

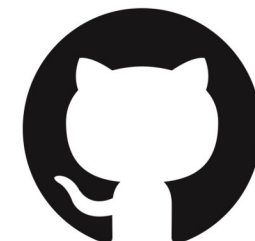
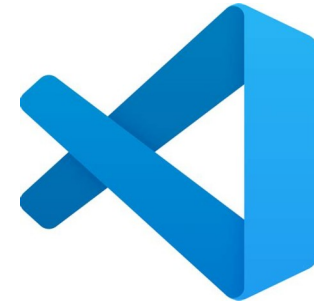
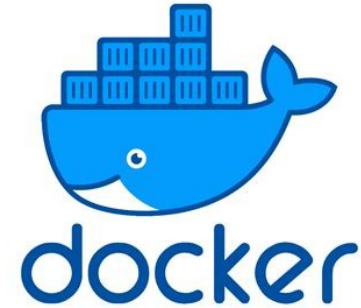
Setup and Introduction to the Shell

Mark Slater

UNIVERSITY OF
BIRMINGHAM

Overview

1. Setup of tools used in the course
2. What is the Command Line?
3. Files and Directories
4. Redirection, Pipes and Filtering
5. Scripts



1. Setting up for the Course

Setting up for the Course

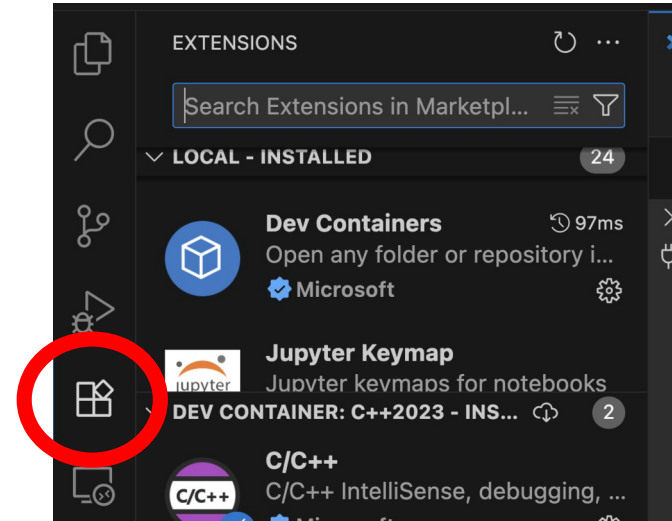
- Throughout this course we will use Docker and the Visual Studio Code IDE (Integrated Development Environment) to:
 1. Provide the same OS agnostic sandbox/environment for everyone
 2. Ensure all Development tools are compatible with the material
 3. Provide Shell access for running the Development Tools
 4. Give a modern, C++ aware editor for code development
- There are many different ways of providing these elements, but we have tried to pick a system that is not only (relatively!) **easy to setup** but also close to how code development is done in the real world
- We will also be relying on **Github for version control**, providing base code to work from and feedback
- The details for the setup can be found here:

Pre-Course Software Setup

Step by Step

1. Go to [Github](#) and register for an account
2. Download [Docker Desktop](#) and install it
3. Run it – it will start up and keep running in the background
4. Download [Visual Studio Code](#) and install it
5. Run VSCode
6. Select the 'Extensions' option on the toolbar
7. Search for and install the following modules:
 1. C/C++
 2. Dev Containers
8. Go to 'View' → 'Command Palette' and run the option 'Dev Containers: Clone Repository in Named Container Volume...'
9. Paste the URL of the testing repository into the dialog:

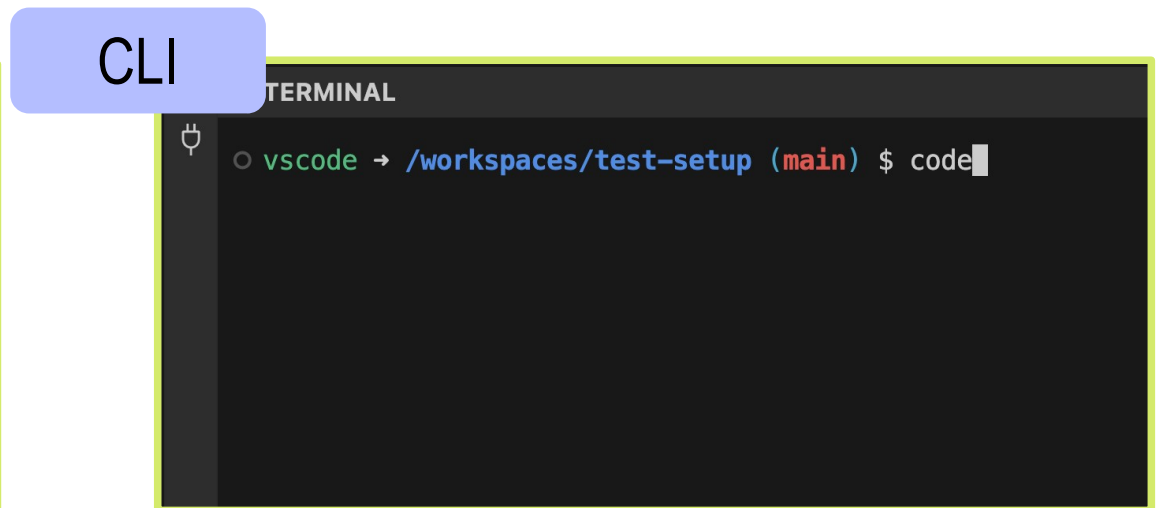
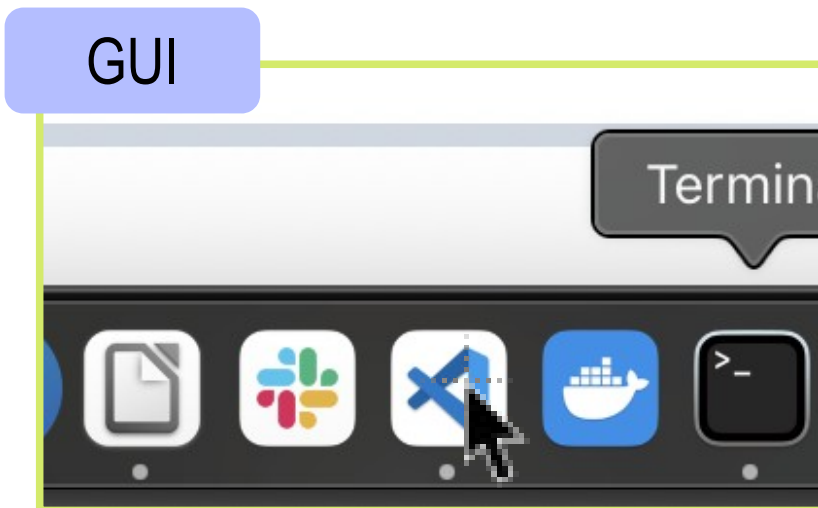
<https://github.com/MPAGS-CPP-2026/test-setup>
10. Provide a name for the container volume to use through the course (e.g 'MPAGS') and then accept the name of the default folder to clone the repo into ('test-setup' in this case)
11. After waiting for the container to build, go to 'Terminal' → 'New Terminal' to get access to the command line



2. What is the Command Line?

What is the Command Line?

- You're probably used to using **graphical interfaces** to control a computer or device
- This is generally done using a **mouse or touch screen** to select buttons or other elements that are shown with images
- In contrast, with the Command Line Interface you type text that is interpreted by the **Shell** such that the appropriate programs are run



Why do we use it?

- Using a CLI may initially seem to make things a lot harder. However, you get a lot more **power and control** and it **scales** a lot better
- For example, imagine having to find the most common word in each of 1000 text files and then output these words into a separate text file?
- With a GUI you would be clicking for a very long time and would be reliant on there being specific functions built into apps that would do what you want
- Using the CLI you can **'chain' together** simpler commands and affect how they work which means the above problem could be solved in **one or two lines**

Interacting with the Command Line

Command Prompt

After typing a command, any output is shown after the command

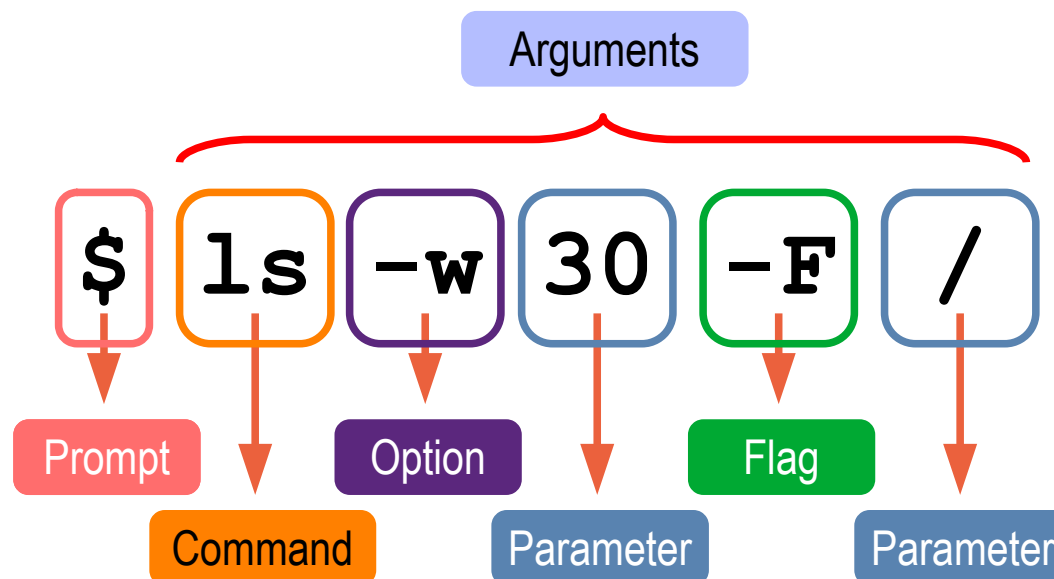
✓ **TERMINAL**

```
● vscode → /workspaces/test-setup (main) $ echo "HELLO"  
HELLO  
● vscode → /workspaces/test-setup (main) $ ls -ltr  
total 12  
-rwxr-xr-x 1 vscode vscode 262 Aug 30 14:38 runtest  
-rw-r--r-- 1 vscode vscode 1141 Aug 30 14:38 README.md  
drwxr-xr-x 6 vscode vscode 4096 Aug 30 14:38 shell-intro-data  
○ vscode → /workspaces/test-setup (main) $ █
```

Cursor – where you type your commands

The Command Prompt

- Most things you type at the command line will be **commands** followed by **extra information** to tell the command how to behave
- All the 'words' supplied at the command line are considered arguments and these arguments can be considered one of:
 1. **Command** – the first argument and what you want to run
 2. **Flag** – an argument that modifies the command (e.g. change output format, verbose output, etc.)
 3. **Option** – an argument that modifies the command and takes a parameter itself (e.g. output width)
 4. **Parameter** – an argument that provides information to the command or a given option (e.g. the input file to run over)



Exercise 1.1

- Type the following in to the terminal:

1. \$ **echo "Hello World"**

→ 'echo' outputs any arguments given

2. \$ **pwd**

→ prints out the current directory

3. \$ **ls**

→ list the contents of a directory

4. \$ **LS**

→ What happens? Note the capitalisation!

5. \$ **ls --help**

→ Most commands accept a form of this

6. \$ **man ls**

→ Another way to get help – use up and down arrows to navigate and 'q' to exit the display program

7. \$ **echo "Hello" [Ctrl-C]**

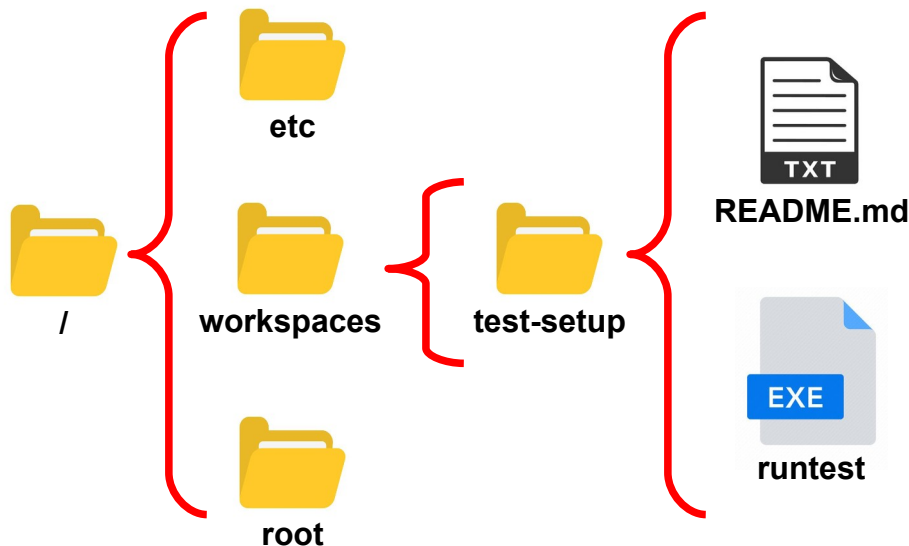
→ Holding down 'Control' and tapping 'C' is the 'Emergency Stop' button for the Shell – it stops programs running immediately (in theory!) and cancels the current input at the prompt

3. Files and Directories

Files and Directories

- The Operating System organises stored information using the **File System**. Data is stored in 'files' and these files are stored in 'directories' or 'folders'
- You are probably familiar with seeing **icons that represent files and folders**
- On the command line, the names are used with a **forward slash ('/')** indicating directory boundaries

GUI

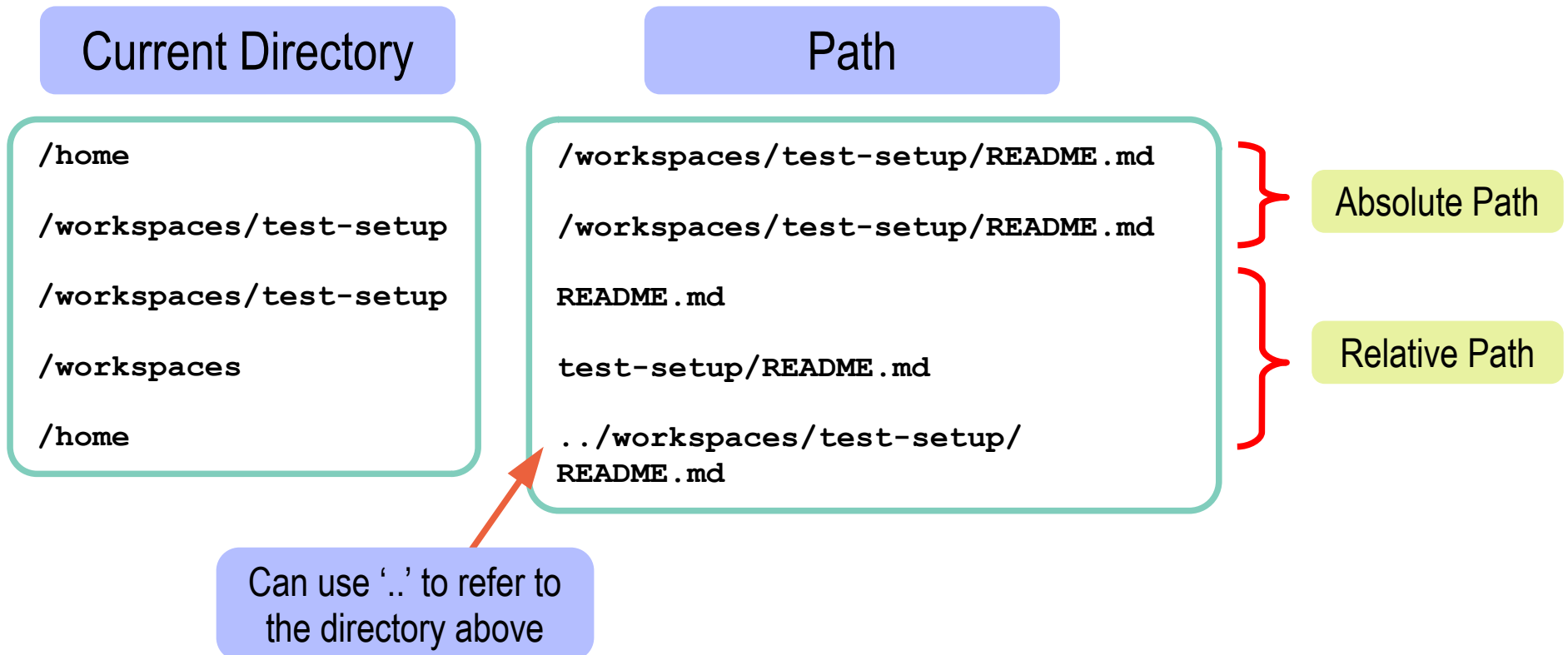


CLI

```
> ▾ TERMINAL
🔌 ● vscode → / $ ls /workspaces/test-setup/README.md
   ○ vscode → / $
```

Relative and Absolute Paths

- There are two ways to reference paths at the Command Line:
 - Absolute Paths** – The path starts with '/' (or similar) which is the *root* (or 'start') of the filesystem and doesn't depend on the current directory
 - Relative Paths** – These paths are found relative to the current directory
- Both are valid and each is useful depending on the use case
- All the following reference the same file:



Useful Commands

```
ls [-F, -a, -l, -r, -t, ..] <path>
```

List the contents of the given directory (the current one if none given)

Some useful arguments:

- F: Add characters to indicate the type of file/dir, e.g. '/' for dirs, '*' for executable
- a: Show all files, including hidden ones (those beginning with '.')
- l: Show in 'long' format which includes metadata info
- t: List files by last modified time
- r: Reverse the list of files

```
pwd
```

Print the current working directory, i.e. the directory you're currently in

```
cd <path, ~, ->
```

Change directory to the given path

If the tilde ('~') is given, change to the defined HOME directory (often /home/<username>)

If a dash ('-') is given, change to the previous directory

Exercise 1.2 – Using 'ls'

- Type the following in to the terminal:

1. \$ **ls -F**

2. \$ **ls -l**

3. \$ **ls -a**

4. \$ **ls -aF**

→ Here are various uses of the 'ls' command. To have a better idea of the differences, look at the help/man pages again!

5. \$ **ls -j**

→ What happens with this command?

Exercise 1.3 – Navigating the File System

- Type the following commands at the terminal. Note commands separated by ';' can be entered individually.
 1. `$ pwd ; cd / ; pwd`
 - As you can see from the prompt, we've changed the current directory. Do an 'ls' to see what files are in the 'root' dir of the filesystem
 2. `$ cd workspaces ; cd test-setup ; pwd`
 - We've now navigated back to where we were
 3. `$ cd -`
 - Where are we now?
 4. `$ cd -`
 - And now?
 5. `$ cd ~`
 - This takes us to the 'Home' directory which will vary by machine and OS. Use 'pwd' to find out where we are
 6. `$ cd /workspaces/test-setup`
 - As long as the path is valid, you can jump straight to the requested directory
 7. `$ cd ..`
 - Where are you now?

Useful Shell Function – Tab Completion

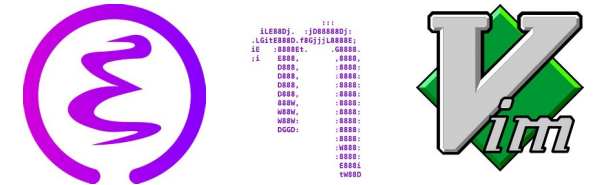
- A very useful function offered by the the Command Line is 'Tab Completion'
- This is where, if there is **no ambiguity** as to what you have started typing, if you press the TAB key, it will **auto fill** the rest of the path or command name
- Give the following a try:
 1. `$ cd /`
 2. `$ cd work [TAB]`
 - The shell should autofill with 'workspaces'. Note you could have just typed 'w' and it would also have worked
 3. `$ cd / ; cd m [TAB] [TAB]`
 - Because there are more than one dir starting with 'm', when you press TAB the second time it shows you the options available

Useful Shell Function – History

- Another useful function of the command line is [Shell History](#)
- As you type commands into the command prompt, they are stored and can be [easily retrieved or searched](#)
- You can just use the [\[UP ARROW\]](#) to scan through your history and then edit the line before pressing [\[ENTER\]](#) to run it again
- Try the following:
 1. Press the [\[UP ARROW\]](#) until you find an appropriate command and then press [\[ENTER\]](#) to run it
 2. To search your history, use [\[Ctrl + R\]](#) and then start typing a previous command you have used (e.g. echo). Again, you can edit it and then run it

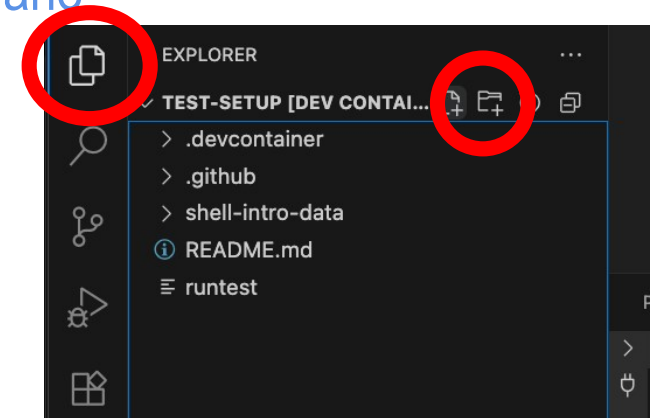
Creating Directories and Files

- We can navigate the file system from the command line – we will now look at modifying it
- To create a new directory you can use the '**mkdir**' command. This takes a path just like '**cd**' so you give any absolute/relative path you want
- To create a file, there are several options:



1. Use the VSCode editor to create a new file and then edit it
2. Use a command line editor such as Vim, Emacs or Nano
3. Use the '**touch**' command to create an empty file

- As an example, in the VSCode File Browser:



1. Create a new file and name it
2. Double-click to open it in the editor, type some text and save it
3. To confirm the file is present and check it's contents, use the '**cat**' command:

```
$ cat <filename>
```

Moving, Copying and Removing Files and Directories

- To manipulate existing files and directories, we use the following three commands:
 1. 'mv' will 'move' a file/dir to a different location
 - Note this also functions as a 'rename' command!
 2. 'cp' copies a file or directory to a different location
 3. 'rm' removes (i.e. deletes) a file or directory
 - BE CAREFUL! Anything removed is gone forever. There is no Recycle bin or Undo for the command line!

Multiple Paths and Wildcards

- You quite often want to perform a command over **multiple files/paths**, e.g. copy a set of files to a single destination, list a particular subset of files, etc.
- To do this, you can use the **wildcard system** in Shell which allows you to specify a **group of files** that match a particular expression (note this is glob rather than regex!)
- The most common is to use a '*' to indicate any (or none) characters, e.g. ***.txt** will list those files ending in **' .txt'**
- Directories will also count, e.g. ***/output** will match all files called **'output.txt'** in any directories within the current one
- Try the following:
 1. **\$ cd /workspaces/test-setup**
 2. **\$ ls shell-intro-data/*/*.dat**
 3. **\$ ls shell-intro-data/*/*.pdb**
- Note the different files that are shown by the **'ls'** command

Useful Commands

rm [-i, -r, ..] <path>

Remove (i.e. delete, destroy!) the given path(s). This cannot be undone!

-i: Interactive mode – force the user to confirm each deletion

-r: Remove directories and their contents as well. Use with care!

cp [-r, ..] <src1> <src2> .. <dest>

Copy the given files/paths to the destination. If the destination is a directory, files/paths will be copied into it

-r: Copy directories and their contents as well

mv <src1> <src2> .. <dest>

Move the given files/paths to the destination. If the destination is a directory, files/paths will be moved into it. Can be used to rename paths.

cat <file path>

Print the contents of the file to the terminal

mkdir [-p, ..] <path>

Create the directory at the path specified

-p: Create all parent directories as well

touch <path>

Creates an empty file at 'path' or updates the timestamps to current time

Glob-style wildcards

Specify wildcards in paths to include all the matching files/directories

*: Matches any number of characters, or none

?: Match any single character

[..]: Matches any characters in the square brackets

[0-9]: Matches any characters in the given range, e.g. 0-9, a-z, A-Z, etc.

Exercise 1.4 – Manipulating Files and Directories

- Try these commands:

1. `$ cd /workspaces/test-setup ; mkdir testdir ; ls`

→ This will create a directory in the 'test-setup' directory

2. `$ mv testdir new_testdir`

→ This is using the 'mv' command as a rename command

3. `$ touch new_testdir/new_file.txt ; ls new_testdir`

→ This will create a new, empty file

4. `$ cp new_testdir/new_file.txt new_file_copy.txt ; ls`

→ This will create a copy of the blank file in the current dir

5. `$ rm -i new_file_copy.txt ; ls ; ls new_testdir`

→ This removes the copy we created but the original is still there

6. `$ rm new_testdir`

→ What happens?

7. `$ rm -ri new_testdir ; ls`

→ And now?

4. Redirection, Pipes and Filtering

Storing the Output of Commands

- You've seen that most commands will provide output to the terminal
- It can often be very useful to [store this output in another file](#) and you can do this using [redirection](#)
- If you put the greater than sign ('>') at the end of a command followed by a filename, any output is sent to a new file with that name
- Try the following:
 1. `$ echo "HELLO AGAIN" > test_output.txt`
 2. `$ cat test_output.txt`
 - To double check, open the file in the VSCode editor
- If the file exists, any contents will be replaced by this output. If you want to append to a file, use '>>':
 1. `$ echo "HELLO YET AGAIN" >> test_output.txt`
 2. `$ cat test_output.txt`

Exercise 1.5 - Chaining Commands together

- Using redirection, we can now chain some basic commands together to find out which file has the fewest lines:
 1. **\$ cd /workspaces/test-setup/shell-intro-data**
 2. **\$ wc -l alkanes/*.pdb > wc_output.txt**
 - This command ('Word Count') with '-l' shows the number of lines in the given files
 3. **\$ sort -n wc_output.txt > sort_output.txt**
 - Sort will sort (unsurprisingly!) the lines of a file. The '-n' option will interpret the first word on a line as a number
 4. **\$ head -n 1 sort_output.txt**
 - This will print out the starting 'n' lines (defaults to 10) of the given file. In this case, it's the first line which will be the file with the fewest lines
- This is an example of using some basic commands and options to filter files/command output which can be used to:
 - Get output into a form that a dedicated application requires
 - Answer general/high-level questions about the information
 - Reorganise the data into different files

Using a Command's Output as Input – Pipes

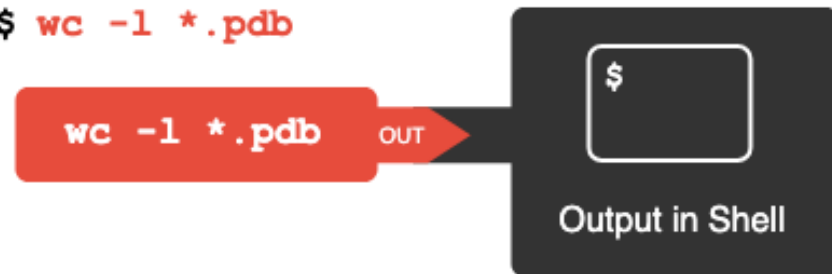
- Though this method of combining simpler commands works, it isn't very efficient and can leave a lot of temporary files
- It would be a lot 'neater' to just take the **output of a command and 'pipe' it as the input to another command**
- A lot of commands are actually designed to do this – i.e. accept data from the 'standard input' (stdin) rather than an input file
- To do this, you use the bar/pipe character '|' between the commands
- To achieve what we just did but without the temporary files, do the following:

```
$ wc -l alkanes/*.pdb | sort -n | head -n 1
```

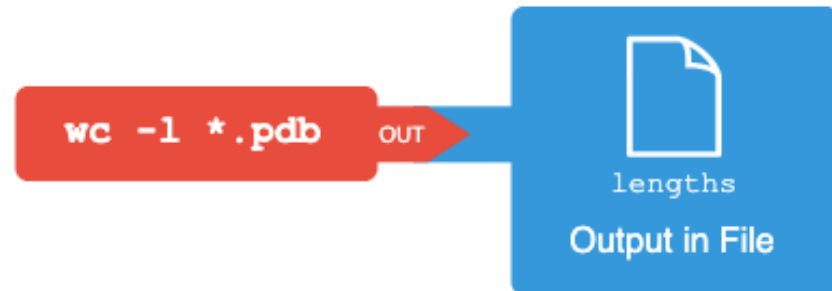
- Note that the input filenames aren't used after the first command

Overview of Redirection and Pipes

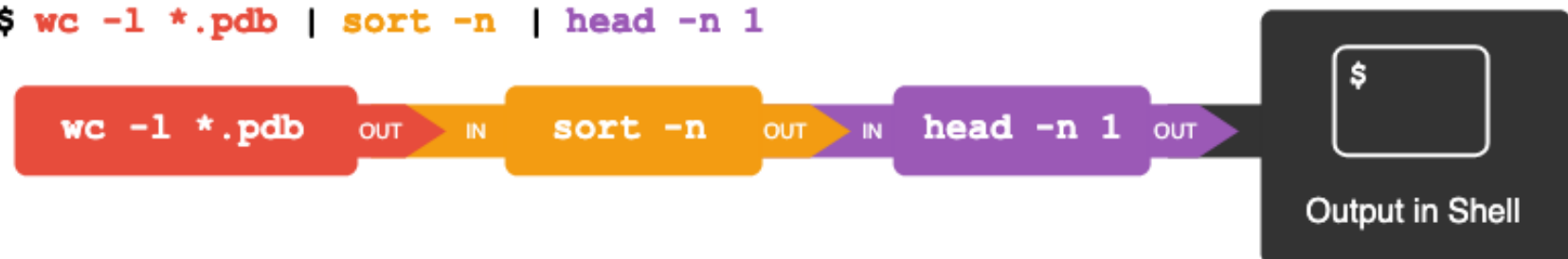
```
$ wc -l *.pdb
```



```
$ wc -l *.pdb > lengths
```



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



Useful Commands

```
wc [-l, -c, -m, -w, ..] <files>
```

Count the number of lines, characters, bytes and/or words for a file or set of files. With no arguments a summary is shown

Some useful arguments:

- l:** Count just lines
- c:** Count bytes (not necessarily the same as characters!)
- m:** Count characters
- w:** Count words

```
sort [-n, -r, ..] <file>
```

Sort the given file/stdin based on alphabet or number

Some useful arguments:

- n:** Interpret sorted key as numeric
- r:** Reverse the sort

```
head/tail [-n N, ..] <file>
```

Show the first/last N (10 by default) lines of input

5. Scripts

Storing Commands To Run Later

- Typing out multiple commands can get quite tiresome and be prone to mistakes, even when using the shell history
- As we will see later in the main C++ course, anything you're likely to do more than once **should be stored and then called by a single command**
- Any sequence of commands you wish to store and 'call' later on can be written in to shell script (typically .sh ending though not required) and called by doing:

```
$ bash shell_script.sh
```

- **Aside:** By setting the 'executable' permissions on a script file and adding a sh-bang (**#!/bin/bash** or similar as the first line) it can be called directly as well:

```
$ ./shell_script.sh
```

Exercise 1.6 – Writing the Script

- We're going to write a (very!) short script that will combine the 'head' and 'tail' commands to find the middle line of a file.
- Create a new file in VSCode called 'middle.sh' in the 'shell-intro-data' folder and add the following to it:

```
head -n 15 alkanes/octane.pdb | tail -n 5
```

- Now run the script using:

```
$ cd /workspaces/test-setup/shell-intro-data
```

```
$ bash middle.sh
```

- You should hopefully see the lines from 11-15 of the octane.pdb file displayed!

Processing Arguments in a Script

- This is a good start but as you have probably realised, the file to run over is [hard-coded into the script](#) so you need to edit the script if you want to run over a different file or you're not in the correct directory, e.g. this will generate an error

```
$ cd ../
```

```
$ bash shell-intro-data/middle.sh
```

- Ideally, we'd like to specify when we run the script what file to run on. This can be done using arguments
- As stated before, arguments given when calling a command are [sent to the program \(or script\)](#) for processing. We can access these in the script with the '\$1, \$2, \$3, ..' variables.
- Change the code inside 'middle.sh' to:

```
head -n 15 $1 | tail -n 5
```

- Now run the script again:

```
$ cd /workspaces/test-setup/shell-intro-data
```

```
$ bash middle.sh alkanes/pentane.pdb
```

- You should hopefully see [lines 11-15 of the pentane.pdb file](#) displayed this time
- You can take it further and replace the '15' and '5' hard-coded values with arguments as well

Summary

- This quick introduction to the Command line and Shell has hopefully shown you how to:
 1. Issue commands at the prompt
 2. Navigate the File System
 3. Modify the files and directories in the file system
 4. Chain basic commands together to perform complicated functions
 5. Store sequences of commands in a script to be run together
- This has barely scratched the surface of what you can do with the Shell (which is a programming language of it's own) but has given you enough knowledge and experience to be comfortable with using it in the course
- A useful website for understanding Shell commands is:

<https://explainshell.com>