Pipes 1 (1)

what are they?

- ways of getting input and output to programs
- allows you to pass the output of one command into the input of another, using the pipe symbol: ' | '

UNIX Philosophy

Brief history of UNIX

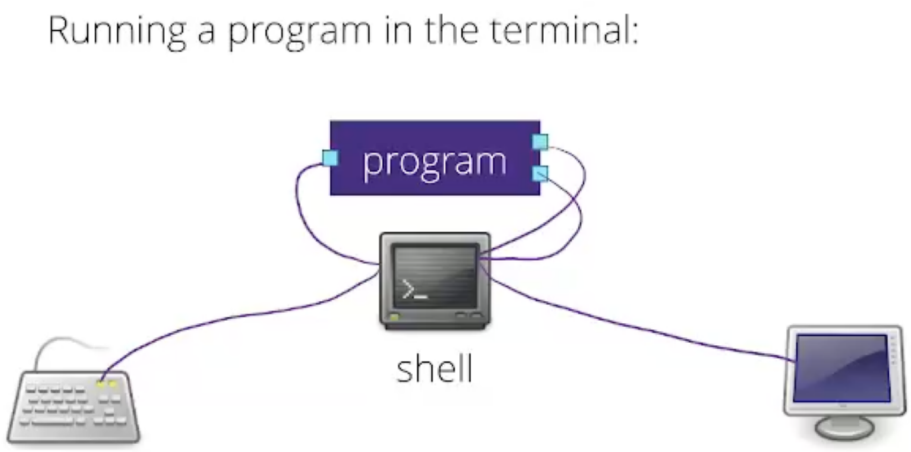
- is an operating system
  - e.g. a suite of programs that make a computer work
- a design philosophy
- a way we should design software in order for it to work

The core of the UNIX philosophy is that it is far easier to maintain 10 small programs than one large one, therefore:

1. each program should do **one thing well**
2. Programs should be able to **cooperate** to perform larger tasks
3. The **universal** interface between programs in the UNIX world is a **text stream**
   a. e.g. debugging is easier if you can read the output of one program that is going into another program and actually be able to understand it

Standard input/output

- every program will be launched with **three file descriptors:**
  - 0 = standard input
  - 1 = standard output (normal output)
  - 2 = standard error (writing error messages)
  - each program will have a standard input where it can read from, and two output pipes which it can write to



- the shell handles standard input
  - it accepts it from the keyboard
  - then passes it to programs etc

pipe example

```
ls -1 | head    // ls -1 puts output into one column
                // head (no args) just shows you the top 10 line

ls -1 | head -n 6 // shows first 6 lines etc etc
```

- the command ls -1 is being piped into the command head

- the pipe character basically says: *“we will send the output of ls -1 to another program”*
- *head* is a program which expects input
- what we’re doing in the above example is taking the standard output that would usually be sent to the terminal, and instead send it to head using the pipe operator

You can can piping commands into commands and then the output of those you can pipe again into other commands

grep

- “Global Regular Expression Parser”
- a sort of pattern matcher, it will remove any lines which dont match the pattern

e.g.

```
ls -1 | grep software
```

- grep will get rid of any lines that ls -1 sends it which dont contain the word “software”

example from personal files:

```
tom@Toms-MacBook-Pro ~ % ls -1 | head
Add_ASM
Applications
Counter.circ
Desktop
Documents
Downloads
IdeaProjects
Library
Logisim
MakeFile
```

```
tom@Toms-MacBook-Pro ~ % ls -1 | head -n 2
Add_ASM
Applications
tom@Toms-MacBook-Pro ~ % ls -1 | grep L
Library
Logisim
tom@Toms-MacBook-Pro ~ % ls -1 | grep op
Desktop
toprow.c
tom@Toms-MacBook-Pro ~ %
```

We can also take the standard output of grep which is currently being displayed in the terminal and pipe it again into another command, for example to sort them in reverse order:

```
tom@Toms-MacBook-Pro ~ % ls -1 | grep .c
Applications
Counter.circ
Documents
IdeaProjects
Music
Pictures
Public
c_projects
c_projects.c
hello_world.c
labTestPart1.c
main1.c
prime_numbers.c
toprow.c
tom@Toms-MacBook-Pro ~ % ls -1 | grep .c | sort -r
toprow.c
prime_numbers.c
main1.c
```

```
labTestPart1.c
hello_world.c
c_projects.c
c_projects
Public
Pictures
Music
IdeaProjects
Documents
Counter.circ
Applications
tom@Toms-MacBook-Pro ~ %
```

Redirect

```
$ cat textfile | sort > outputfile
```

- in this, you take a text file and pipe it to sort which then redirects the sorted output to a text file instead of the standard output (terminal or console)
- so the greater than `>` character sends a text stream to a file

```
$ sort < inputfile > output file
```

- `<` the less than character does the reverse
- in this example, sort is going to get its input from this file, instead of the standard input

So these two ways to use redirects:

1. Redirecting standard output to a file
2. Redirecting a file to standard input as a text stream

there are some more, however:

- when writing to a file that already exists, do we overwrite or add it to the end of the existing file?

```
$ COMMAND > FILE // overwrites the file

$ COMMAND >> FILE // appends to the file
```

what about when we want to redirect the standard error instead of stdout?

- to do this we must specify pipe number 2
  - **stdout is number 1**
  - **stderr is number 2**
  - e.g. we can specify to redirect stdout to one file and also stderr to a different file

```
ERROR REDIRECT:

$ COMMAND > FILE 2> errorFILE

$ COMMAND > FILE 2>&1 // this redirects stdout to a file and the
                      // std err to the same file could use dif
                      // this is the &1 part


// you have to do it in this order however,

INVALID:
```

```
$ COMMAND 2>&1 FILE
```

- first, we have to send stderr to the same place as stdout which is the terminal
  - NEEDS MORE NOTES

Ignoring output

- sometimes a command will give a noisy output which you don't want to see
- so you could send this to a file so it won't appear on the stdout, but this also takes up memory
- you could instead redirect to dev null:

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

- also works for stderr outputs
- everything you write to dev null is thrown away

files vs streams

a program that uses a standard stream can be told to use a file instead:

```
PROGRAM < FILE // standard input

PROGRAM > FILE // standard output

PROGRAM 2> FILE // standard error
```

also, a program that expects a filename can be told to use standard in/out instead:

- using the filename

Filenames with dashes

- file names starting with dashes are generally considered **bad**
- dashes are part of how options are presented to programs

Advanced Tricks & Tools

tee

```
$ ls | tee FILE
```

- tee takes a filename as an argument and writes a copy of input to it
  - it will both write this to standout and to the file
- useful to insert in the middle of a long list of a complex pipeline of UNIX commands
  - wont impact output going forward but allows you to check and debug

pagers

```
$ ls | less
```

- pagers take direct control for how the terminal draws things
- less displays text on screen one page at a time
  - helpful for making use of large output of commands
  - allows you to scroll

    - and search

`more` exists but is more limited to `less`

sed

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

OUTPUT:

Hello Universe
```

- stands for stream editor
- it can change the text that is going from standard input to standard output
- e.g. s/ONE/TWO/ replaces the first match for ONE with TWO
    - it is a powerful tool

need a file, want a pipe

- if PROGRAM wants a file to read in from, how can I pipe something in?

```
$ PROGRAM <(SOMETHING)


$ cat <(echo "Hi"
Hi //output


$ echo <(echo "Hi")
/dev/fd/63
```

- NEED NOTES AND UNDERSTANDING

Subshell to argument

```
$ COMMAND $(SOMETHING)
```

- NEED NOTES AND UNDERSTANDING