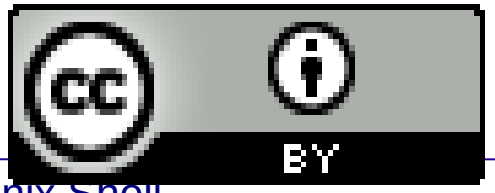
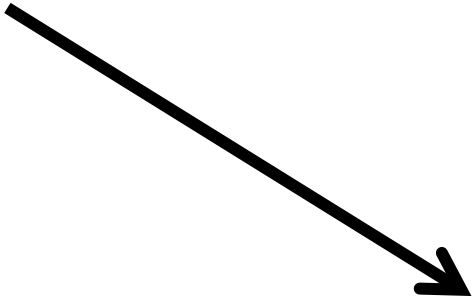
The Unix Shell

Advanced Shell Tricks



“How should I do this?”



"How should I do this?"

With smartphones, you'll often hear people say something like

"There's an app for that… check this out!"

"How should I do this?"

With smartphones, you'll often hear people say something like

"There's an app for that… check this out!"

Whereas Unix shell programmers will say

"There's a shell trick for that… check this out!"

In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters

```
$ wc –l *.pdb | sort | head -1
```

In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files

```
$ wc –l *.pdb > lengths
```

In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

```
$ SECRET_IDENTITY=Dracula
$ echo $SECRET_IDENTITY
Dracula
```

In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

Very powerful when used together

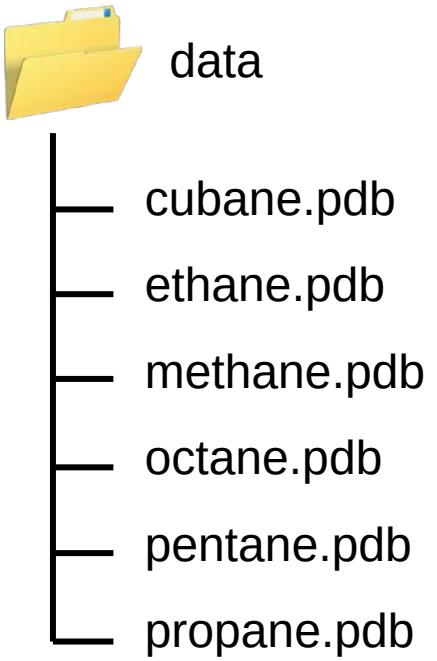
In previous episodes, we've seen how to:

– Combine existing programs using pipes & filters
– Redirect output from programs to files
– Use variables to control program operation

Very powerful when used together

But there are other useful things we can do with these – let's take a look…

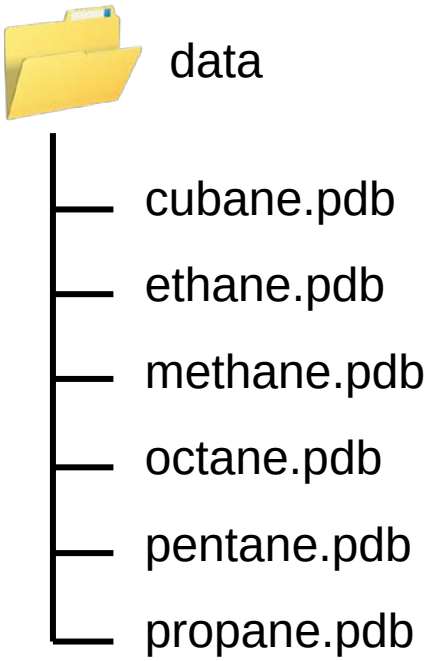
First, let's revisit redirection…



First, let's revisit redirection…

```
$ ls *.pdb > files
```

← list all pdb files
redirect to a file



First, let's revisit redirection…

```
$ ls *.pdb > files
```

← list all pdb files
redirect to a file

The 'redirection' operator

data

- cubane.pdb
- ethane.pdb
- methane.pdb
- octane.pdb
- pentane.pdb
- propane.pdb

First, let's revisit redirection…

```
$ ls *.pdb > files
```

← list all pdb files
redirect to a file

But what about adding this together with other results generated later?

data

- cubane.pdb
- ethane.pdb
- methane.pdb
- octane.pdb
- pentane.pdb
- propane.pdb
- butane.ent
- heptane.ent
- hexane.ent
- nonane.ent
- decane.ent

First, let's revisit redirection…

$ ls *.pdb > files ← list all pdb files redirect to a file

But what about adding this together with other results generated later?

$ ls *.ent > more-files

data

- cubane.pdb
- ethane.pdb
- methane.pdb
- octane.pdb
- pentane.pdb
- propane.pdb
- butane.ent
- heptane.ent
- hexane.ent
- nonane.ent
- decane.ent

First, let's revisit redirection…

```
$ ls *.pdb > files
```

← list all pdb files
redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
```

We just want the ent files

data

- cubane.pdb
- ethane.pdb
- methane.pdb
- octane.pdb
- pentane.pdb
- propane.pdb
- butane.ent
- heptane.ent
- hexane.ent
- nonane.ent
- decane.ent

First, let's revisit redirection…

```
$ ls *.pdb > files
```

← list all pdb files redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

← append files into a single new file

data

- cubane.pdb
- ethane.pdb
- methane.pdb
- octane.pdb
- pentane.pdb
- propane.pdb
- butane.ent
- heptane.ent
- hexane.ent
- nonane.ent
- decane.ent

First, let's revisit redirection…

```
$ ls *.pdb > files
```

← list all pdb files redirect to a file

But what about adding this together with other results generated later?

```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

← append files into a single new file

Instead, we can do…

```
$ ls *.ent >> files
```

data

- cubane.pdb
- ethane.pdb
- methane.pdb
- octane.pdb
- pentane.pdb
- propane.pdb
- butane.ent
- heptane.ent
- hexane.ent
- nonane.ent
- decane.ent

First, let's revisit redirection…

```
$ ls *.pdb > files
```

← list all pdb files redirect to a file

But what about adding this together with other results generated later?

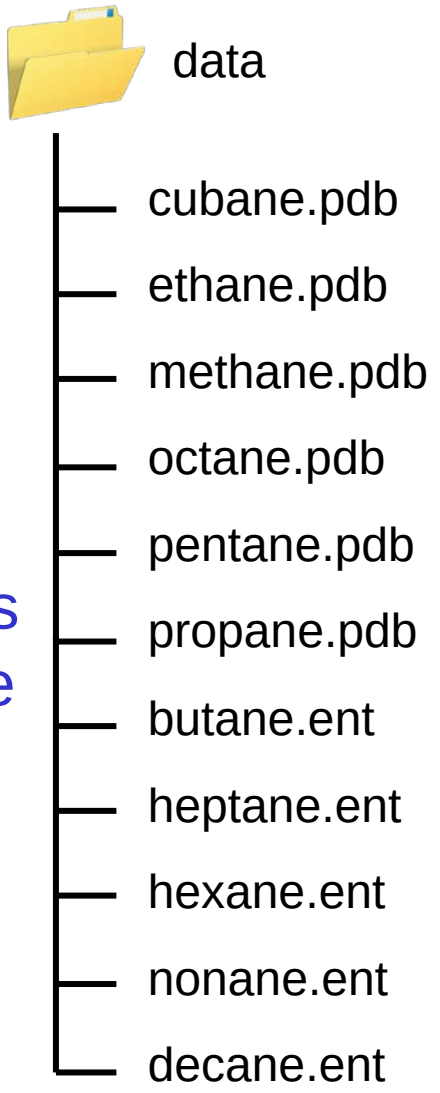
```
$ ls *.ent > more-files
$ cat files more-files > all-files
```

← append files into a single new file

Instead, we can do…

```
$ ls *.ent >> files
```

Note the double >'s – the append' operator

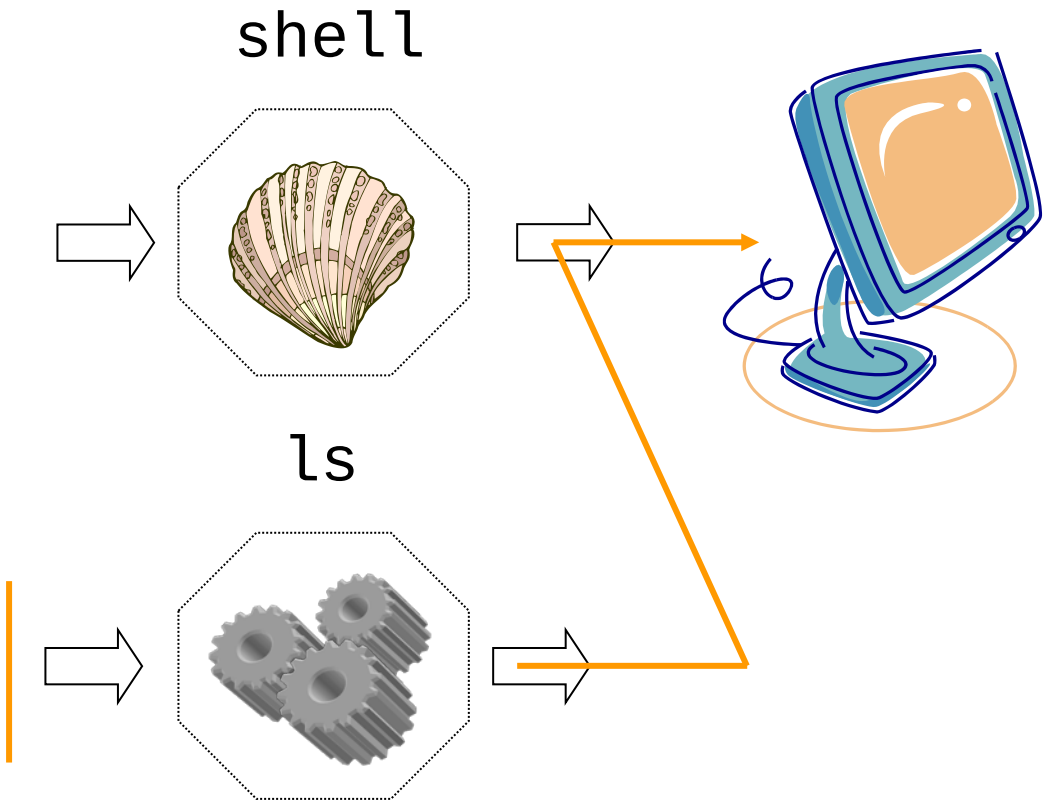


We know that…

Normally, standard output is directed to a display:

We know that…

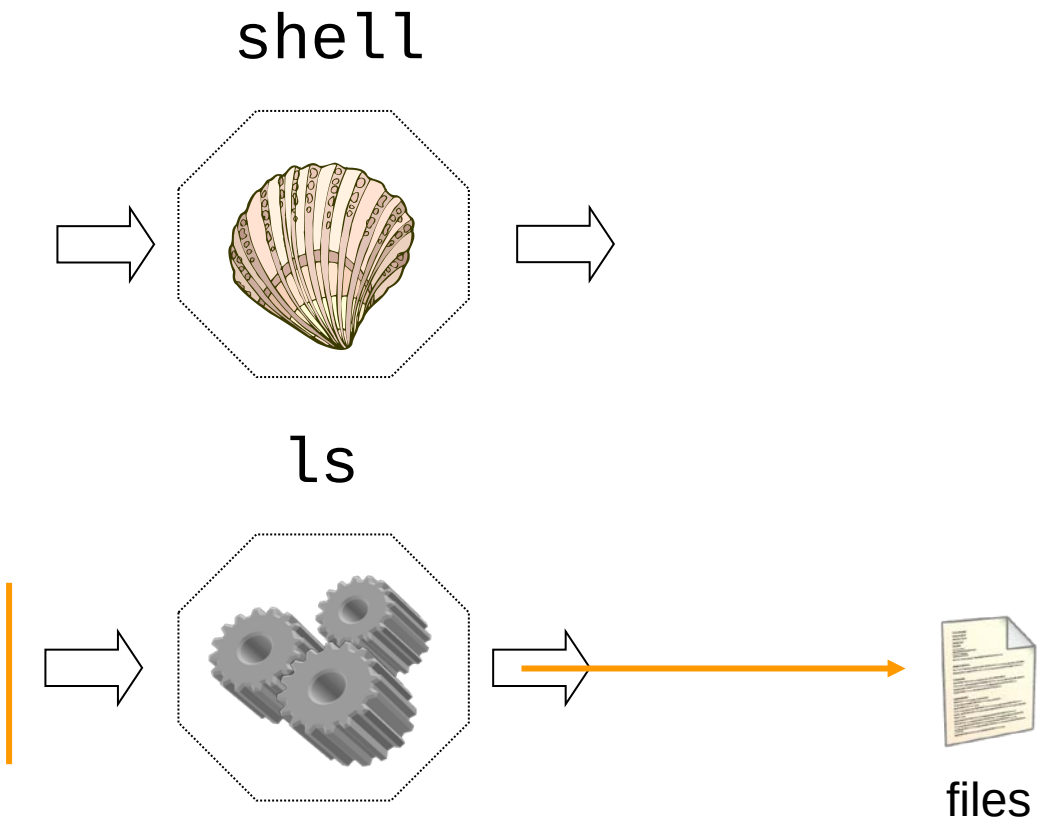
Normally, standard output is directed to a display:



We know that…

Normally, standard output is directed to a display:

But we have redirected it to a file instead:



But what happens with error messages?

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or directory
```

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or directory
```

No files are listed in *files*, as you might expect.

But what happens with error messages?

For example…

```
$ ls /some/nonexistent/path > files
ls: /some/nonexistent/path: No such file or directory
```

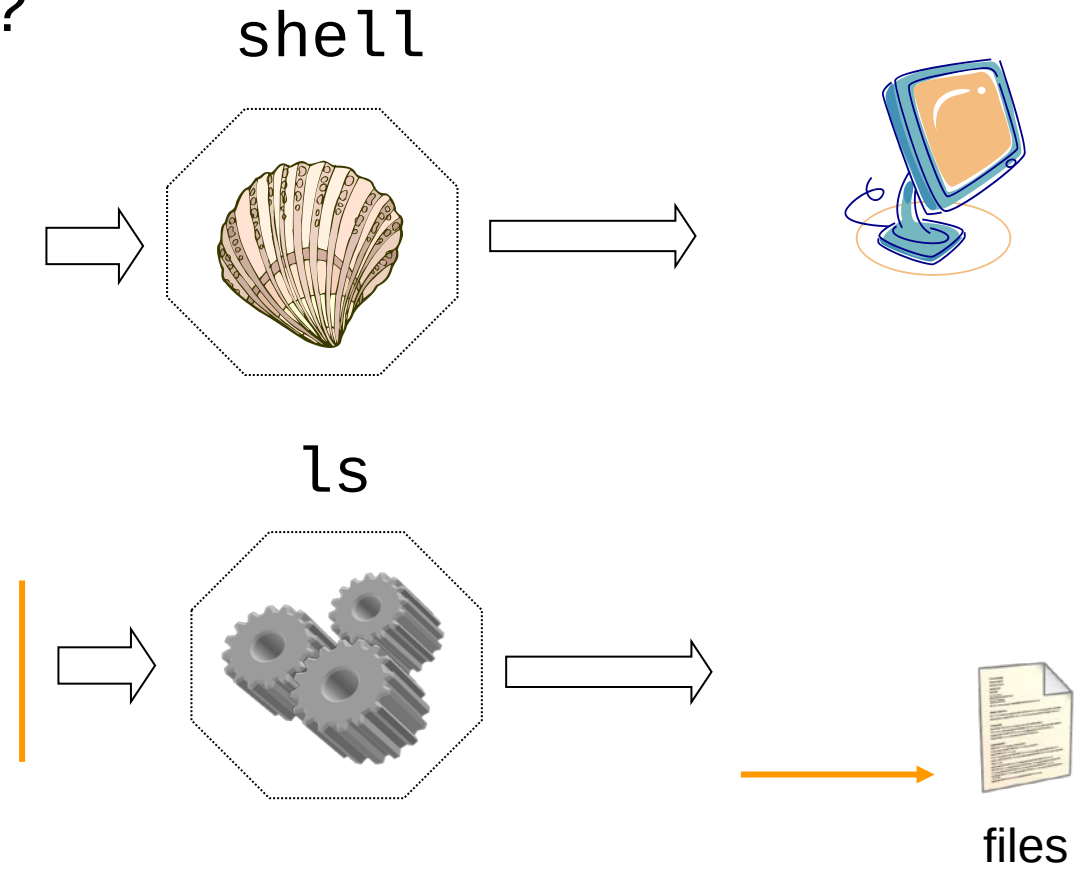
No files are listed in *files*, as you might expect.

But why isn't the error message in *files*?

This is because error messages are sent to the *standard error* (stderr), separate to stdout

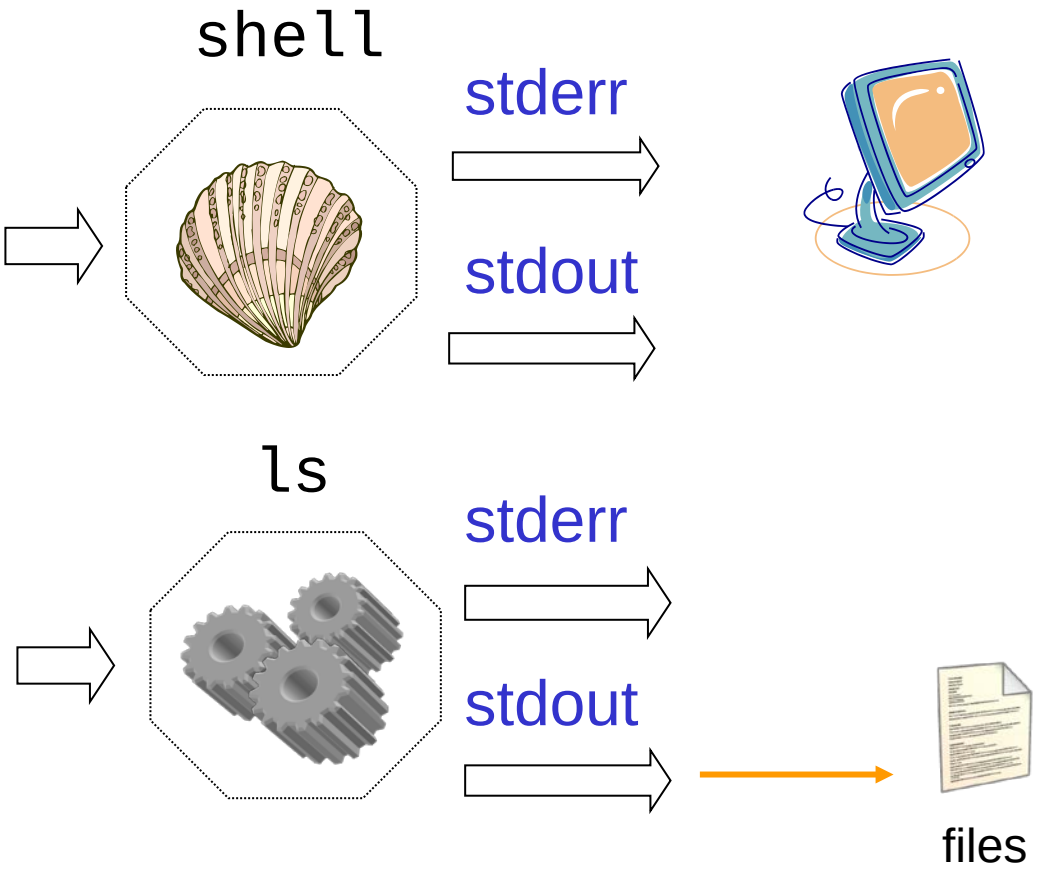
This is because error messages are sent to the *standard error* (stderr), separate to stdout

So what was happening with the previous example?



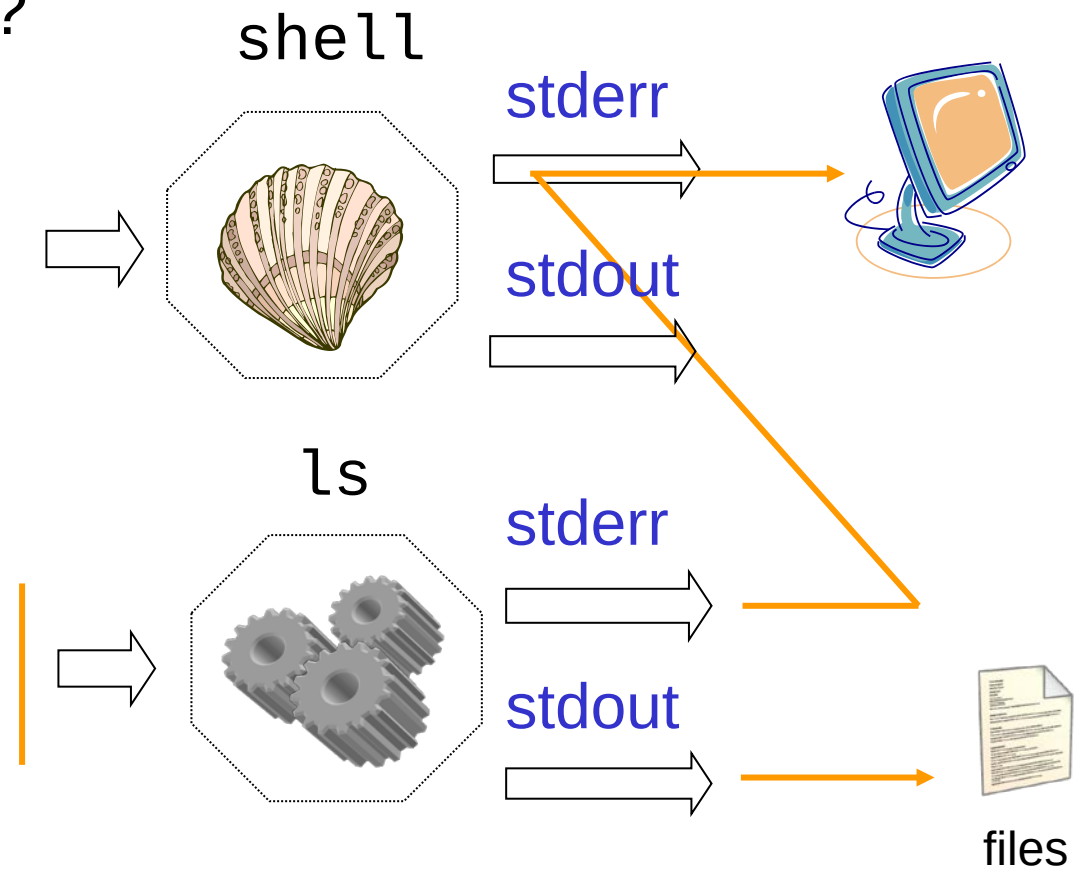
This is because error messages are sent to the *standard error* (stderr), separate to stdout

So what was happening with the previous example?



This is because error messages are sent to the *standard error* (stderr), separate to stdout

So what was happening with the previous example?



We can capture standard error as well as standard output

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Redirect as before, but with a slightly different operator

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2> error-log
```

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2> error-log
```

We can use both stdout and stderr redirection – at the same time

We can capture standard error as well as standard output

To redirect the standard error to a file, we can do:

```
$ ls /some/nonexistent/path 2> error-log
```

Now we have any error messages stored in *error-log*

To redirect both stdout and stderr, we can then do:

```
$ ls /usr /some/nonexistent/path > files 2> error-log
```

Which would give us contents of */usr* in *files* as well.

So why a ‘2’ before the ‘>’ ?

So why a ‘2’ before the ‘>’ ?

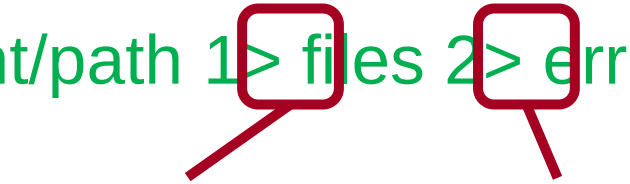
Both stdout and stderr can be referenced by numbers:

```
$ ls /usr /some/nonexistent/path 1> files 2> error-log
```

So why a ‘2’ before the ‘>’ ?

Both stdout and stderr can be referenced by numbers:

$ ls /usr /some/nonexistent/path 1> files 2> error-log



Refers to stdout

Refers to stderr

So why a ‘2’ before the ‘>’ ?

Both stdout and stderr can be referenced by numbers:

$ ls /usr /some/nonexistent/path 1> files 2> error-log

To just redirect both to the same file we can also do:

$ ls /usr /some/nonexistent/path &> everything

With ‘&’ denoting both stdout and stderr

So why a ‘2’ before the ‘>’ ?

Both stdout and stderr can be referenced by numbers:

$ ls /usr /some/nonexistent/path 1> files 2> error-log

To just redirect both to the same file we can also do:

$ ls /usr /some/nonexistent/path &> everything

With ‘&’ denoting both stdout and stderr

We can also use append for each of these too:

$ ls /usr /some/nonexistent/path 1>> files 2>> error-log

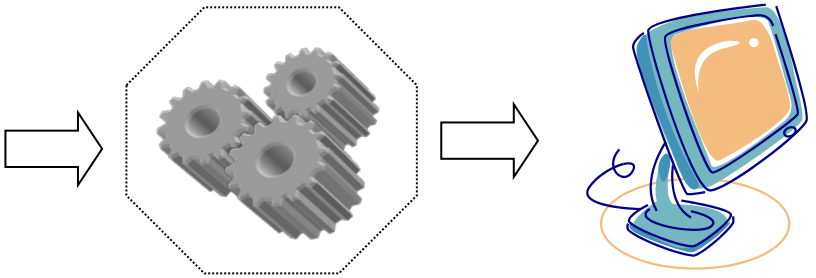
>	1>	Redirect stdout to a file
	2>	Redirect stderr to a file
	&>	Redirect both stdout and stderr to the same file

>	1>	Redirect stdout to a file
	2>	Redirect stderr to a file
	&>	Redirect both stdout and stderr to the same file
>>	1>>	Redirect and append stdout to a file
	2>>	Redirect and append stderr to a file
	&>>	Redirect and append both stdout and stderr to a file

We've seen how pipes and filters work with using a single program on some input data…

We've seen how pipes and filters work with using a single program on some input data…

```
a_program 1 2 3
```

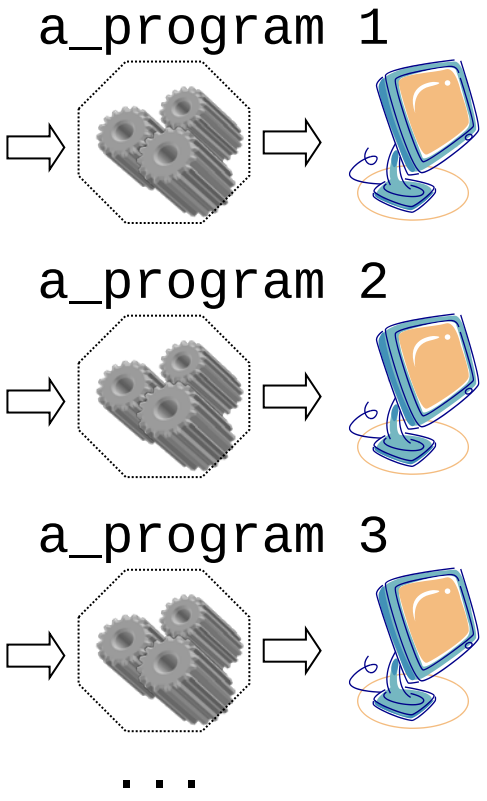


We've seen how pipes and filters work with using a single program on some input data…

But what about running the same program *separately*, for each input?

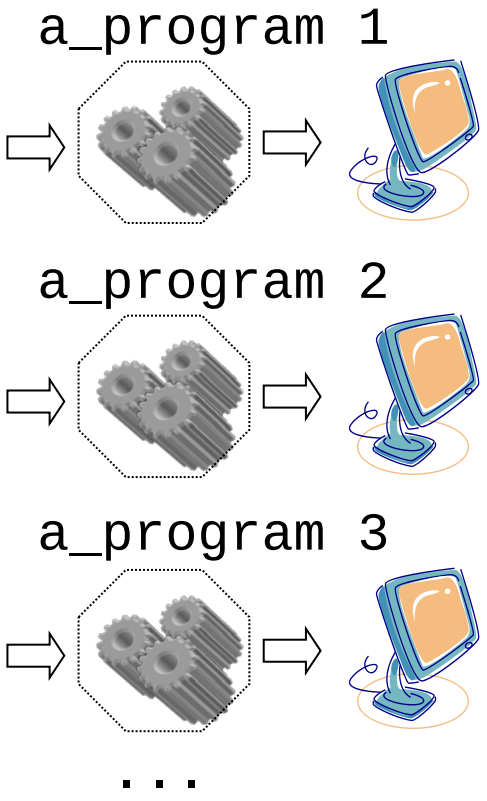
We've seen how pipes and filters work with using a single program on some input data…

But what about running the same program *separately*, for each input?



We've seen how pipes and filters work with using a single program on some input data…

But what about running the same program *separately*, for each input?

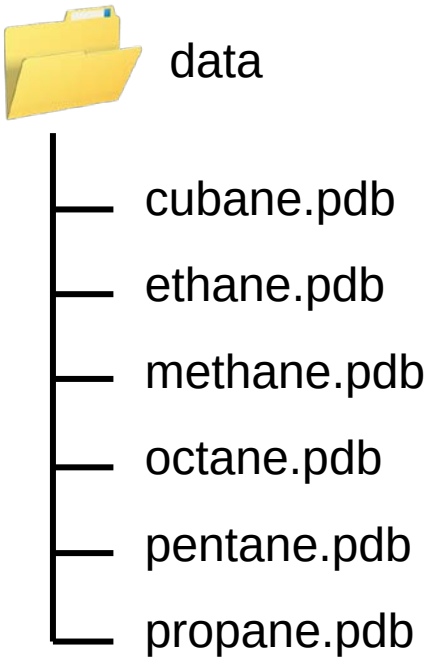


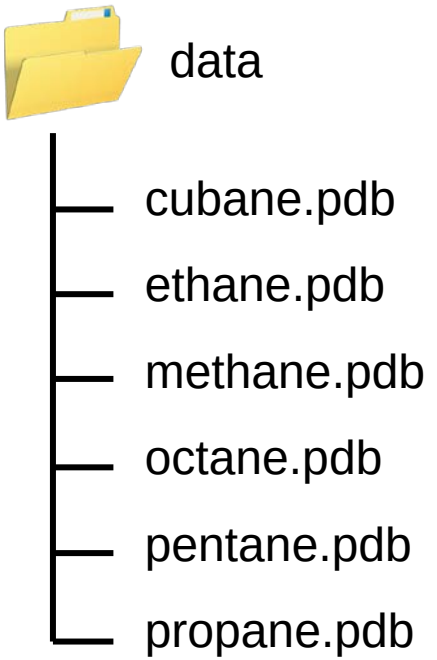
We can use *loops* for this…

So what can we do with loops?

So what can we do with loops?

Let’s go back to our first set of pdb files, and assume we want to compress each of them



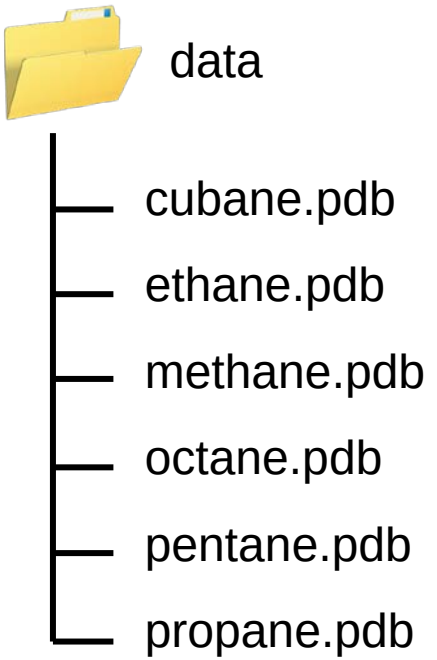


So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```



So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

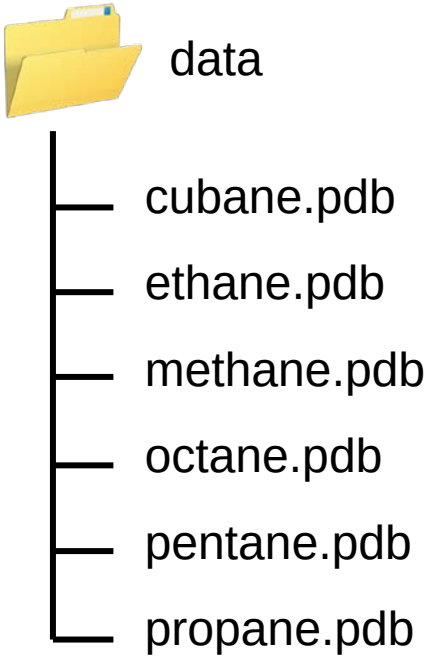
```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

← typical output from the zip command

So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

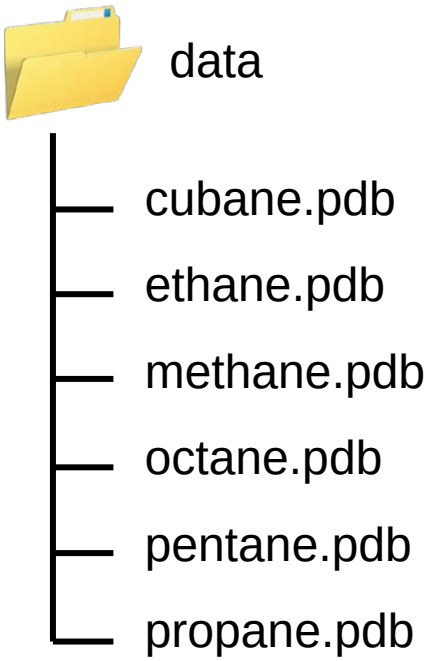
We could do the following for each:




```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

← typical output from the zip command

The zip file we wish to create



So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

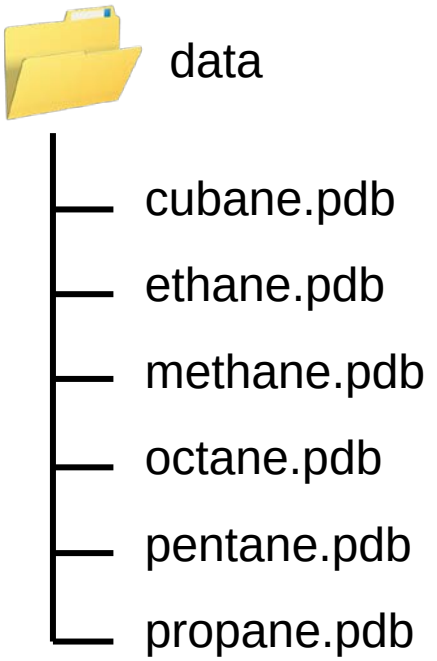
We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

← typical output from the zip command

The zip file we wish to create

The file(s) we wish to add to the zip file



So what can we do with loops?

Let's go back to our first set of pdb files, and assume we want to compress each of them

We could do the following for each:

```
$ zip cubane.pdb.zip cubane.pdb
  adding: cubane.pdb (deflated 73%)
```

Not efficient for many files

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

For each pdb file in this directory…

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

Run this command

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

This is the end of the loop

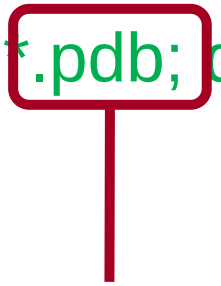
Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

The semicolons separate each part of the loop construct

Using a loop, we can iterate over each file, and run *zip* on each of them:

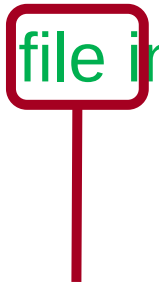
```
$ for file in *.pdb; do zip $file.zip $file; done
```



This expands to a list of every pdb file

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```



This variable holds the next pdb file in the list

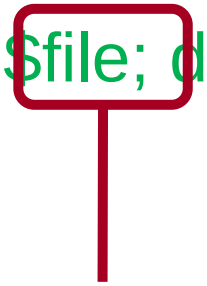
Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```

We reference the 'file' variable, and use '.' to add the zip extension to the filename

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
```



We reference the 'file' variable again

Using a loop, we can iterate over each file, and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
  adding: methane.pdb (deflated 66%)
  adding: octane.pdb (deflated 75%)
  adding: pentane.pdb (deflated 74%)
  adding: propane.pdb (deflated 71%)
```

Using a loop, we can iterate over each file,
and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
...
```

In one line, we've ended up with all files zipped

Using a loop, we can iterate over each file,
and run *zip* on each of them:

```
$ for file in *.pdb; do zip $file.zip $file; done
  adding: cubane.pdb (deflated 73%)
  adding: ethane.pdb (deflated 70%)
...
```

In one line, we've ended up with all files zipped

```
$ ls *.zip
cubane.pdb.zip    methane.pdb.zip   pentane.pdb.zip
ethane.pdb.zip    octane.pdb.zip    propane.pdb.zip
```

Now instead, what if we wanted to output the first line of each pdb file?

Now instead, what if we wanted to output the first line of each pdb file?

We could use head -1 *.pdb for that, but it would produce:

```
==> cubane.pdb <==
COMPND    CUBANE

==> ethane.pdb <==
COMPND    ETHANE

==> methane.pdb <==
COMPND    METHANE
...
```

Now instead, what if we wanted to output the first line of each pdb file?

We could use head -1 *.pdb for that, but it would produce:

head produces this
(it's not in the file)

```
==> cubane.pdb <==
COMPND    CUBANE

==> ethane.pdb <==
COMPND    ETHANE

==> methane.pdb <==
COMPND    METHANE
...
```

Now instead, what if we wanted to output the first line of each pdb file?

We could use head -1 *.pdb for that, but it would produce:

head produces this (it's not in the file)

this is actually the first line in this file!

```
==> cubane.pdb <==
COMPND    CUBANE

==> ethane.pdb <==
COMPND    ETHANE

==> methane.pdb <==
COMPND    METHANE
...
```

Now instead, what if we wanted to output the first line of each pdb file?

We could use head -1 *.pdb for that, but it would produce:

head produces this (it's not in the file)

this is actually the first line in this file!

```
==> cubane.pdb <==
COMPND    CUBANE

==> ethane.pdb <==
COMPND    ETHANE

==> methane.pdb <==
COMPND    METHANE
...
```

Perhaps we only want the actual first lines…

However, using a loop:

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
```

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
```

We use $file as we did before, but this time with the head command

However, using a loop:

```
$ for file in *.pdb; do head -1 $file; done
COMPND    CUBANE
COMPND    ETHANE
COMPND    METHANE
COMPND    OCTANE
COMPND    PENTANE
COMPND    PROPANE
```

What if we wanted this list sorted in reverse afterwards?

What if we wanted this list sorted in reverse afterwards?

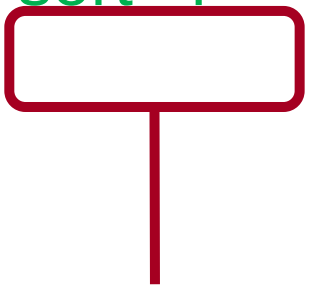
Simple!

```
$ (for file in ls *.pdb; do head -1 $file; done) | sort –r
```

What if we wanted this list sorted in reverse afterwards?

Simple!

```
$ (for file in ls *.pdb; do head -1 $file; done) | sort –r
```



Using a pipe, we can just add this on the end

What if we wanted this list sorted in reverse afterwards?

Simple!

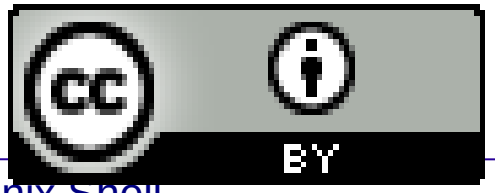
```
$ (for file in ls *.pdb; do head -1 $file; done) | sort –r
COMPND    PROPANE
COMPND    PENTANE
COMPND    OCTANE
COMPND    METHANE
COMPND    ETHANE
COMPND    CUBANE
```

`zip`	Create a compressed zip file with other files in it
`for …; do … done;`	Loop over a list of data and run a command once for each element in the list

created by

Steve Crouch

July 2011



More Tricks

From Alan Iwi at CEDA

Operations on multiple files: xargs

- This does not work

```
$ find acsoe | ls
acsoe     presentations
$
```

- Find pipes a list of files to ls.
- ls ignores input and just does a normal listing of the current working directory.

- Lots of commands expect a list of arguments not input. Is there anything to help?

Operations on multiple files: xargs

- The “xargs” command runs the same command on all files specified in the input.
- Usually used with “find” output, e.g.:

```
find . -name '*.nc' | xargs chmod u=rwx
```

Changes permissions on all .nc files.

Operating on multiple files: xargs (continued)

- by default splits the file list into *batches*:

```
chmod 644 file1 file2 … file100
    chmod 644 file101 file102 …
```

- use “-n 1” if the command can only process one file at a time:

```
find . -name '*.tar' | xargs -n 1 tar -tvf
```

  - displays contents of all 'tar' files found

The ssh agent

- Stores secret keys in memory.
- Avoids repeated typing of the pass phrases.
- Can talk to a forwarding mechanism.
  For example:
  - your workstation → jasmin-login1 → jasmin-sci1
  - jasmin-login1 does not have the private key
  - authentication traffic forwarded from end to end:
    - jasmin-sci1 sends challenge
    - workstation sends response, proving your identity

The ssh agent (continued)

- To start the agent and load your secret key:
  - Linux: session manager should start agent for you. Use ssh-add to load key (if not done automatically when ssh first used).
  - Windows: launch Pageant and click “add key”.
  - Enter your pass phrase.
- For authentication forwarding:
  - Linux: use “ssh -A” (often the default)
  - Windows: in PuTTY, go to Connection → SSH → Auth, and “Allow agent forwarding”

Other ways to move data around

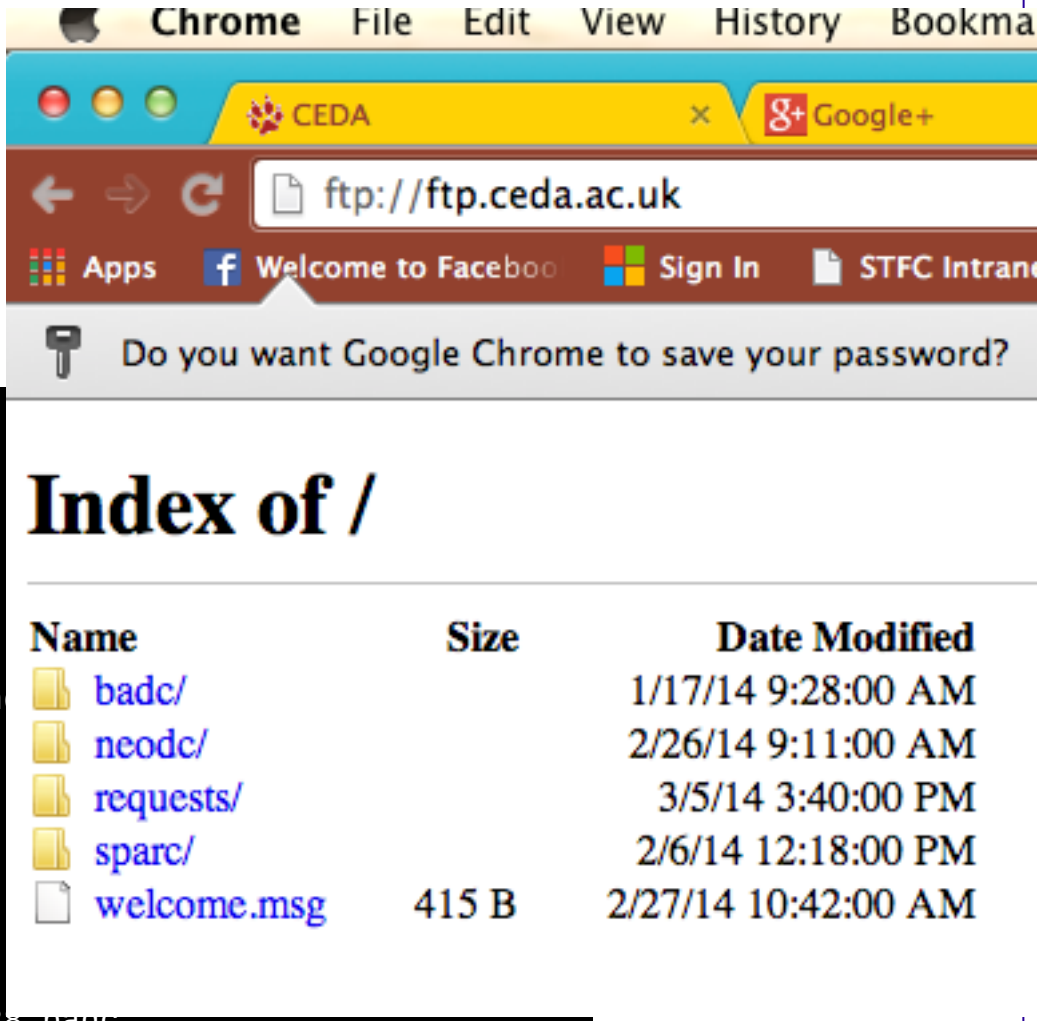
There are a lot of tools to help you move data from one machine to another. Common ones are:

- FTP
- SFTP
- Rsync
- Wget
- Curl

Transferring data with FTP

Can use most browsers to ftp files

Can also use a command line interface too (easy to script)




```
vpn-2-150:~ sjp23$ ftp ftp.ceda.ac.uk
Connected to ftp1.ceda.ac.uk.
220 JASMIN BADC/NEODC FTP server
Name (ftp.ceda.ac.uk:sjp23): spepler
331 Password required for spepler
Password:
230-Welcome to the CEDA ftp server.

 This server provides read-only access to the BADC an
 archives and users 'requests' areas.

230 User spepler logged in
Remote system type is UNIX.
Using binary mode to transfer files.
ftp> ls
229 Entering Extended Passive Mode (|||65173|)
150 Opening ASCII mode data connection for file list
drwxr-xr-x   2 badc     byacl       28672 Jan 17 09:28 badc
drwxrwxr-x   2 badc     byacl        8192 Feb 26 09:11 neodc
drwxrwx--- 1812 badc     byacl      249856 Mar  5 15:40 requests
drwxr-xr-x   2 badc     byacl        4096 Feb  6 12:18 sparc
-rw-r--r--   1 badc     ftp           415 Feb 27 10:42 welcome.msg
226 Transfer complete
ftp>
```

Secure Shell

Transferring data with sftp

- Like scp, this uses ssh. However, gives an interactive interface like ftp.
- Usage (Linux):
  - "sftp *host*" or "sftp *username@host*"
  - ftp commands e.g. cd, lcd, put, get
- Windows:
  - psftp (in PuTTY suite) works similarly from command line
  - also Filezilla GUI
- As before, set up ssh keys first.

Transferring data with rsync

- copies files over the network (or locally)
- where destination files already exist, copies only what is required to update any differences
- push / pull files over ssh:

  rsync -e ssh *user@host*:*remote_path local_path* ← pull

  rsync -e ssh *local_path user@host*:*remote_path* ← push

  - requires no special configuration (though remember to set up ssh keys)
  - similar to scp syntax, e.g. remote path is relative to home directory unless starts with /

Transferring data with rsync (continued)

- Useful flags for rsync:

  -r (recursive) – go down the directory tree copying stuff.

  -c (checksum) – when deciding what files to send, look not only at size and timestamp but if necessary also file contents

  --delete – remove files from destination not present at source end. *(Test with -n first!)*

  -v (verbose) – list files that are transferred (or deleted)

  -n (dry run) – go through the motions but do not actually transfer (or delete) files. Useful with -v.

  -a (archive) – copy recursively and try to copy permissions, ownership, etc.

Pattern matching: globs

- Unix shells recognises various wildcards in filenames. We have seen these two:

  * matches any number of characters

  ? matches one character

- These filename matching patterns, known as “globs”, are replaced with a list of matching filenames before the command is executed.

```
$ ls
1 3 5    a1   b1   c1   d1
2 4 a    b    c    d

$ ls *1
1 a1     b1   c1   d1

$ ls ??
a1 b1    c1   d1
```

Pattern matching: globs

- Here is another glob for you

  […] matches any of the characters listed (or range of characters, e.g. [0-9])

```
$ ls [a-c]*
a a1   b   b1 c   c1
```