel.hh
-rwxr-x--- 1 phsdba epp 21M 2011-10-13 18:14 myfit
drwxr-xr-x 2 phsdba epp   6 2011-10-13 18:13 src
```

File owner has read and write permission.
Users in file's group have read permission.
Other users have read permission.

For this file “other” users do not have any permissions.
Additionally to the permissions detailed above, the owner and members of the file's group have execute permission, which permits those users to execute the program/script.

The “d” here indicates the item is a directory.
The execute permission on directories indicates the ability to “cd” into that directory.

File Permissions 2



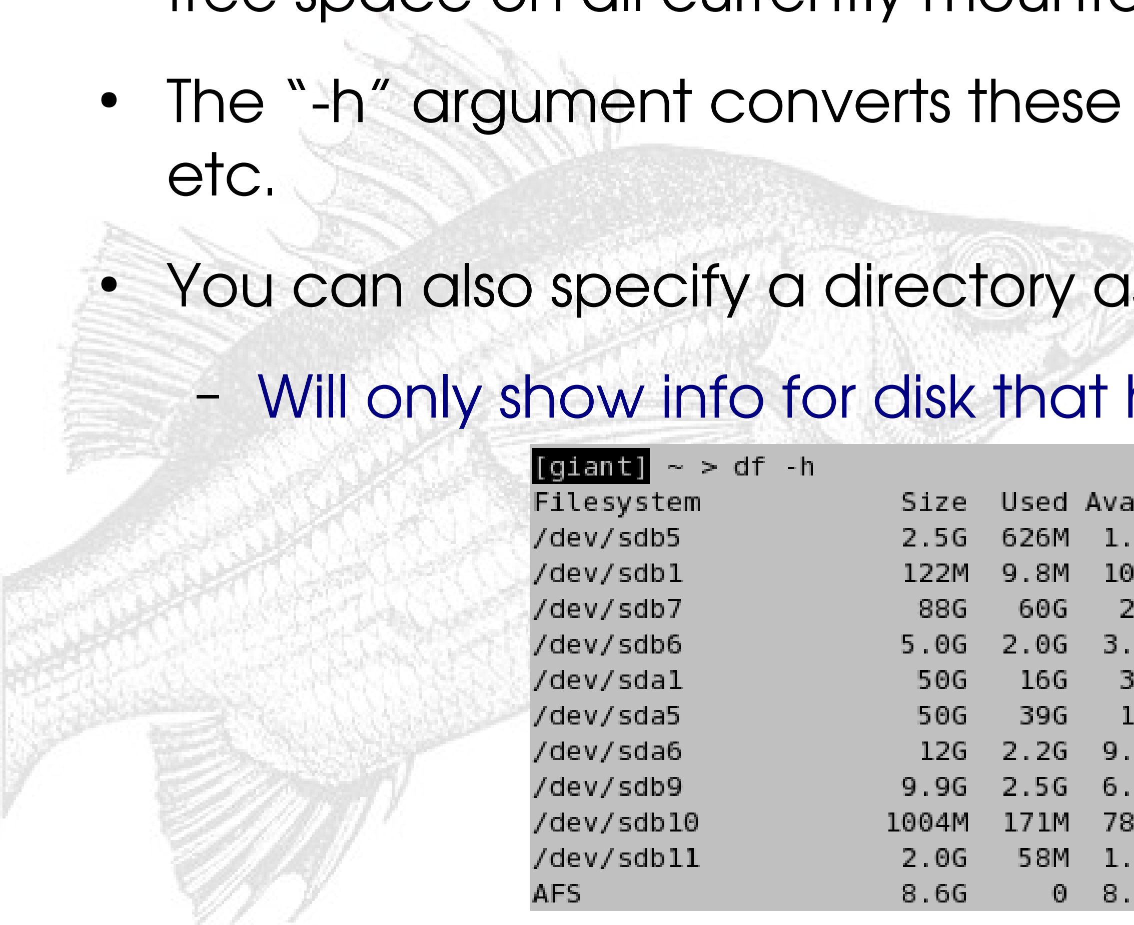
- There is a command to change the permissions of a given file, `chmod` e.g.

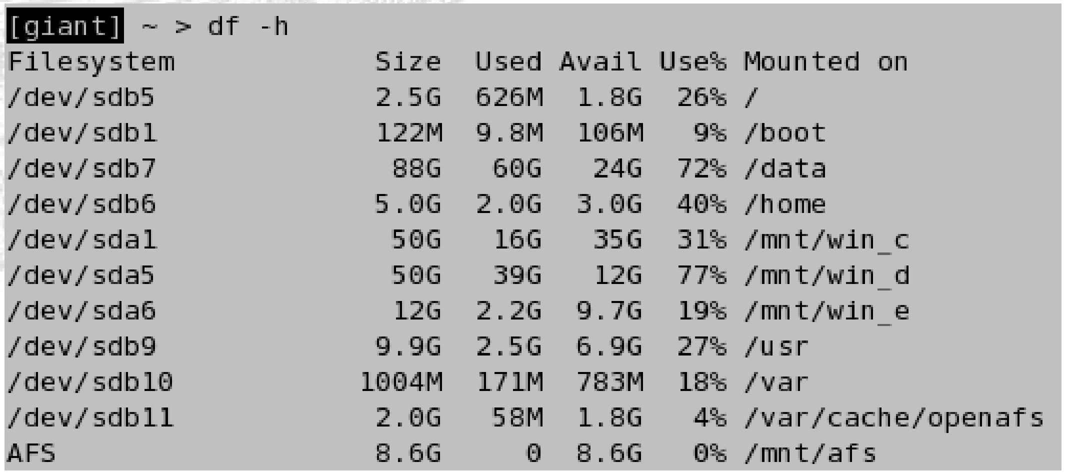
```
chmod g+w myfile

chmod 640 myfile
```

- The first example adds write permission for the group, while the second sets rw for owner, r for group and no permissions for other users
- There are two commands to change the owner or group of a file, `chown` and `chgrp`
- In general these operations can only be performed by the "root" user (or super-user) or sometimes the owner of the file
- Look at the man pages for more information

Disk Space 1

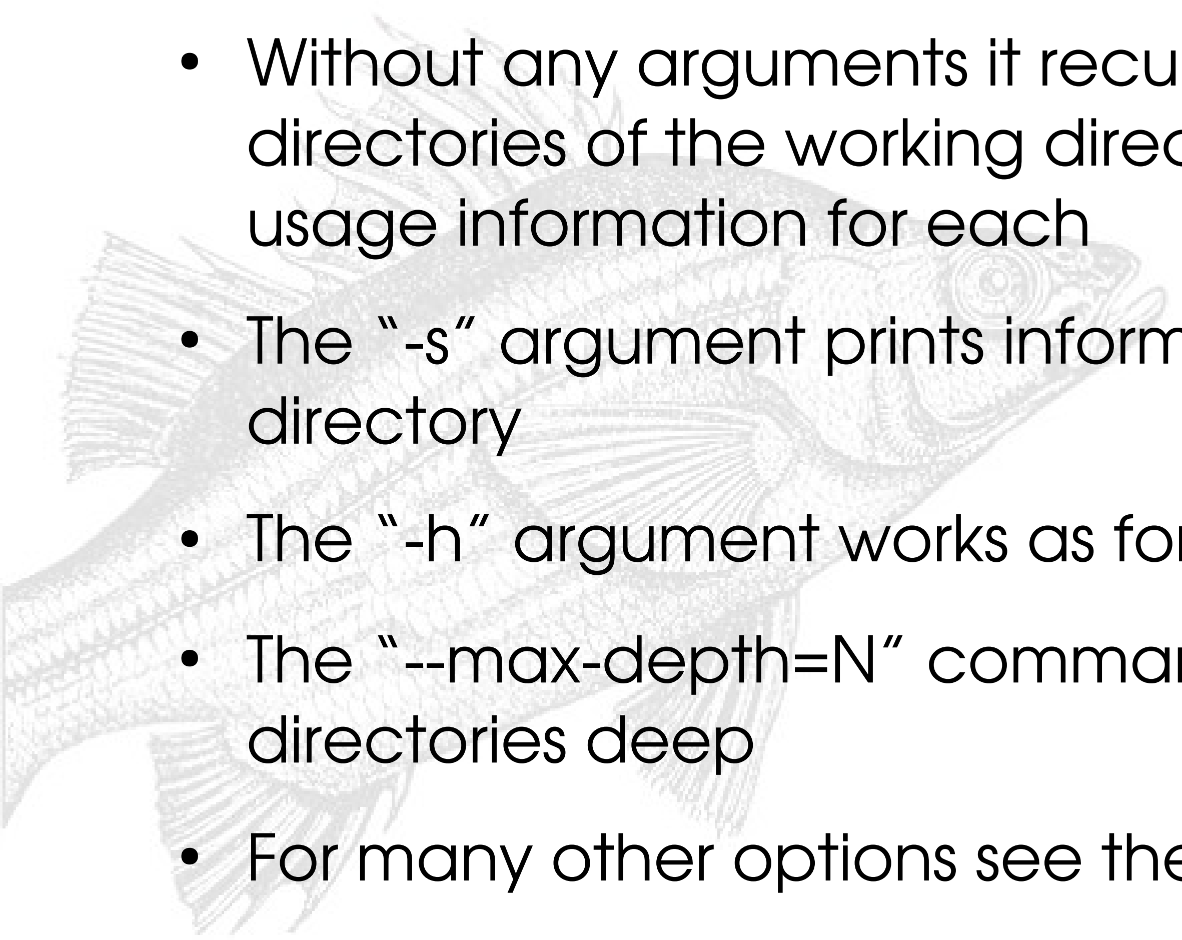
- You'll often need to check the remaining space on a disk
- For this task there is the command df
- Used without any arguments it lists the disk usage and remaining free space on all currently mounted devices
- The "-h" argument converts these amounts to units of kB, MB, GB etc.
- You can also specify a directory as an argument.
  - Will only show info for disk that holds that directory





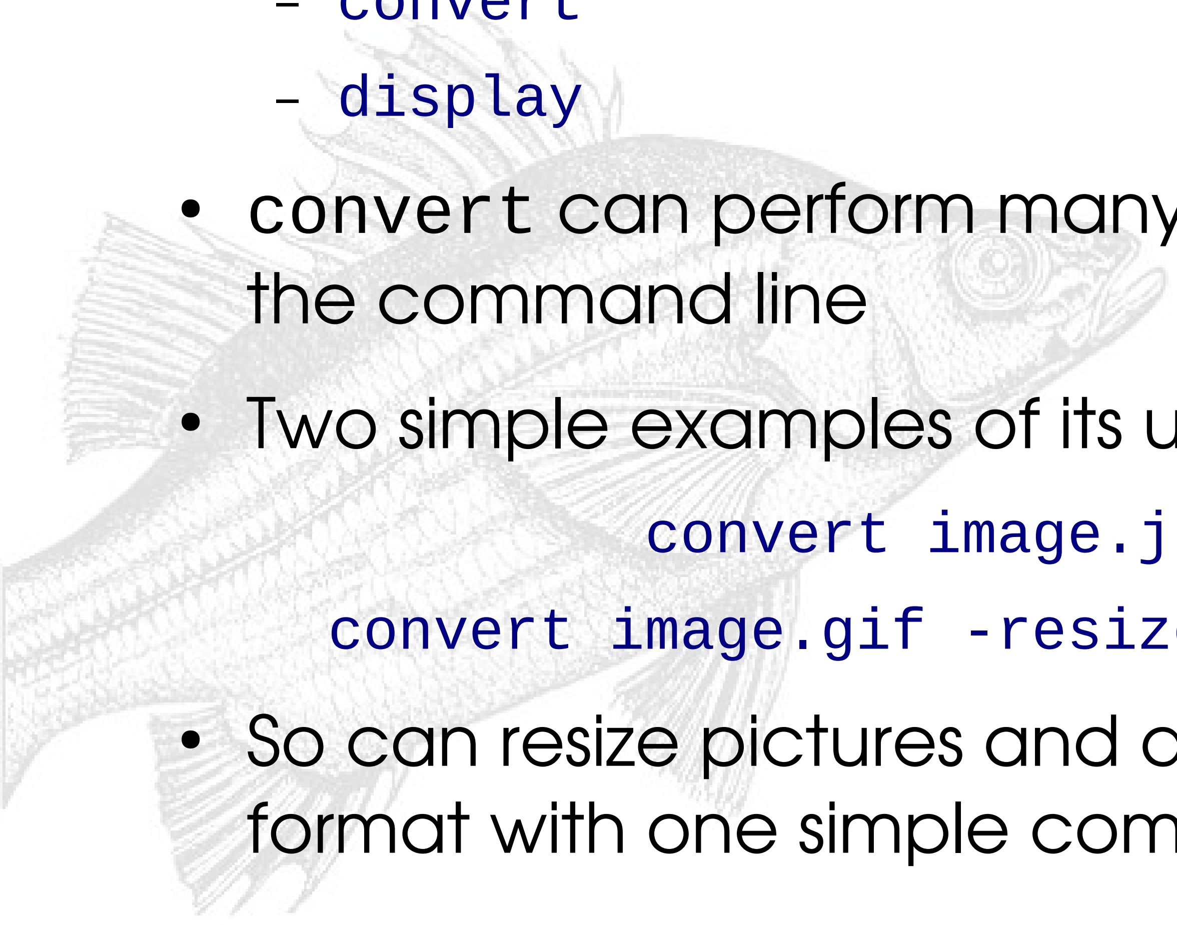
```
[giant] ~ > df -h
Filesystem            Size  Used Avail Use% Mounted on
/dev/sdb5             2.5G  626M  1.8G  26% /
/dev/sdb1             122M  9.8M  106M   9% /boot
/dev/sdb7              88G   60G   24G  72% /data
/dev/sdb6             5.0G  2.0G  3.0G  40% /home
/dev/sda1              50G   16G   35G  31% /mnt/win_c
/dev/sda5              50G   39G   12G  77% /mnt/win_d
/dev/sda6              12G  2.2G  9.7G  19% /mnt/win_e
/dev/sdb9             9.9G  2.5G  6.9G  27% /usr
/dev/sdb10           1004M  171M  783M  18% /var
/dev/sdb11            2.0G   58M  1.8G   4% /var/cache/openafs
AFS                   8.6G     0  8.6G   0% /mnt/afs
```

Disk Space 2



- It is often useful to know exactly how much space you are using in a certain directory or with a certain set of files
- For this there is the command `du`
- Without any arguments it recurses through all sub-directories of the working directory and prints the disk usage information for each
- The "-s" argument prints information only for the current directory
- The "-h" argument works as for `df`
- The "--max-depth=N" command only recurses N directories deep
- For many other options see the man pages

ImageMagick Tools 1

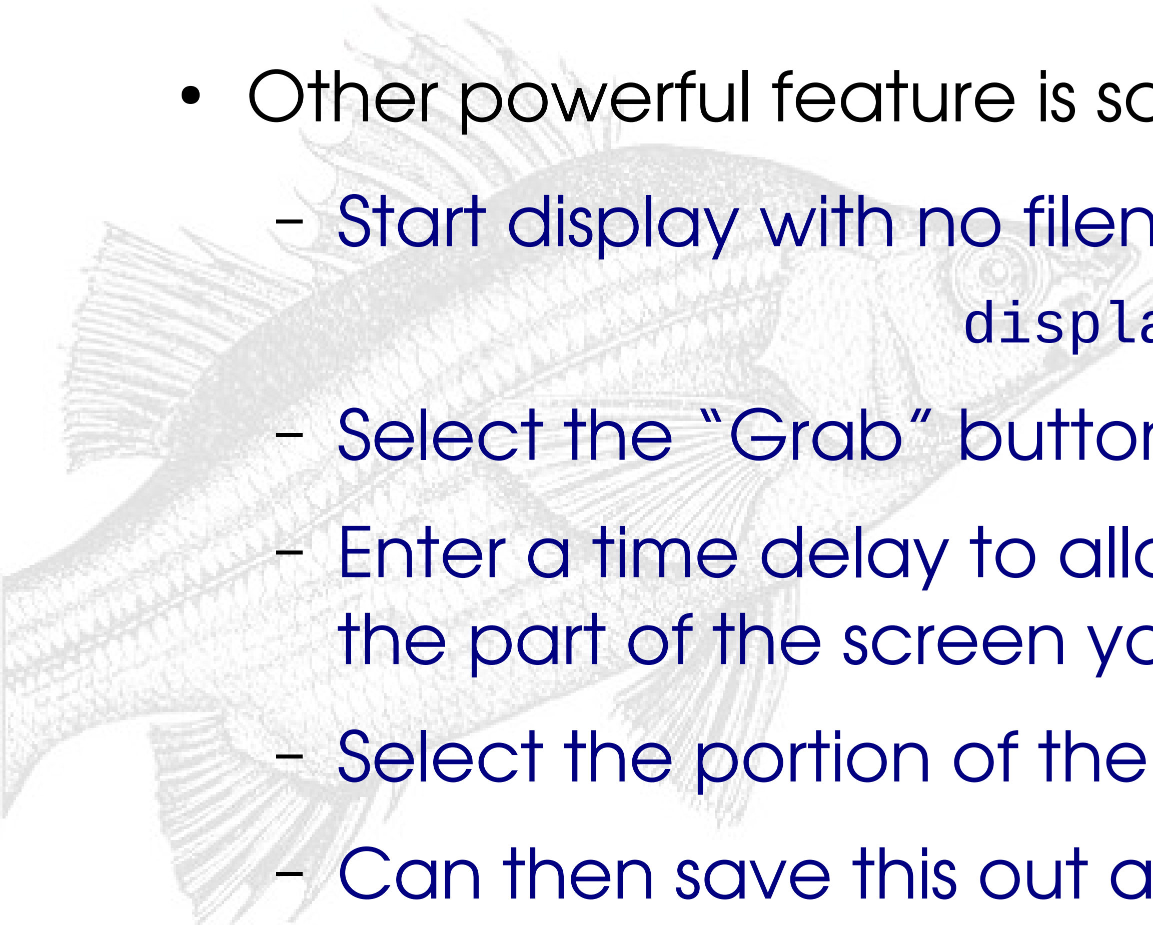


- The ImageMagick toolkit contains many tools for manipulating images
- Two very useful commands are
  - `convert`
  - `display`
- `convert` can perform many complex functions on the command line
- Two simple examples of its use:

```
          convert image.jpg image.png
convert image.gif -resize 50% image-small.gif
```

- So can resize pictures and also change their format with one simple command

ImageMagick Tools 2



- Simplest use for `display` is viewing images:

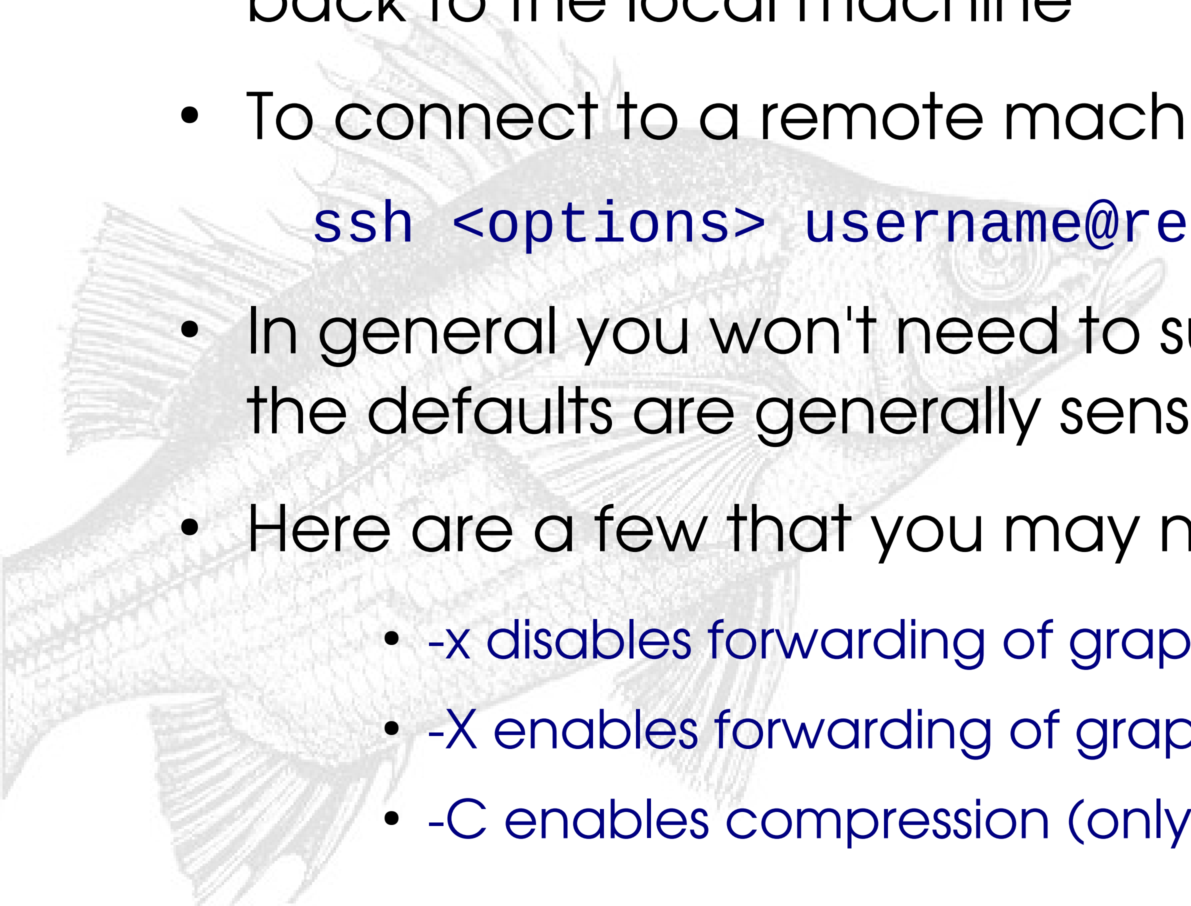
```
display image.eps
```

- Can then save images in other formats (graphical interface to some `convert` functions)
- Other powerful feature is screen grabbing
  - Start display with no filename argument

    ```
    display &
    ```

  - Select the "Grab" button
  - Enter a time delay to allow you to navigate to the part of the screen you wish to grab
  - Select the portion of the screen required
  - Can then save this out as any image format

SSH – Secure SHell



- The SSH protocol allows secure, authenticated connections to remote machines
- It also allows you to treat a shell on a remote machine as if it were local – graphical output etc. is tunnelled back to the local machine
- To connect to a remote machine type:

```
ssh <options> username@remotehost.remotedomain
```

- In general you won't need to supply any options since the defaults are generally sensible
- Here are a few that you may need occasionally:
  - -x disables forwarding of graphics
  - -X enables forwarding of graphics
  - -C enables compression (only for VERY slow connections)

scp

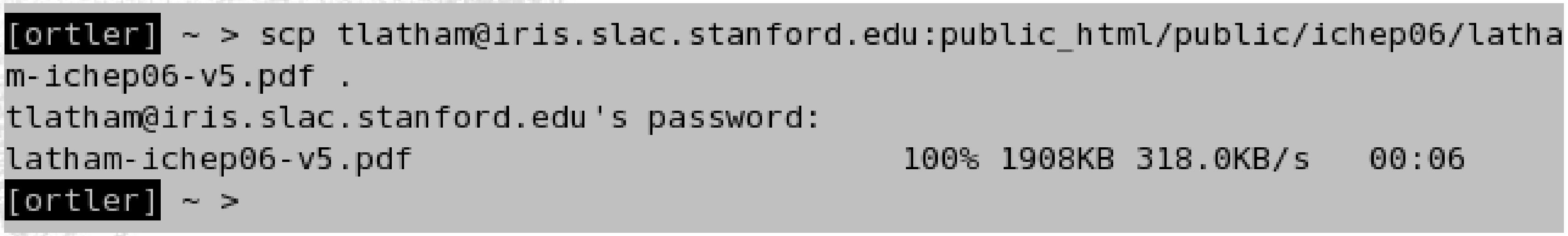
- Copies files from one machine to another
- Uses the ssh protocol
- Copy a remote file to local machine:

```
scp “user@remote.host:/path/to/file” local_file
```

- Copy a local file to a remote machine:

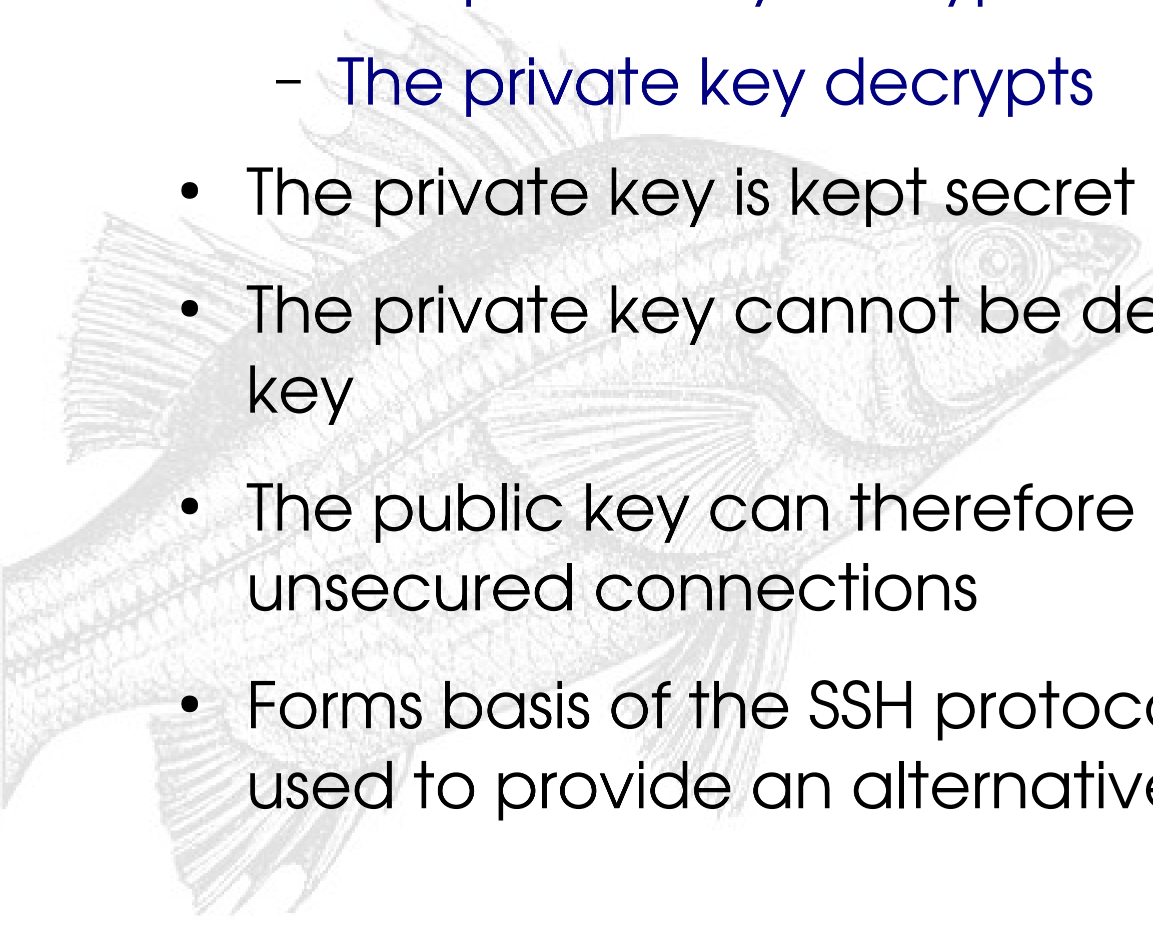
```
scp local_file user@remote.host:/path/to/file
```

- If no remote path supplied defaults to home directory (but you always need the colon)
- Can copy directory structures using “-r” option



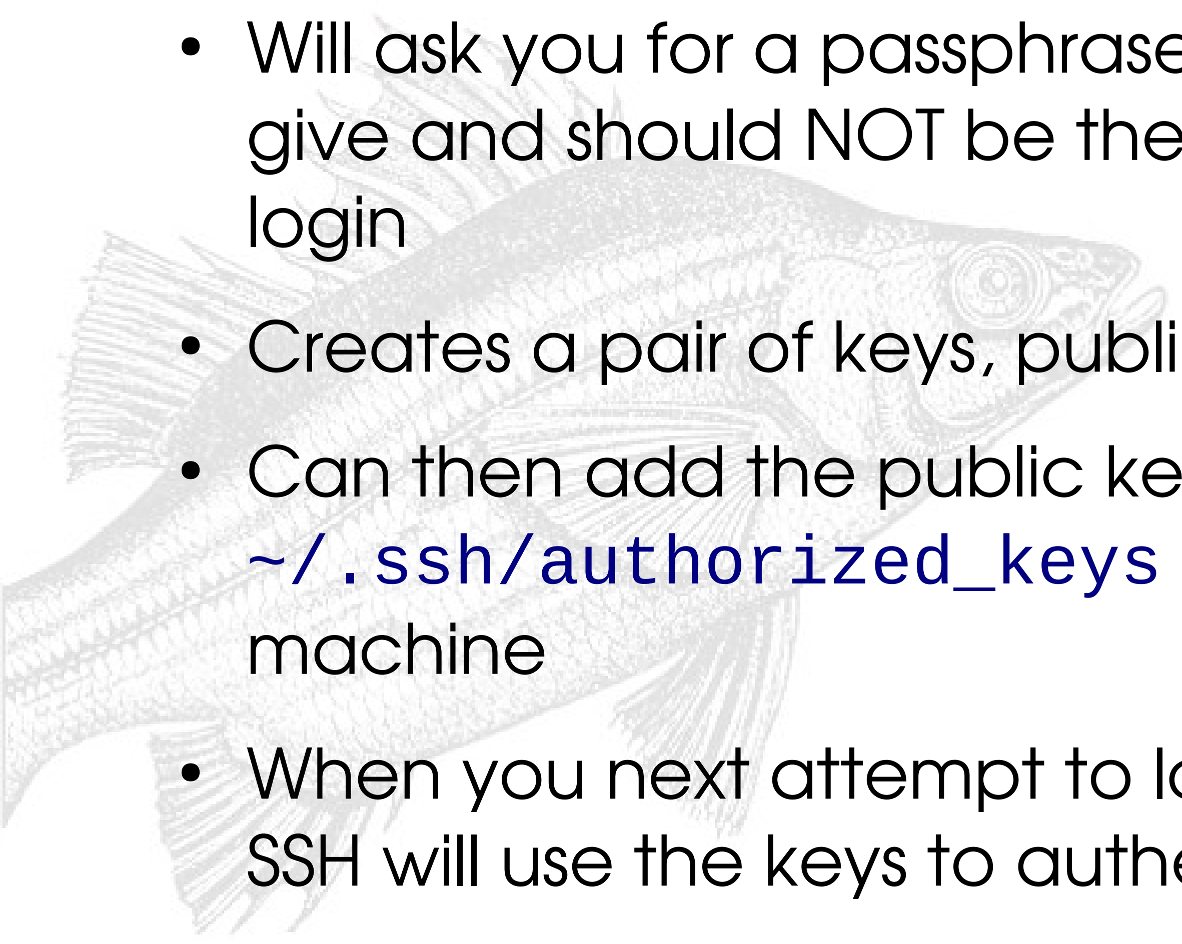
```
[ortler] ~ > scp tlatham@iris.slac.stanford.edu:public_html/public/ichep06/latha
m-ichep06-v5.pdf .
tlatham@iris.slac.stanford.edu's password:
latham-ichep06-v5.pdf                          100% 1908KB 318.0KB/s   00:06
[ortler] ~ >
```

Public/Private Keys



- Public/private key cryptography was a great step forward in computer security
- A pair of keys is created:
  - The public key encrypts
  - The private key decrypts
- The private key is kept secret (hence the name!)
- The private key cannot be determined from the public key
- The public key can therefore be transmitted freely over unsecured connections
- Forms basis of the SSH protocol itself but can also be used to provide an alternative authentication method

SSH Keys



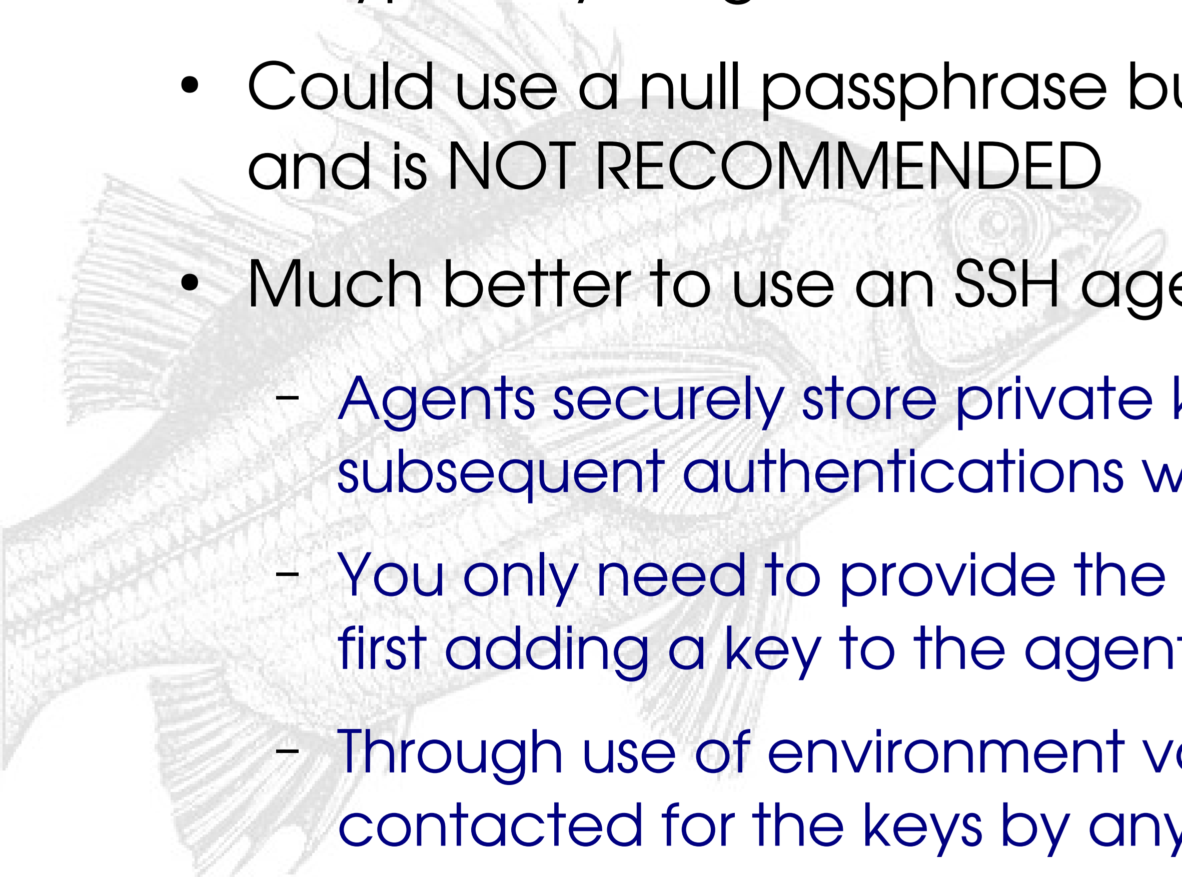
- You can create SSH keys using the command:

```
ssh-keygen -t rsa -b 1024
```

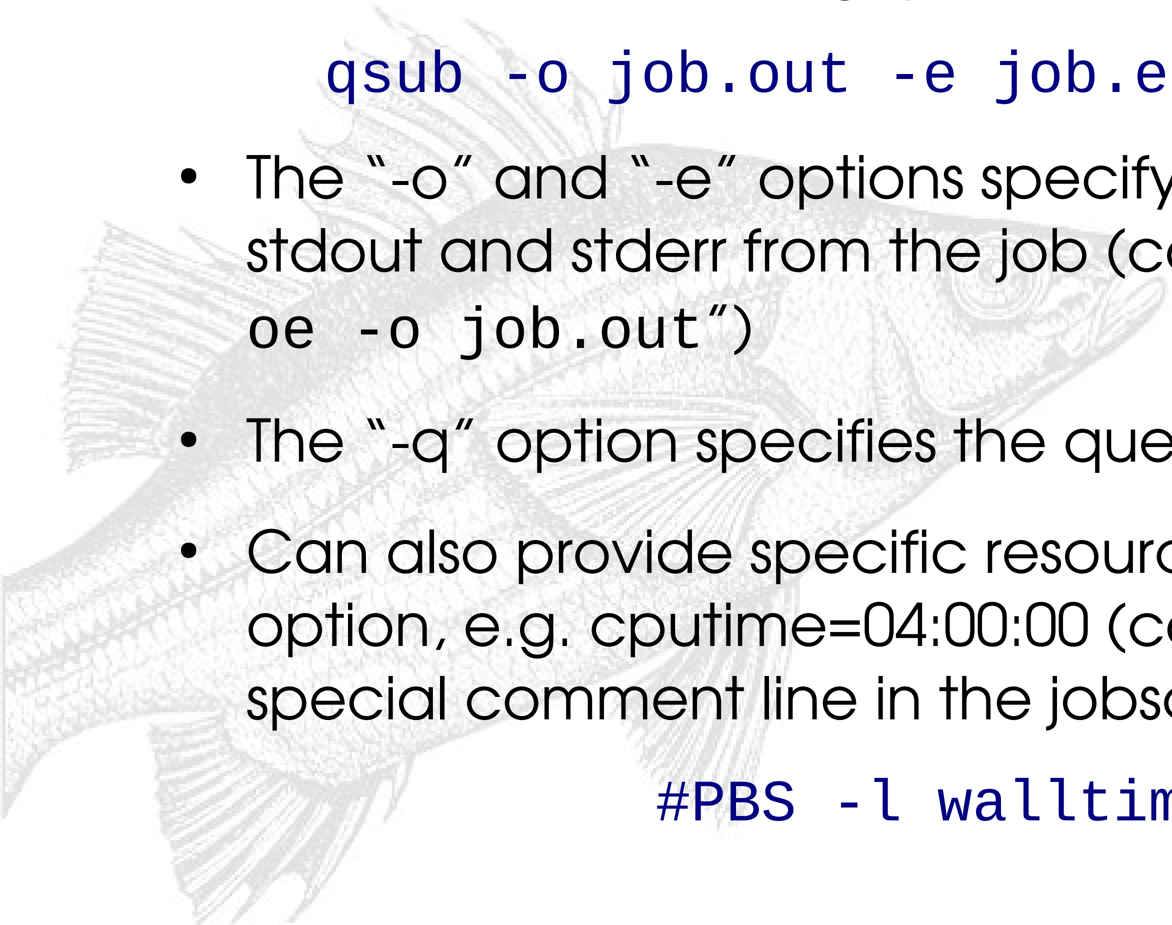
  - can also specify `dsa` key type
- Will ask you for a passphrase, which you should give and should NOT be the same as your system login
- Creates a pair of keys, public and private
- Can then add the public key into the `~/.ssh/authorized_keys` file on a remote machine
- When you next attempt to log in to that machine SSH will use the keys to authenticate

SSH Agents

- You are now being asked for your SSH key passphrase when you authenticate rather than your remote login password
- Not a great step forward, would rather we didn't need to type anything
- Could use a null passphrase but this is not at all secure and is NOT RECOMMENDED
- Much better to use an SSH agent:
  - Agents securely store private keys to be used for subsequent authentications with remote systems
  - You only need to provide the passphrase once, when first adding a key to the agent
  - Through use of environment variables the agent is contacted for the keys by any program requiring them



Portable Batch System



- PBS is a system for submitting jobs to a queue for processing by dedicated batch machines
- The job is a shell script that you provide
- Job is submitted using `qsub` command, e.g.

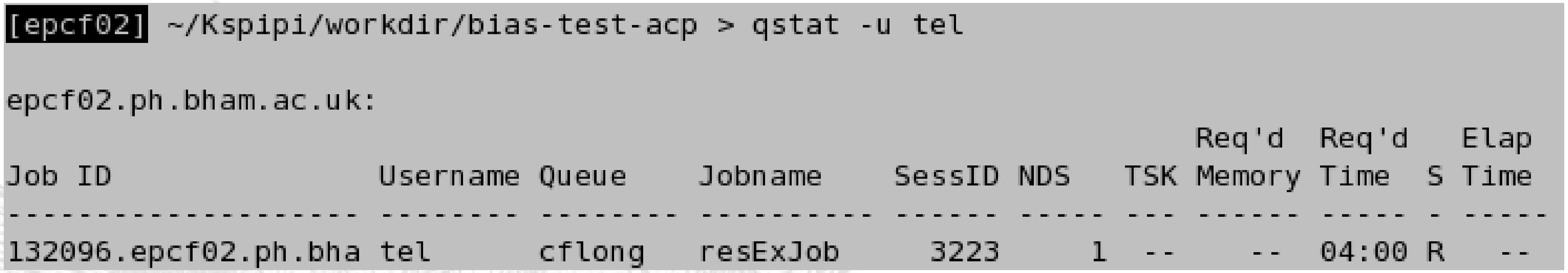
```
qsub -o job.out -e job.err -q long jobscript
```

- The "-o" and "-e" options specify the log files to hold the stdout and stderr from the job (can combine these using "`-j oe -o job.out`")
- The "-q" option specifies the queue (not always needed)
- Can also provide specific resource requests using the "-l" option, e.g. cputime=04:00:00 (can also be done using a special comment line in the jobscript):

```
#PBS -l walltime=04:00:00
```

PBS – continued

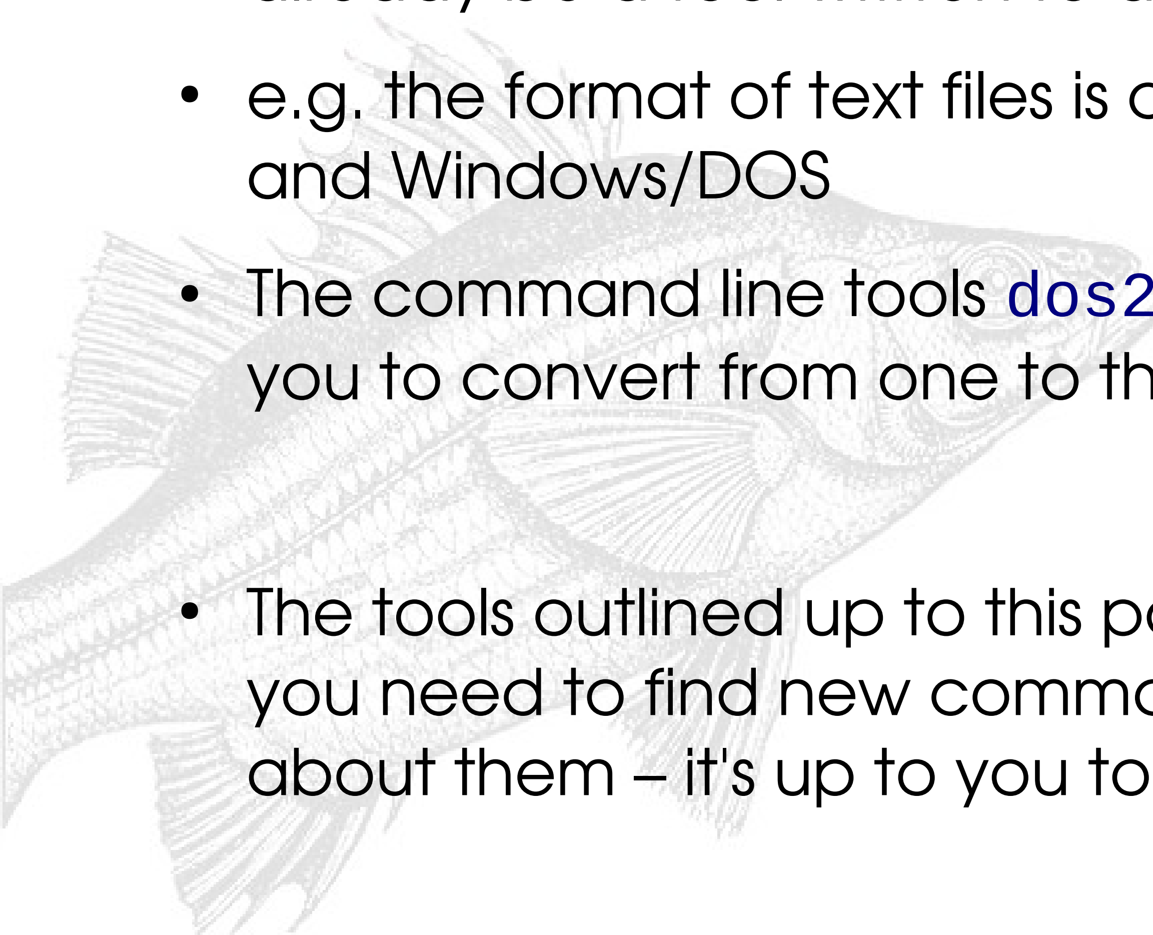
- Once jobs are submitted can check on their progress using the `qstat` command
- The "-u username" option can be used to limit output to info on your jobs



```
[epcf02] ~/Kspipi/workdir/bias-test-acp > qstat -u tel

epcf02.ph.bham.ac.uk:
                                                                   Req'd  Req'd    Elap
Job ID               Username Queue    Jobname    SessID NDS   TSK Memory Time   S Time
-------------------- -------- -------- ---------- ------ ----- --- ------ ----- - -----
132096.epcf02.ph.bha tel      cflong   resExJob     3223     1  --     --  04:00 R   --
```

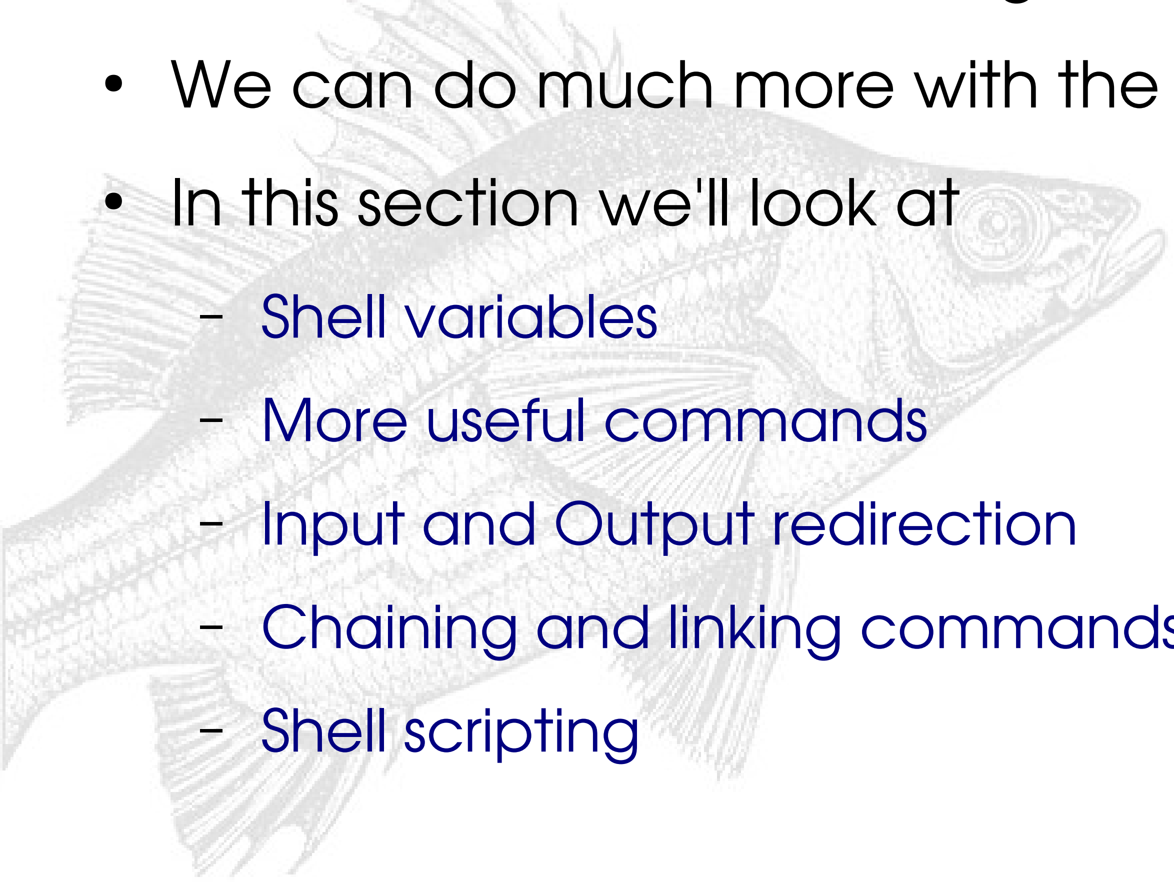
- Most fields are self explanatory, the "S" field shows the status, "R" meaning running, "Q" meaning queued i.e. waiting to start. See the man pages for more possible states.
- When the job ends the log files are returned from the batch worker to the specified location

Other handy tools



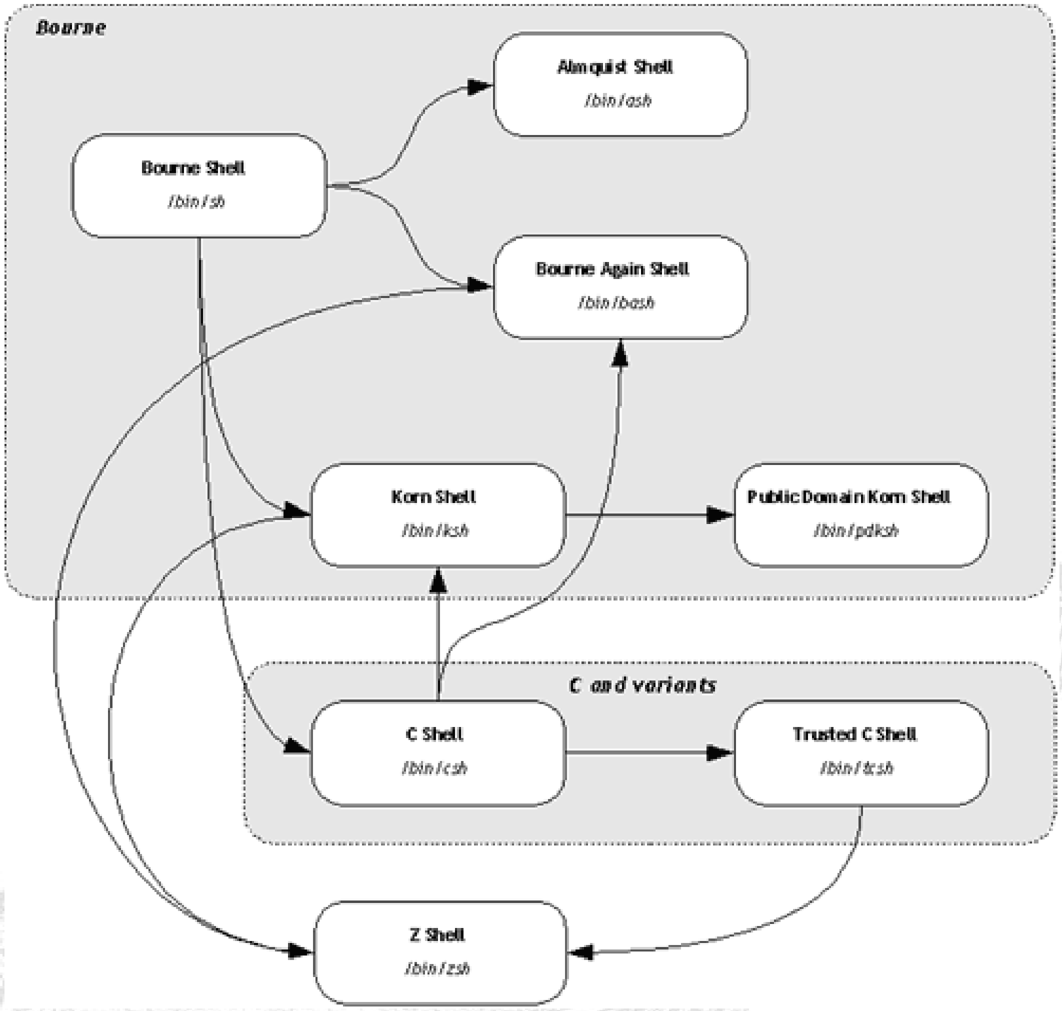
- There are many handy command line tools installed on most Linux systems
- If you run into a problem there will, in many cases, already be a tool written to do the job
- e.g. the format of text files is different between UNIX and Windows/DOS
- The command line tools `dos2unix` and `unix2dos` allow you to convert from one to the other.
- The tools outlined up to this point provide you with all you need to find new commands and information about them – it's up to you to explore further.

Shells in more detail



- The shell provides a ***Command Line Interface*** between the user and the OS.
- Note that it's nothing more than another program.
- So far we've looked at using the shell to run programs.
- We can do much more with the shell though.
- In this section we'll look at
  - Shell variables
  - More useful commands
  - Input and Output redirection
  - Chaining and linking commands
  - Shell scripting

Shell Flavours



- Several different shells available.
- `bash` and `(t)csh` are the most common of the two main families.
- Unfortunately, the different families have different syntax and behaviour for certain operations.
- This booklet is a *little* `bash` specific, but we'll point out the main `(t)csh` variations.
- ***Shell choice is personal – so experiment – and argue about the relative merits of each with other users!***

Changing Shell

- As the shell is just another program, you can start a shell within your existing shell, e.g.

```
[me@here ]$ tcsh
```

  - *(NB – I'll write the command prompt from now on for clarity)*

```
[me@here ]$ ...Some tcsh stuff...
[me@here ]$ exit
```

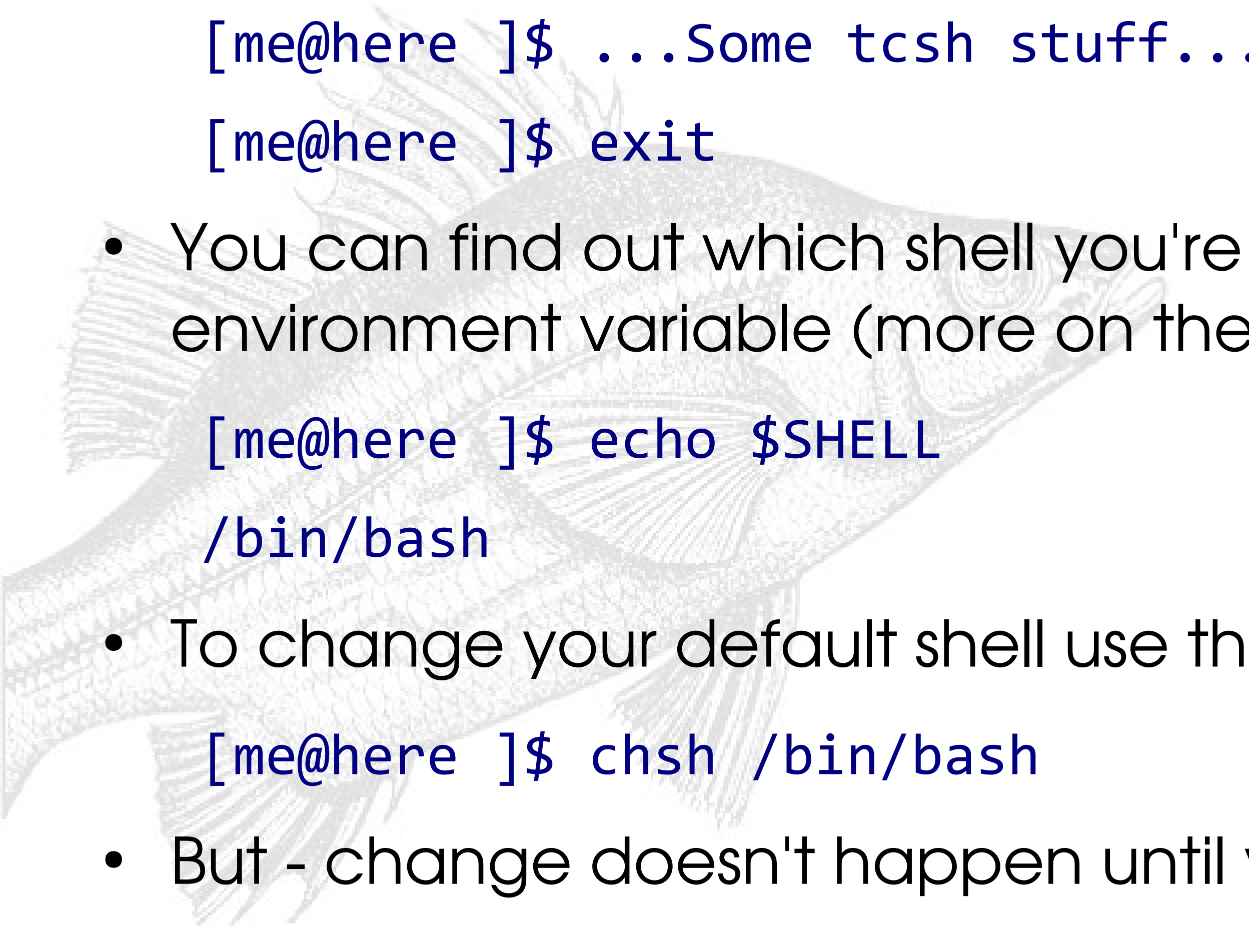
- You can find out which shell you're using through the `SHELL` environment variable (more on these later):

```
[me@here ]$ echo $SHELL
/bin/bash
```

- To change your default shell use the `chsh` command:

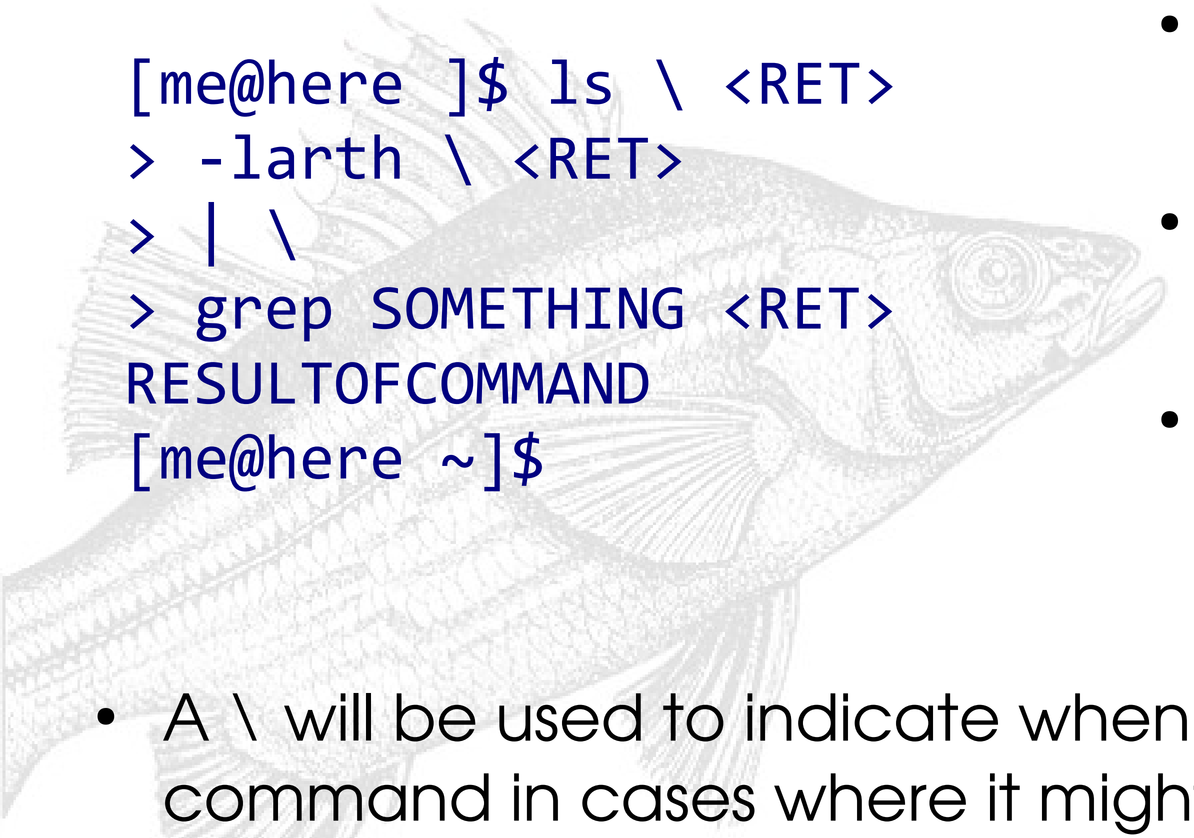
```
[me@here ]$ chsh /bin/bash
```

- But - change doesn't happen until you login again.



Complex Commands

- As we go through this section, commands will get longer.
- Whilst the shell will wrap text onto a new line automatically, you can also force this with a '\' as follows.



```
[me@here ]$ ls \ <RET>
> -larth \ <RET>
> | \
> grep SOMETHING <RET>
RESULTOFCOMMAND
[me@here ~]$
```

- NB: <RET> means press return (just here for clarity).
- See that you can split up command and its options.
- Don't worry about the ' | ' and the `grep` command, we'll deal with those later.

- A \ will be used to indicate when we have a single command in cases where it might not be obvious
- It'll also be useful later in scripts to clarify commands.

Shell Variables 1: bash

- You can define variables inside the shell

```
[me@here ~]$ myVar=”hello”
```

Variable value

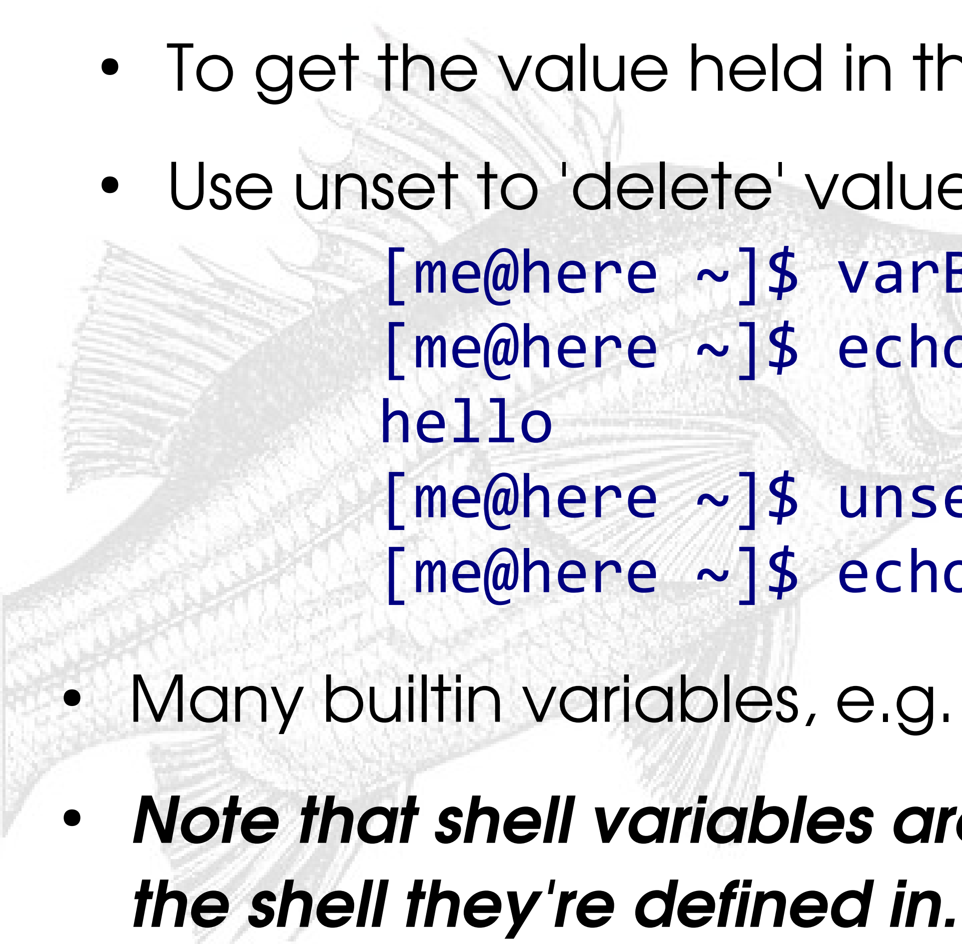
Variable name

Note lack of spaces!

- To get the value held in the variable, preface name with `$`
- Use unset to 'delete' value in variable.

```
[me@here ~]$ varB=$myVar
[me@here ~]$ echo $varB
hello
[me@here ~]$ unset myVar
[me@here ~]$ echo $myVar
```

- Many builtin variables, e.g. `$SHELL` (current shell).
- ***Note that shell variables are only visible in ('have scope in') the shell they're defined in.***

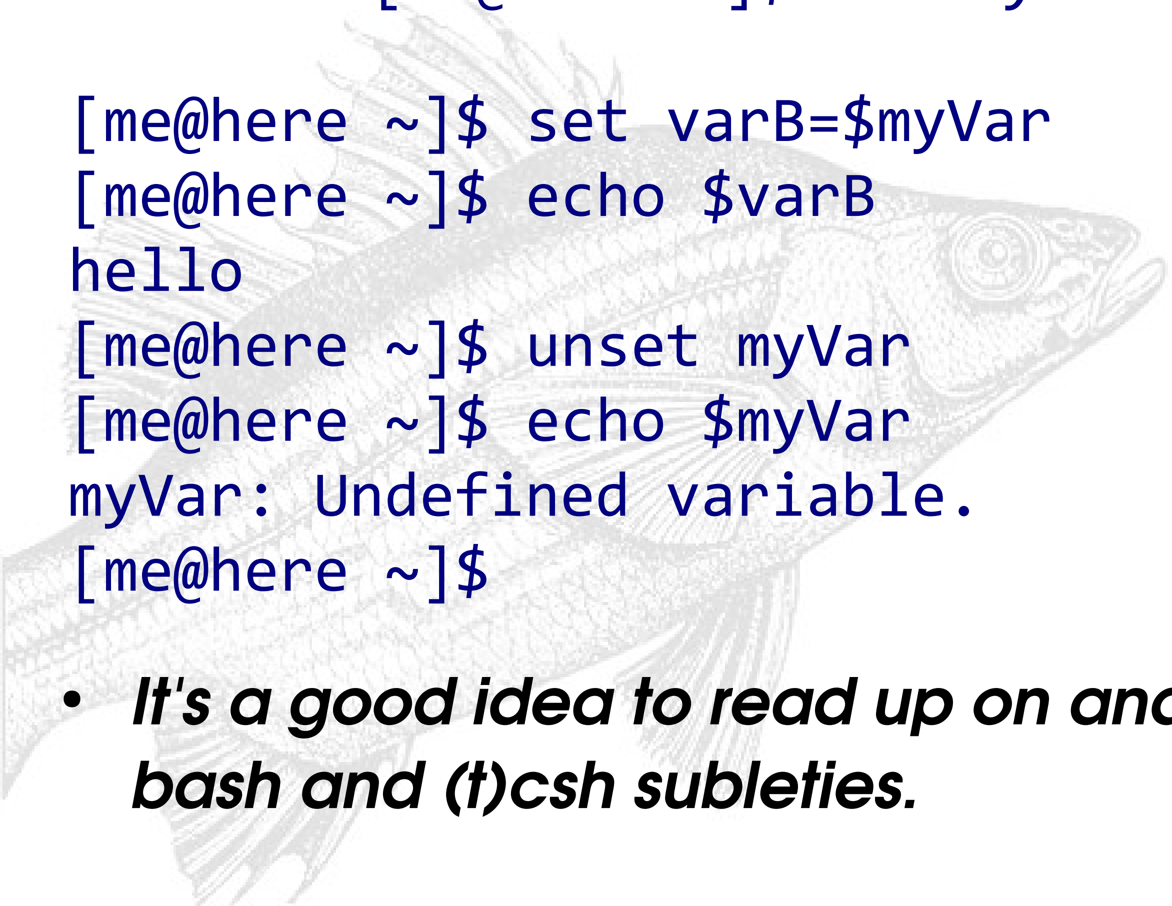


Shell Variables 2: (t)csh

- Unfortunately, bash and (t)csh differ even at the level of variables...
- In (t)csh, variables are defined using `set`

Spaces optional!

```
[me@here ~]$ set myVar = ”hello”
```



```
[me@here ~]$ set varB=$myVar
[me@here ~]$ echo $varB
hello
[me@here ~]$ unset myVar
[me@here ~]$ echo $myVar
myVar: Undefined variable.
[me@here ~]$
```

- Otherwise, use of variables is the same as in bash.
- `unset` is also the same in (t)csh.

- ***It's a good idea to read up on and experiment with these bash and (t)csh subleties.***

Environment Variables: bash

- If we want variables to be visible in child processes of our shell we, in bash, '**export**' them to ***Environment Variables***.
- In bash, try this example:

```
[me@here ~]$ myVar=”hello”
[me@here ~]$ bash
[me@here ~]$ echo $myVar

[me@here ~]$ exit
[me@here ~]$ export myVar
[me@here ~]$ bash
[me@here ~]$ echo $myVar
hello
[me@here ~]$ exit
```

Define the variable, then start a subshell.

Oops, no variable! Exit the subshell.

We export our variable and restart a subshell. Bingo! Variable visible in subshell.

- ***Try This:*** What happens to the value of myVar in the main shell if you change its value in the main shell (and vice versa)?

Environment Variables: (t)csh

- (t)csh uses setenv to specify environment variables.
- Whereas bash 'promotes' variables to the environment, (t)csh treats them separately:

```
[me@here ~]$ set myVar=”hello”
[me@here ~]$ echo $myVar
hello
[me@here ~]$ setenv myVar “goodbye”
[me@here ~]$ echo $myVar
hello
[me@here ~]$ tcsh
[me@here ~]$ echo $myVar
goodbye
[me@here ~]$ exit
[me@here ~]$ unsetenv myVar
```

If we did want to have an EV “hello” we'd use $myVar here

Shell variables have priority

But EV visible in subshell

Have to use unsetenv to unset EVs.

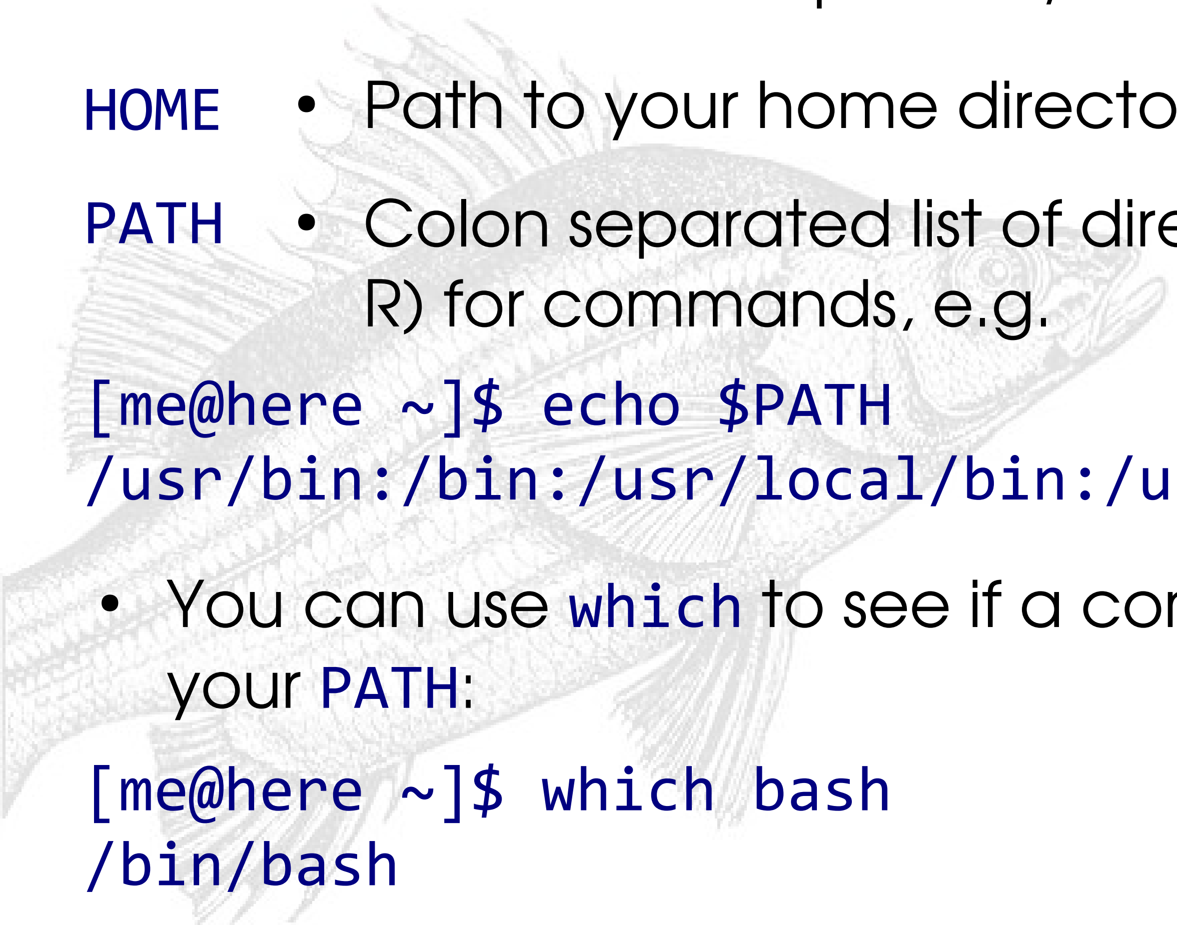
- ***There's much more on these wonderful differences in the main textbooks on bash and (t)csh (see later).***

More Environment Variables

- Linux (and other Unices) uses many environment variables, a complete list of those declared can be output via

```
[me@here ~]$ env                    [me@here ~]$ printenv
```

- Some of the more important/useful ones are:



`HOME` • Path to your home directory, e.g. `/home/me`

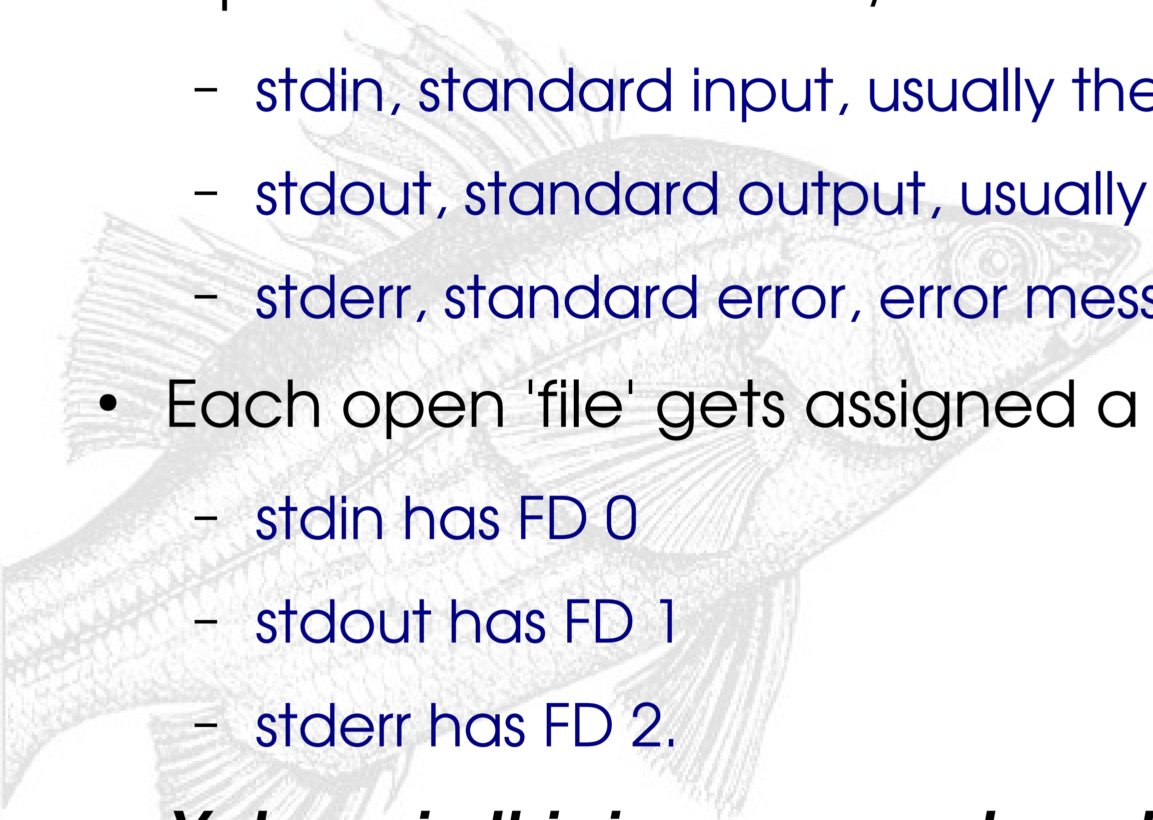
`PATH` • Colon separated list of directories searched (from L to R) for commands, e.g.

```
[me@here ~]$ echo $PATH
/usr/bin:/bin:/usr/local/bin:/usr/X11R6/bin
```

- You can use `which` to see if a command is available through your `PATH`:

```
[me@here ~]$ which bash
/bin/bash
```

Chaining Commands



- Whilst the commands we've looked at are useful on their own, they become Really Useful when chained together.
- Linux enables this chaining through I/O redirection.
- It provides three main I/O 'files' or streams:
  - stdin, standard input, usually the keyboard
  - stdout, standard output, usually the terminal
  - stderr, standard error, error messages output to the terminal
- Each open 'file' gets assigned a 'file descriptor':
  - stdin has FD 0
  - stdout has FD 1
  - stderr has FD 2.
- ***Yet again this is an area where bash and (t)csh are REALLY idiosyncratic...***

Basic I/O redirection

- Simplest case – we want to record the output of a command into a file:

```
[me@here ~]$ ls -l 1>output.txt
```

'Connect FD1(stdout) to output.txt'

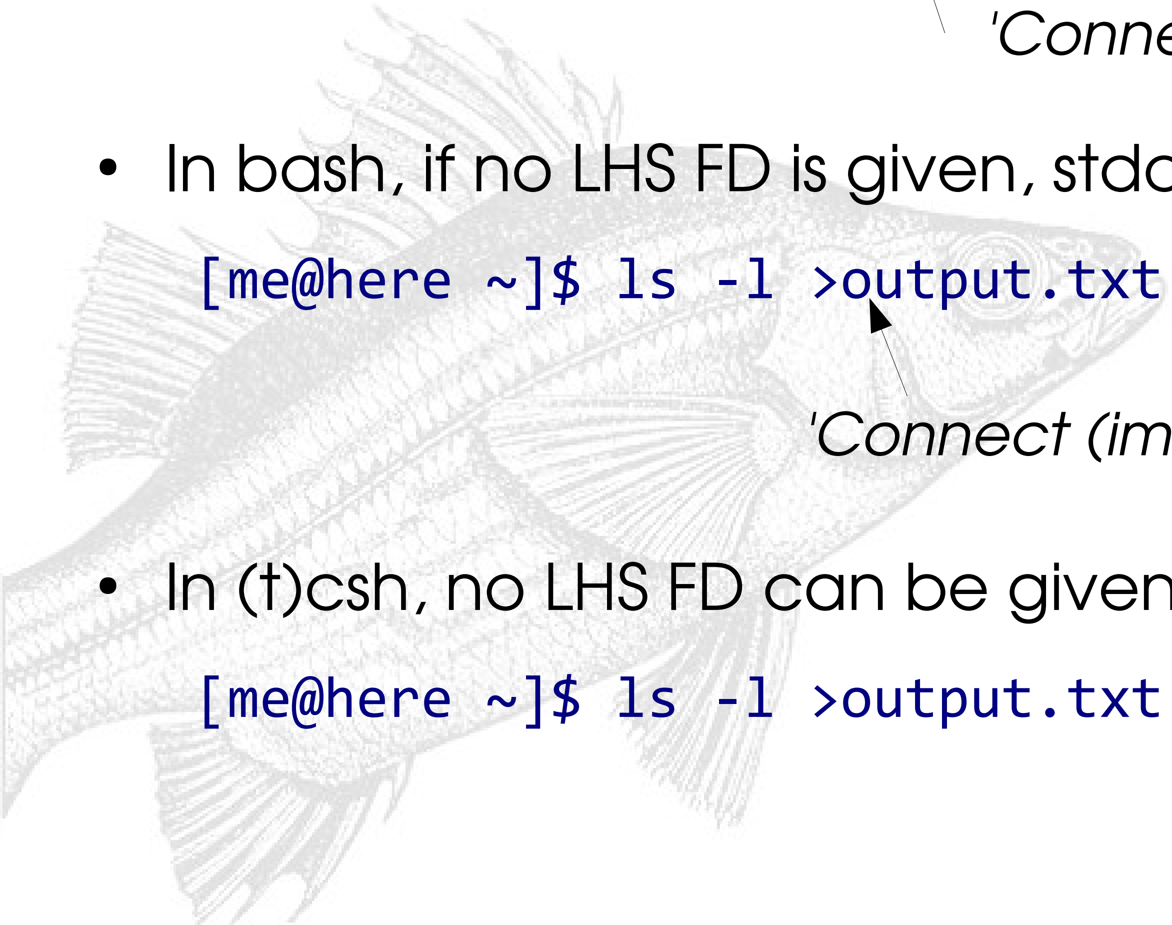
- In bash, if no LHS FD is given, stdout is assumed, so identically

```
[me@here ~]$ ls -l >output.txt
```

'Connect (implicit) FD1(stdout) to output.txt'

- In (t)csh, no LHS FD can be given, so we can only do

```
[me@here ~]$ ls -l >output.txt
```



Basic I/O redirection

- With >, if the file redirected to exists, it'll be overwritten
- If we want to append to the file instead, we do

```
[me@here ~]$ ls -l 1>>output.txt
```

'Connect and append stdout to output.txt'

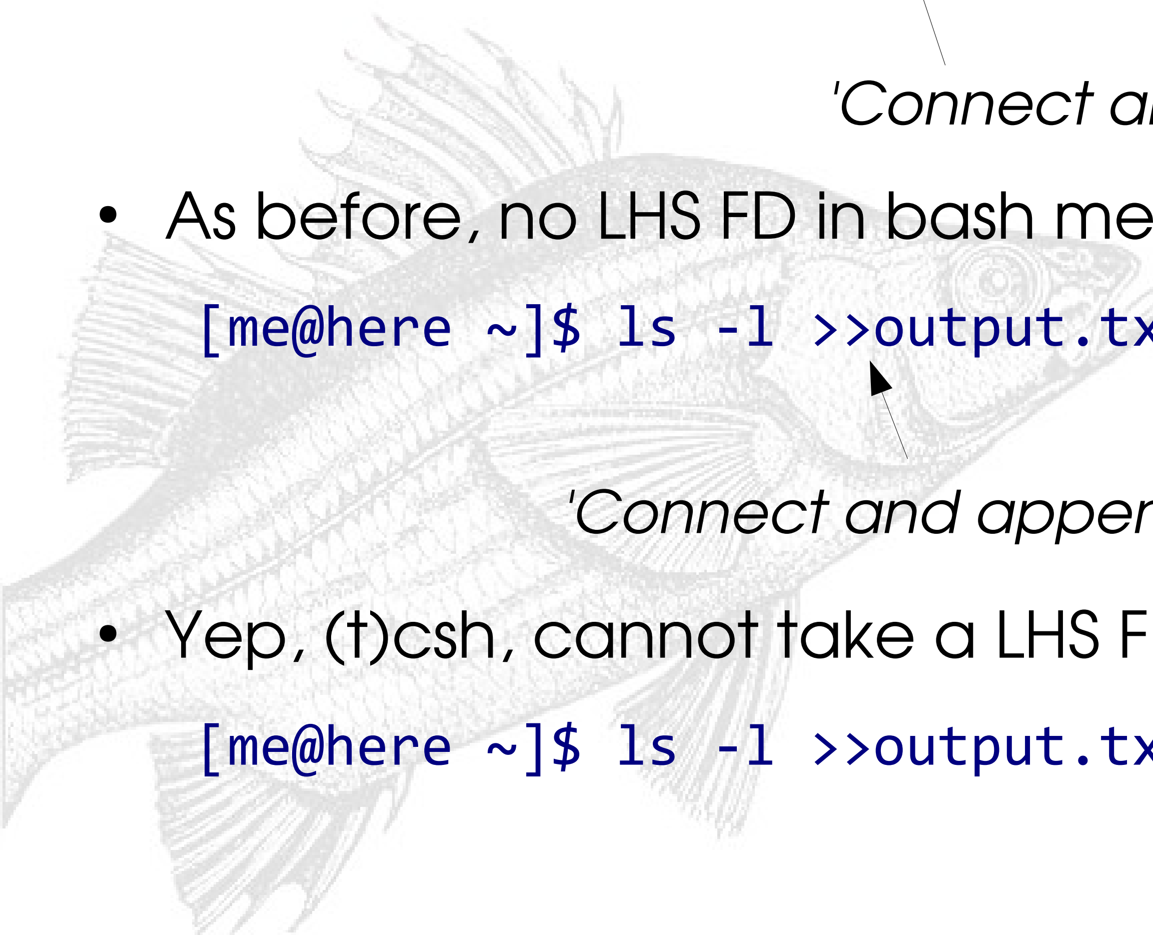
- As before, no LHS FD in bash means implicit stdout:

```
[me@here ~]$ ls -l >>output.txt
```

'Connect and append (implicit) stdout to output.txt'

- Yep, (t)csh, cannot take a LHS FD, so we can only do

```
[me@here ~]$ ls -l >>output.txt
```



Multiple Streams, One File

- Sometimes we want to redirect stdout (FD1) and stderr (FD2) to the same file (e.g. monitoring batch jobs, compilations)
- To do this in bash we use `>&FD`

```
[me@here ~]$ ls -l *.dat *.tex  1>all.txt 2>&1
```

'Connect stderr to where FD1 is pointing'

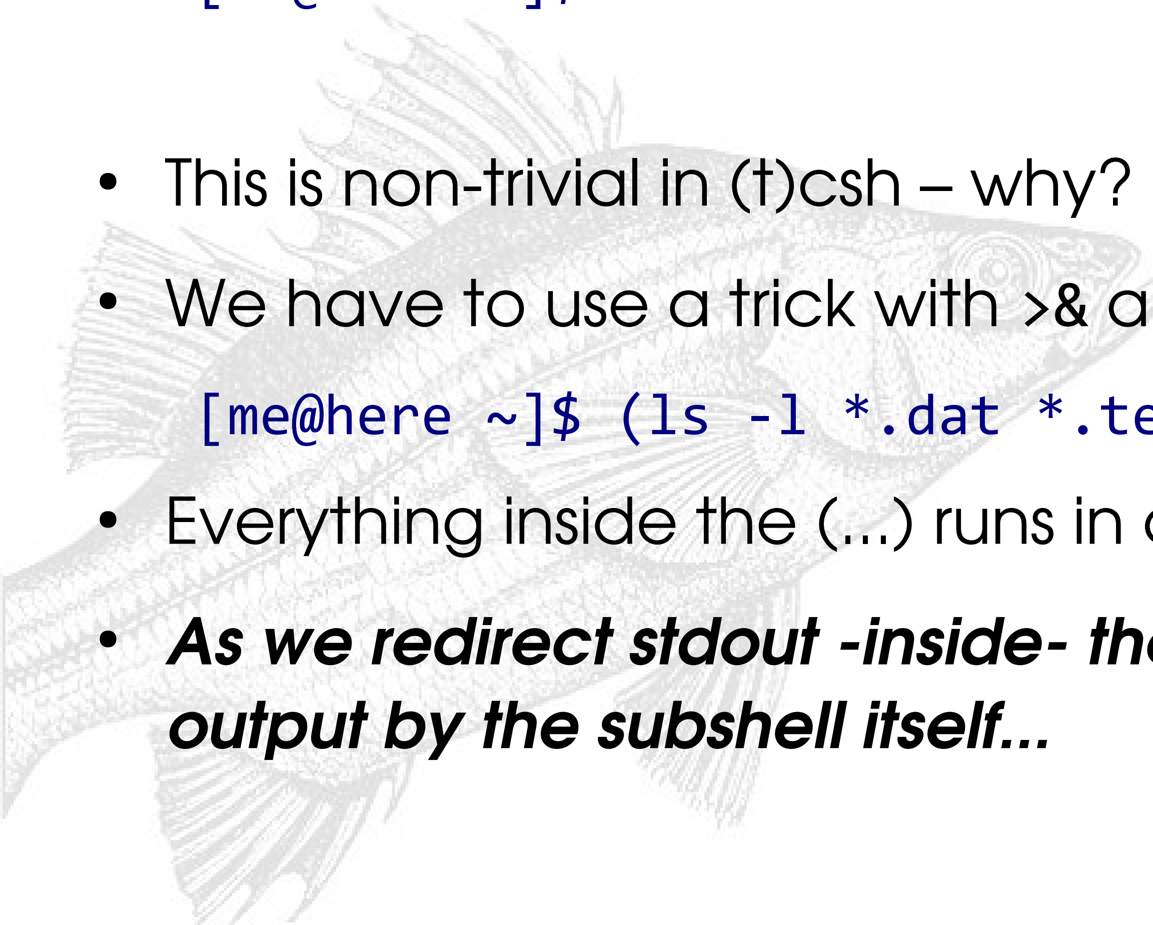
- Swapping order ***does not*** produce the same result!

```
[me@here ~]$ ls -l *.dat *.tex 2>&1 1>all.txt
```

- You can read this L to R as: *'FD2 pointed to where FD1 is pointing, then FD1 pointed away to all.txt'* : i.e. FD2 does not follow the redirection of FD1...
- As before, in (t)csh we just leave off the FD numbers

```
[me@here ~]$ ls -l *.dat *.tex >& all.txt
```

Output to Multiple Files



- It's often more useful to redirect stdout to one file and stderr to another (typical example is code compilation).
- To do this in bash we do

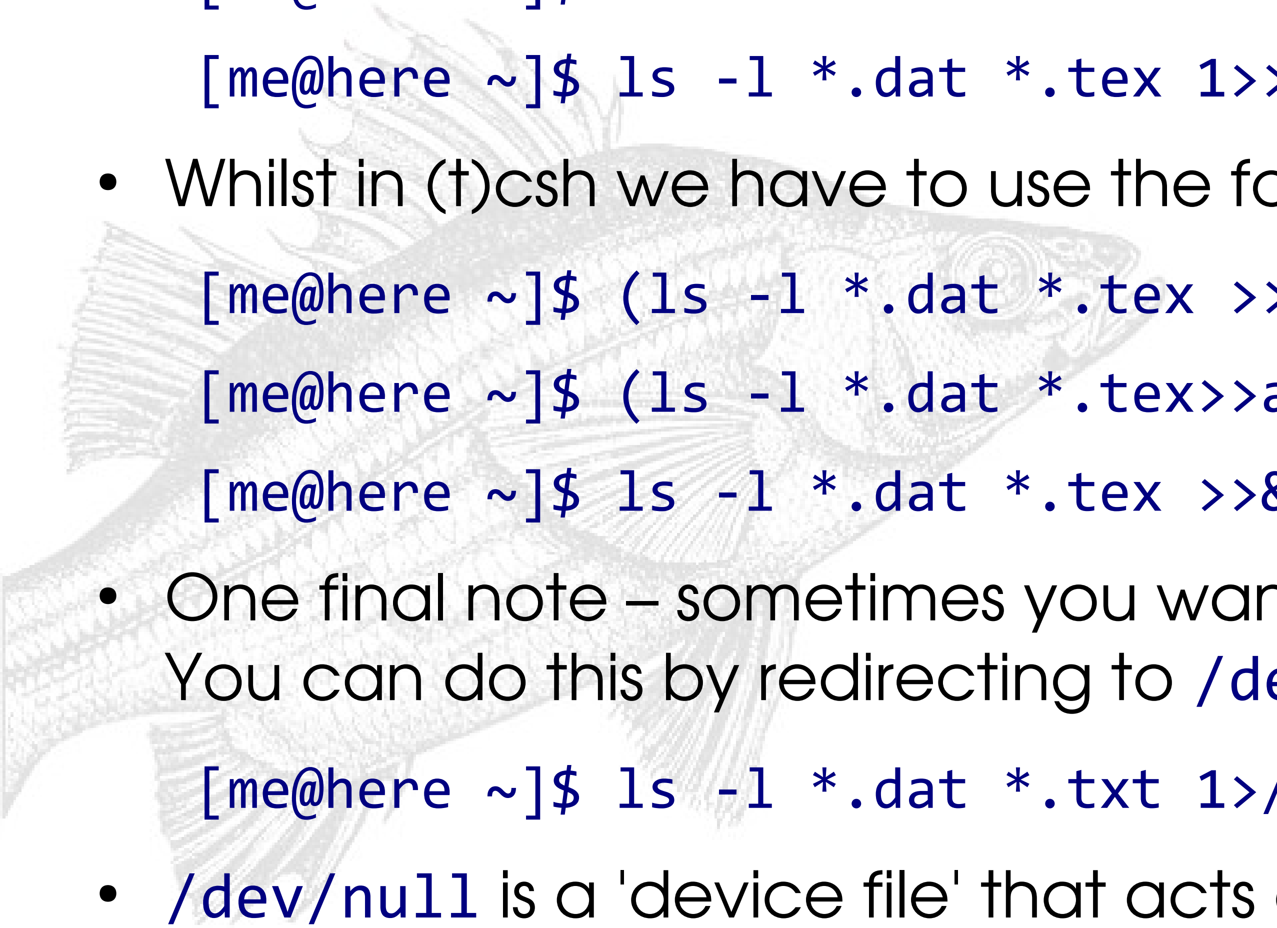
```
[me@here ~]$ ls -l *.dat *.tex 1>out.txt 2>err.txt
```

- This is non-trivial in (t)csh – why?
- We have to use a trick with >& and a ***subshell***

```
[me@here ~]$ (ls -l *.dat *.tex >all.txt) >& err.txt
```

- Everything inside the (...) runs in a separate shell
- ***As we redirect stdout -inside- the subshell, only stderr is output by the subshell itself...***

Multiple Streams: Append



- With multiple streams we can of course append to the resultant output files using >>
- To do this in bash we can, for example, do

```
[me@here ~]$ ls -l *.dat *.tex 1>out.txt 2>>err.txt
[me@here ~]$ ls -l *.dat *.tex 1>>allout.txt 2>&1
```

- Whilst in (t)csh we have to use the forms

```
[me@here ~]$ (ls -l *.dat *.tex >>all.txt) >& err.txt
[me@here ~]$ (ls -l *.dat *.tex>>all.txt) >>& err.txt
[me@here ~]$ ls -l *.dat *.tex >>& allout.txt
```

- One final note – sometimes you want to suppress ALL output. You can do this by redirecting to `/dev/null`

```
[me@here ~]$ ls -l *.dat *.txt 1>/dev/null 2>&1
```

- `/dev/null` is a 'device file' that acts as a 'black hole'...

Input Redirection

- Input redirection works in very similar way to output:

```
[me@here ~]$ program 0<input.txt
```

'Connect stdin to file input.txt'

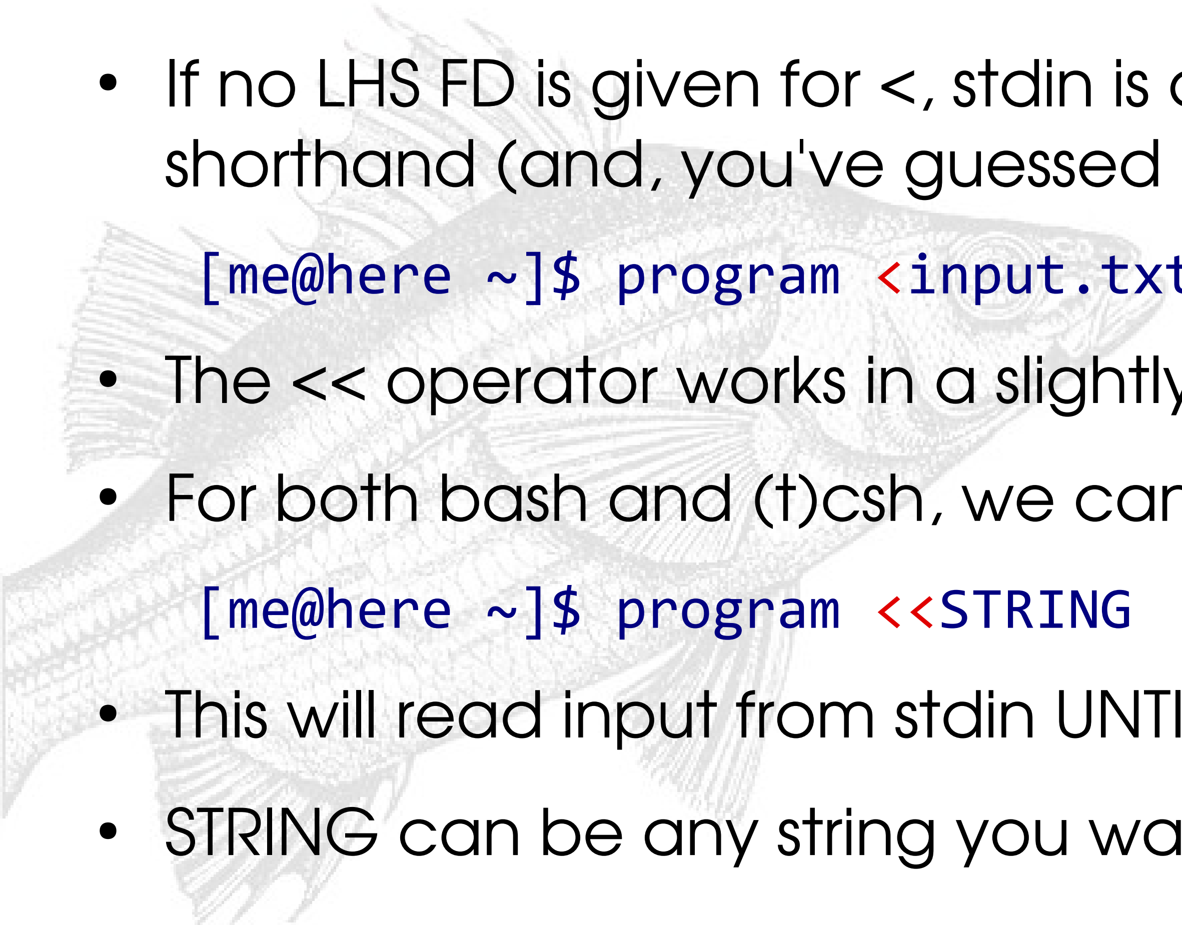
- If no LHS FD is given for <, stdin is assumed, giving the shorthand (and, you've guessed it, the only way in (t)csh):

```
[me@here ~]$ program <input.txt
```

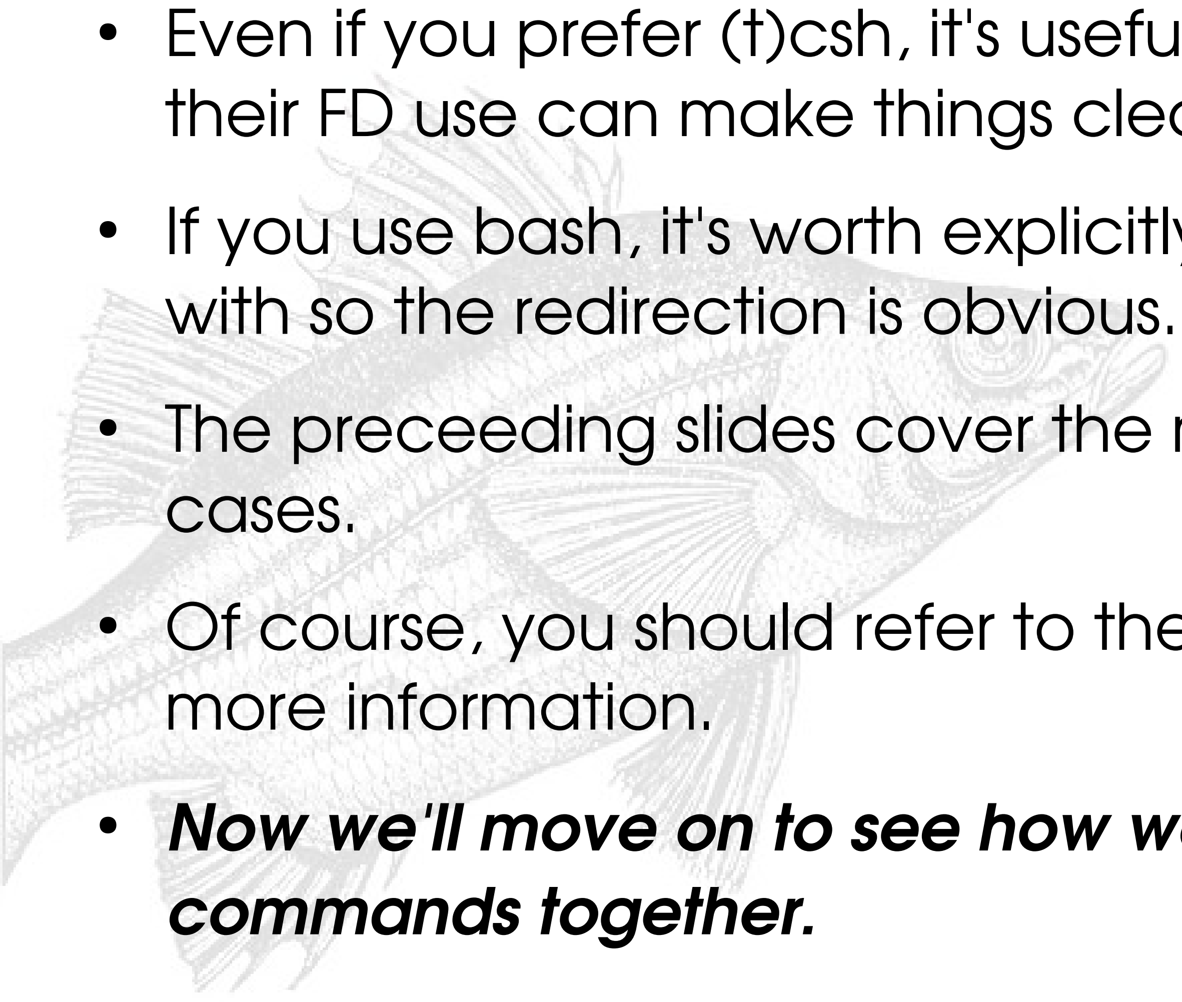
- The << operator works in a slightly different way.
- For both bash and (t)csh, we can write

```
[me@here ~]$ program <<STRING
```

- This will read input from stdin UNTIL it reads STRING.
- STRING can be any string you want, e.g. END, EOF etc.



Combining Input/Output



- Input and output redirection can be easily combined, for instance:

```
[me@here ~]$ program 0<input.txt 1>output.txt 2>&1
```

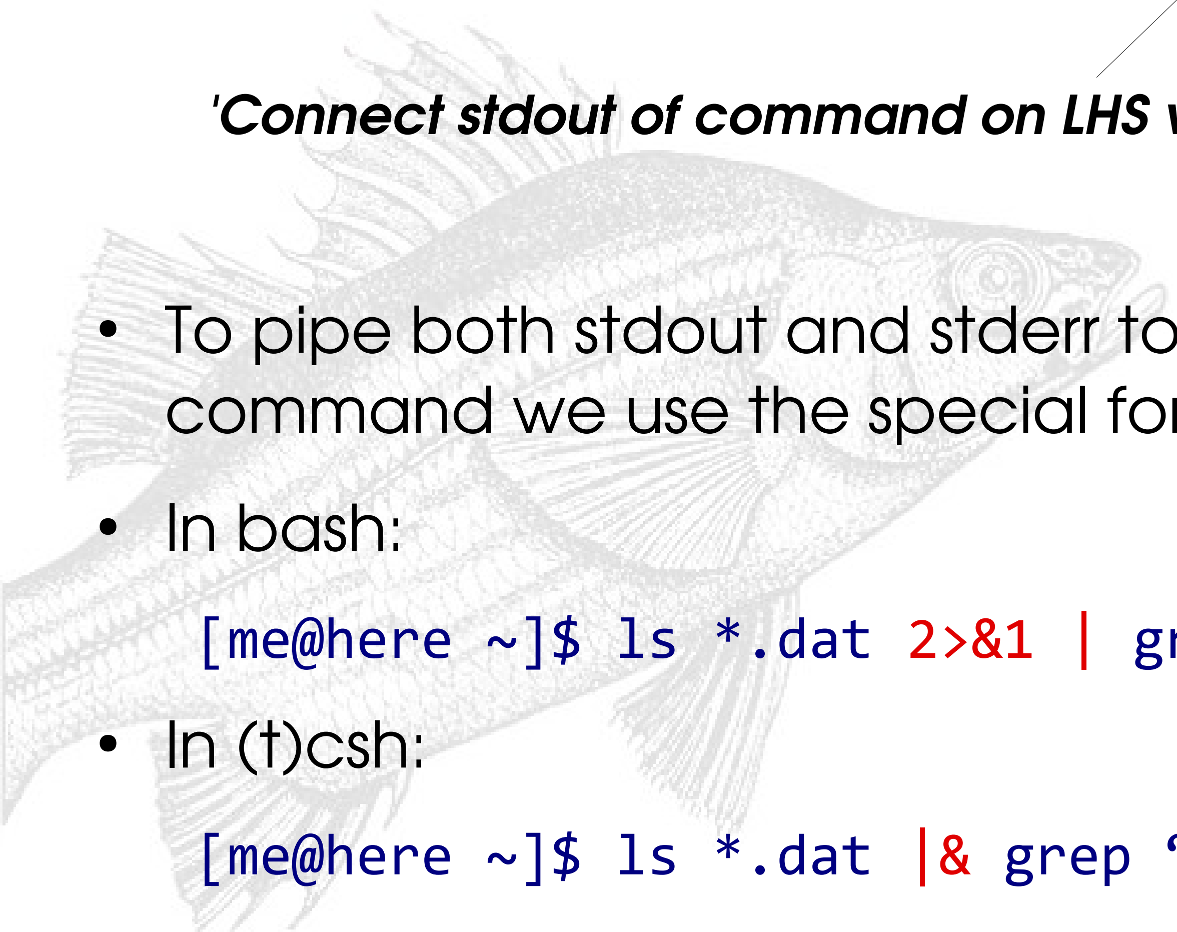
- Even if you prefer (t)csh, it's useful to look at the bash forms as their FD use can make things clearer.
- If you use bash, it's worth explicitly writing the Fds in to start with so the redirection is obvious.
- The preceeding slides cover the majority of common use cases.
- Of course, you should refer to the shell documentation for more information.
- ***Now we'll move on to see how we use I/O to chain commands together.***

Pipes

- We can connect the stdout of one program to the stdin of another through ***pipes***:

```
[me@here ~]$ cat particles_a.dat | grep “e-”
```

'Connect stdout of command on LHS with stdin of command on RHS'



- To pipe both stdout and stderr to the subsequent chained command we use the special forms:
- In bash:

```
[me@here ~]$ ls *.dat 2>&1 | grep “particles”
```

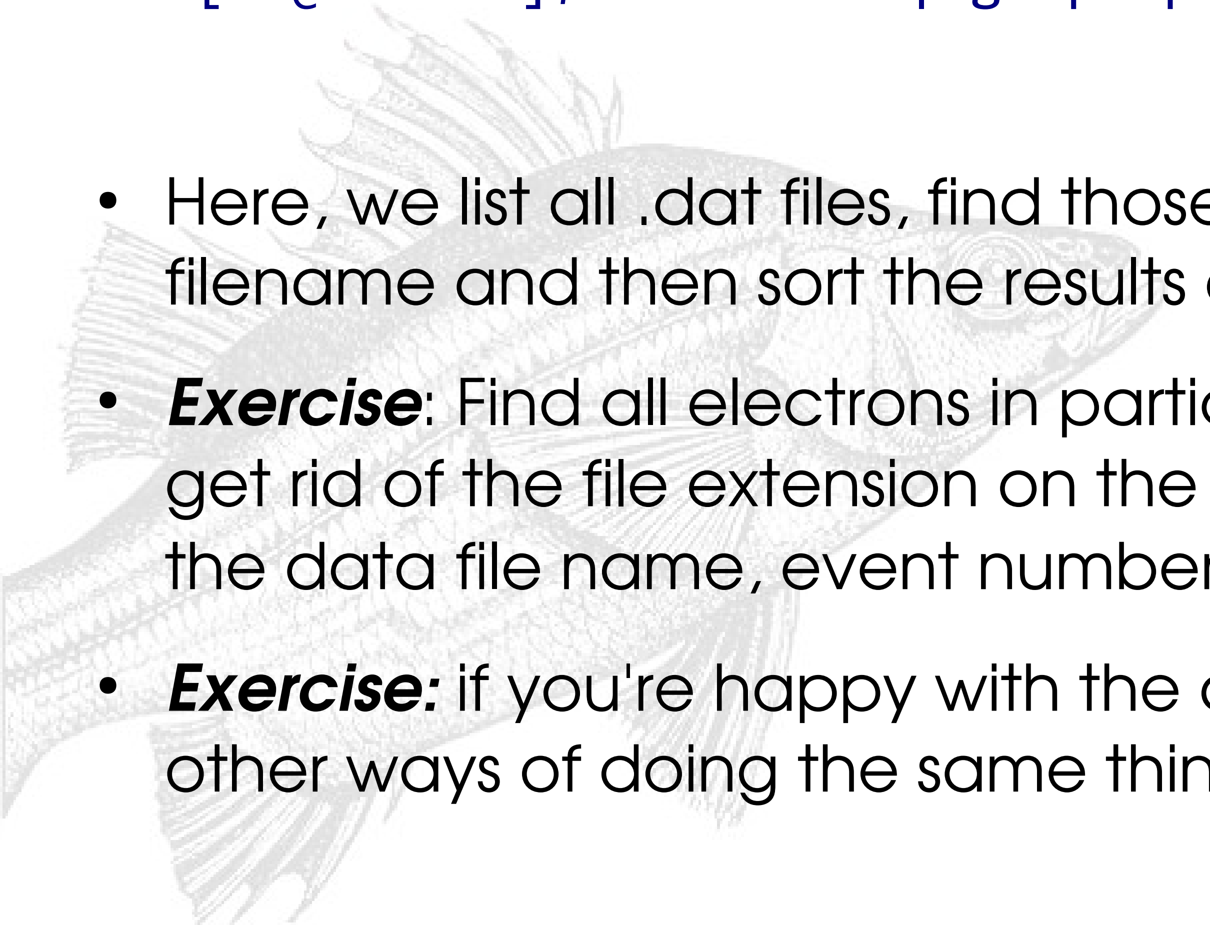
- In (t)csh:

```
[me@here ~]$ ls *.dat |& grep “particles”
```

Multiple Pipes

- Pipes are extremely powerful as they allow us to chain several commands together:

```
[me@here ~]$ ls *.dat | grep “particles” | sort -d
```



- Here, we list all .dat files, find those with “particles” in the filename and then sort the results alphabetically.
- ***Exercise***: Find all electrons in particles_a.dat, sort on p_t, get rid of the file extension on the data file name, print out the data file name, event number and p_t ***in that order***.
- ***Exercise:*** if you're happy with the above, can you find other ways of doing the same thing?

Combined Pipes and I/O

- Naturally, we can combine pipes with I/O redirection

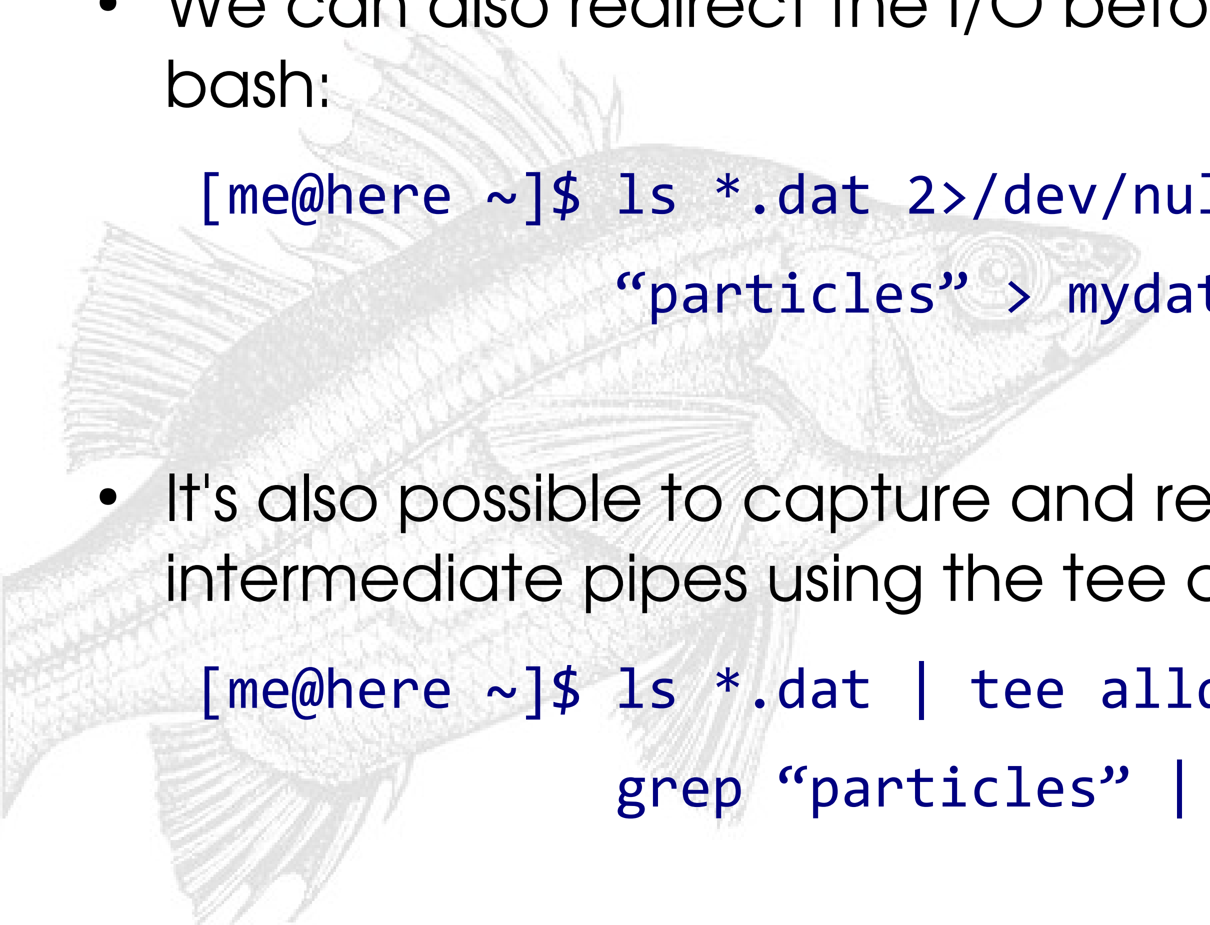
```
[me@here ~]$ ls *.dat | grep “particles” > out.txt
```

- We can also redirect the I/O before the pipe, at least in bash:

```
[me@here ~]$ ls *.dat 2>/dev/null | grep \
             “particles” > mydata.txt
```

- It's also possible to capture and redirect stdout in intermediate pipes using the tee command

```
[me@here ~]$ ls *.dat | tee alldata.txt | \
             grep “particles” | sort -d > mydata.txt
```

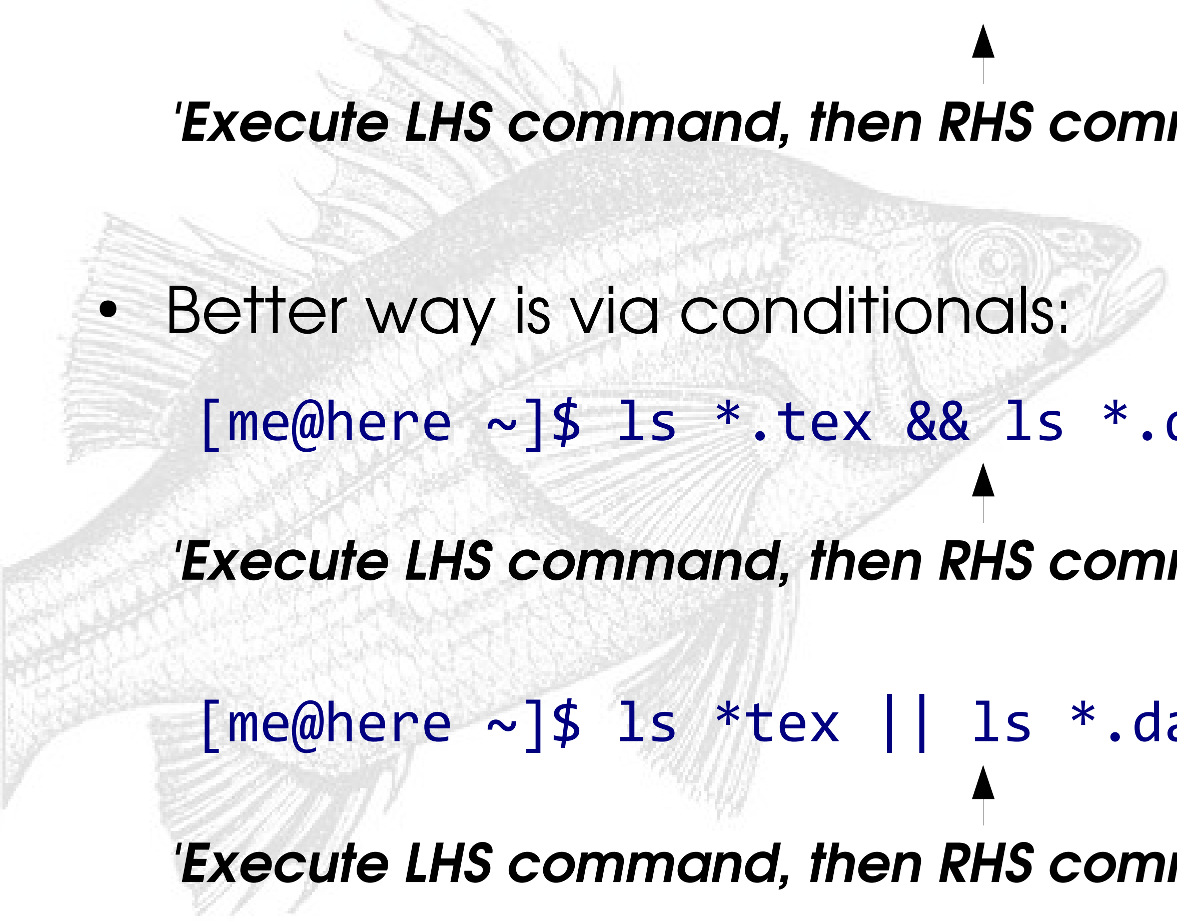


Linking Commands

- Commands can be linked to be executed in sequence based on the success or failure of previous commands
- Simplest link is via ;

```
[me@here ~]$ ls *.tex ; ls *.dat
```

'Execute LHS command, then RHS command, no matter result of LHS'



- Better way is via conditionals:

```
[me@here ~]$ ls *.tex && ls *.dat
```

'Execute LHS command, then RHS command IF LHS was successful'

```
[me@here ~]$ ls *tex || ls *.dat
```

'Execute LHS command, then RHS command IF LHS was unsuccessful'

Further Resources

- We've already covered quite a lot.
- If you need further information, there's lots more in the man pages for bash and (t)csh (and don't forget Google!).
- O'Reilly also have a couple of really good textbooks: