Pipes 2 (1)

- what are the more advanced things you can do with pipes?

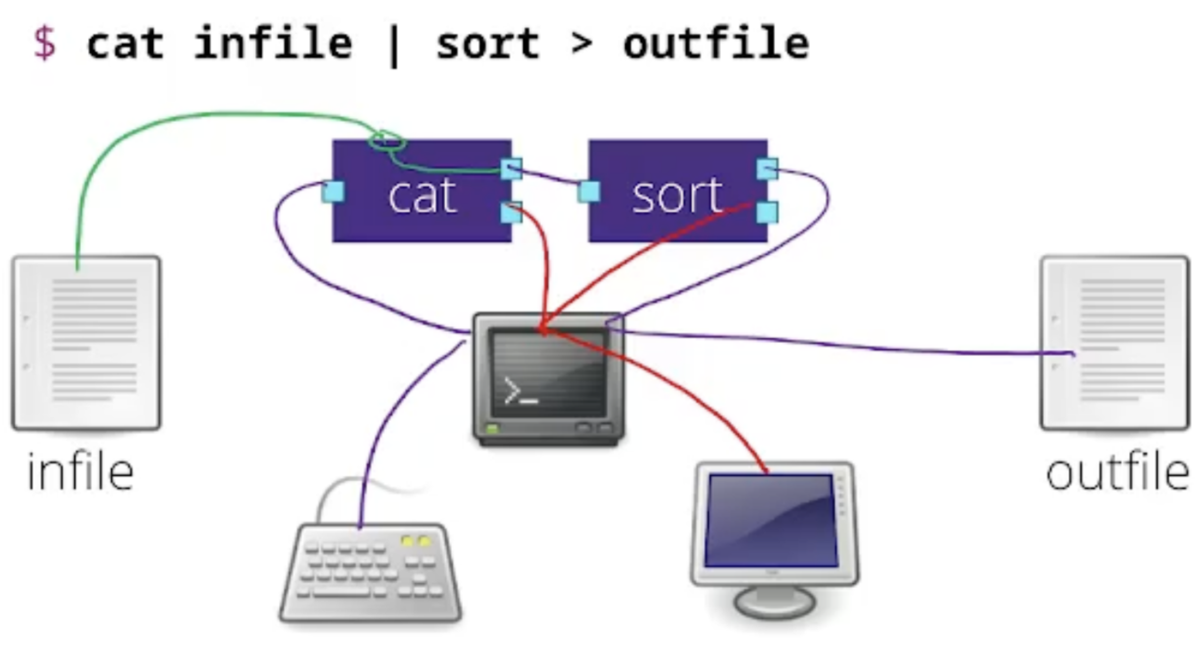
Redirects

```
cat inputfile | sort
```

- cat a file and pipe it to sort, the outoput of the sorted file will be displayed in the shell
- but say you want to save that output to a new file instead?
- the answer would be to use a **redirect**

```
cat inputfile | sort > outputfile
```

- the > operator tells you where to put the result of a command, instead of simple displaying it in the shell



Redirecting Files to Standard Input as text stream

- the opposite of above so use the less than operator `<`
- pushes the result to standard input

```
sort < inputfile > outputfile
```

- sort gets passes the input file and then the sorted result is piped to the output file

What is the practicasl difference in using the less than or ommitting it?

- often you get the same result

Many tools such as **sort** are designed in a way that if you pass them a file name as an argument, they know that you want them to read input from that file instead of standard input.

It is important to understand the difference:

- running sort file means sort wil take that file as an argument whic sort (and similar programs) know they need to open and read
- running it with the < operator means that sort will get its standard input from the given file

But which process is openign the file?

- is it sort or the shell?

This is really important as some programs will not know what to do with a file name you pass them as an argument

- they might have only been designed to work with text gathered from standard input (not via files)

An alternative way to the above code example:

```
cat filename.txt | sort
```

- here we are passing the output of *cat* into *sort* as sort's standard input
- however this is unnecessary as the shell would do this using the < > operators shown previously, all cat is doing here is opening the file

cat followed by a pipe (|) is a **useless use of cat**

- cat in this situation isnt concatenating files together in this situation, it is a glorified file reader here, which the shell couldve done for us

Redirecting to files that already exist

- **do you want to overwrite the exisiting file? or append to it?**

```
sort unsortedfile > outputfile   // overwrites the output file

sort inputfile >> outputfile  // appends to the outputfile, the
```

Error Redirects

- redirecting standard error instead of standard output, or redirect them both to seperate files
    - great for seperation
- to do this you must specify **pipe number 2** as part of the redirect:

```
COMMAND > FILE 2> FILE2         // redirects SE to FILE2 and St
```

Sometimes you might want to capture **all** of the output from a program, similar to how it appears in the terminal, e.g. having a file which displays the output and error messages together:

- helpful for when you want to analyse a process that has been left running for example

```
COMMAND > FILE 2>&1

e.g.
ls -1 > FILE 2>&1

now FILE contains the output of ls -1 and any errors as well
```

- you redirect SO to a file, then say you want to redirect SE to the **same place as SO**
    - that is what the **&1** part is doing
- the **order matters** as the shell interprets them in order

this command would be **invalid:**

```
COMMAND 2>&1 > FILE // INVALID
```

- this would send SE to where SO is currently, i.e. the terminal and then later would redirect SO to the FILE, but we never update where we want to redirect SE, it is still going to the terminal

Example Question:

**Which of the below would redirect both standard output and standard error from myprogram
to log.txt?**

```
A: ./myprogram 1> log.txt 2>&1

B: ./myprogram 2&1> log.txt
```

CORRECT ANSWER: A

- firstly directs standard output to log.txt then `2>&1` will direct standard error to the **address** of where standard out was redirected to
- `2&1>` is simply invalid syntax for option B

Ignoring a command's output

- say if a command is being very noisy and you dont want to see its output:

```
COMMAND > /dev/null
```

- anything you write to /dev/null is **thrown away**

Recap of Files vs Streams

- you can send a file to a programs standard input as a text stream like this:

```
PROGRAM < FILE
```

- you can redirect the Standard Output of a program to a file like so:

```
PROGRAM > FILE
```

- you can redirect a program's Standard Error instead/as well if you specify pipe 2:

```
PROGRAM 2> FILE
```

If you have a program which has been designed to expect a filename, you can tell it to instead use standard input or standard output instead by passing it the filename " - " (single dash) if the program supports it

```
pdftotext file.pdf - | grep software
```

- this command expects a file to output to however you can pass it the dash and then this allows you to pipe the standard output into other places/commands
    - e.g. now can use grep to search the SO of the pdftotext command, whereas before it would just create a text file

If you encounter a program that doesnt support the above, you can use the filename **/dev/stdin**

- this gives the same behaviour as the - filename

Filenames with dashes are generally considered bad

- if you make one and need to use it, refer to the current directory first so the program knows this is an agrument and not an option

```
rm ./ -f
```

- this lets rm know that it is a file being passed and not an option -f

Advanced Piping

tee

```
ls | tee FILE
```

- tee takes a filename as an argument and then **whatever** you write to tee as standard input it will copy this to the file as well as writing to standard output
- helpful for inserting in the middle of a long pipeline to debug and keep track of the flow

pagers

```
ls | less
```

- less is a pager, they display text on your screen one page at a time
- you can use less to scroll through terminal output
    - up/down arrows scroll
    - space/enter = advance page
    - / opens a search
    - q = quit

sed

- stands for **stream editor** - it can change text using regular expression as it passes from its standard input to its standard output, e.g.
- sed is a ‘transformer’ for text

```
echo " Hello World | sed -e 's/World/Universe/'


OUTPUT:

Hello Universe
```

- you change the pattern ‘World’ into ‘Universe’

Advanced Edge Cases for Piping

need a file, want a pipe?

- say if we have a program which isnt designed to handle streams of text? and expects to be given a filename so it can read from that file…
- instead of just creating a new file to feed it the input, as this would be a waste of storage, we can instead do:

```
PROGRAM <(COMMAND)   // enclose the command we use in parenthese
```

- the program then thinks we are giving it a filename

```
cat <(echo "Hi")
```

- cat usually expects a filename as it needs to concatenate the file it is given

```
echo <(echo "Hi")
```

subshell to argument

- where you want to take the result of a command and pass it to another command as **neither** standard input or as a pretend file but as an **argument for the command**
    - something youd usually type

do this using the dollar sign:

```
COMMAND $(SOMETHING)



OLD FASHIONED WAY OF WRITING THIS USING BACKTICKS:

COMMAND `SOMEHTHING`
```

- useful when you want to call a program which specific arguments **depending on the results of running another command**