Intro to the Shell

In case you want to look at this when we're not available...

J. Koziej C. Van West

The Cooper Union

Summer '22

Welcome to the shell!

The shell looks like this:

```
user@host:dir$ _
```

`user` is your username, `host` is the hostname of the computer, and `dir` is your current directory (generally starts at ~, an alias for `/home/user/`). _ represents your cursor position.

Talking to it

Typing a command at the prompt and pressing `Enter` runs it – for example,

```
user@host:~$ date
_
```

(typing `date`, then pressing `Enter`) might give the output

```
user@host:~$ date
Thu Jul 14 07:44:41 PM EDT 2022
user@host:~$ _
```

Talking to it

When typed again, the date will (of course) change:

```
user@host:~$ date
Thu Jul 14 07:44:41 PM EDT 2022
user@host:~$ date # let's just make sure
Thu Jul 14 07:45:16 PM EDT 2022
user@host:~$ _
```

Note that any text following a # is considered a **comment** and ignored by the shell. I'll use comments to make brief notes on commands throughout, so know they're not necessary if you just want to run a command.

How to get information

Commands are usually documented via another command, called `man`. If you don't know how to use a command, typing `man <command>` is a quick way to find out. If we didn't know how to use `date`, we could view the manual page by running

```
user@host:~$ man date
_
```

How to get information

This would open `less`, a file-viewing command, with the documentation file (known as a **manpage**):

```
DATE(1)       User Commands       DATE(1)

NAME
       date - print or set the system
              date and time

SYNOPSIS
       date [OPTION]... [+FORMAT]
       date [-u|--utc|--universal]
...
```

Command structure

The first word in a command is its **name** (the name of the program to be run, in fact), and all subsequent words are known as **flags** or **arguments**. For example,

```
user@host:~$ echo hello world
hello world
user@host:~$ _
```

is the result of running the command echo with the arguments `hello` and `world`.

Command structure

Flags are a subset of arguments; the term usually refers to predefined options preceded by – or ––. For example, running

```
user@host:~$ date --utc
Fri Jul 15 12:09:22 AM UTC 2022
user@host:~$ _
```

tells `date` to give the date in universal time by passing it the `--utc` flag.

Flags often have a short and a long version; `-v` and `--verbose` are often equivalent, for example. Flags may also sometimes be combined: `rm -r -v -f` may be written `rm -rvf`. Different commands have different ways of interpreting flags, so check the manual page if you’re not sure!

Directories and navigation

The shell sees space as a tree:

```
user@host:~$ cbonsai -pl -L16 -b2 -s6

                   &&&   & &
          &         &&\ &&&& &&
        &&&&&_\&&  &/&&\ &  &
      & &&&&_&&&&|&\|/~ &
            & &&  \//&&&
              &   \/&&&& &
                   /~~
            (---./~~~\.---)
             (           )
              (_________)
user@host:~$ _
```

Looking around

All the “stuff” on the machine – its files and directories – are contained within other directories, leading up to the root directory (denoted /, a single slash). The program tree is a great way to visualize this:

```
user@host:~$ tree
.
├── garbage
│   └── gpa.txt
└── my_stuff
    ├── schedule.pdf
    └── todo
        └── list.txt

3 directories, 3 files
user@host:~$ _
```

Looking around

The most used command for looking around is `ls`, which prints the contents of a single directory. Used on the directory tree above, you might see

```
user@host:~$ ls
garbage  my_stuff
user@host:~$ ls -a # show hidden files
. .. garbage  my_stuff
user@host:~$ ls -a my_stuff # list a directory
. .. schedule.pdf  todo
user@host:~$ ls -aF my_stuff # classify files
./  ../  schedule.pdf todo/
user@host:~$ _
```

Around 12% of my own command history consists of `ls` with occasional flags. It is the closest thing the shell has to a handheld lantern.

Changing directories

You can see where you are in the tree using pwd, and move around using cd. In the example tree from above, you might navigate like so:

```
user@host:~$ pwd
/home/user
user@host:~$ cd garbage
user@host:~$ pwd
/home/user/garbage
user@host:~$ ls
gpa.txt
user@host:~$ cd ../my_stuff/todo
user@host:~$ pwd
/home/user/my_stuff/todo
user@host:~$ ls
list.txt
user@host:~$ _
```

Changing directories

There are a few special directory names you should be aware of:

`.`	the directory you're in
`..`	the directory above yours
~	your home directory (as a prefix)
/	the root directory

For example, `cd ..` would move you one directory upwards, and `cd ../..` would move you *two* directories upwards: it looks in `..` (the parent directory) to find another `..` (the parent of the parent directory) and goes there. `cd /` would take you to the root directory, and `cd ~` would take you home.

Dealing with files

Files can be copied using cp, moved using `mv`, and deleted (forever, so be careful) using `rm`:

```
user@host:~$ ls
original.txt
user@host:~$ cp original.txt copy.txt
user@host:~$ ls
copy.txt  original.txt
user@host:~$ mv original.txt moved_from.txt
user@host:~$ ls
copy.txt  moved_from.txt
user@host:~$ rm moved_from.txt
user@host:~$ ls
copy.txt
user@host:~$ _
```

Dealing with files

Two common commands to view a file's contents are cat, which prints the file out, and less, which opens it in an interactive viewer:

```
user@host:~$ ls
secrets.txt
user@host:~$ cat secrets.txt
What makes you think that we would just tell you?
Try looking harder.
user@host:~$ ls -a
.  ..  .actual_secrets  secrets.txt
user@host:~$ less .actual_secrets
_
```

Creating and editing

There are many ways to create or edit a file. Some involve streams (as you'll see in the next section). The most direct way, however, is to use a command-line editor. Here's a brief list:

`nano`	intuitive, integrated help bar
`vi`	powerful, but takes a while to learn
`emacs`	similar to `vi` in difficulty
`ed`	don't even think about it

Creating and editing

For example, to jot down some curmudgeonly thoughts using nano, simply run

```
user@host:~$ nano complaints.txt
_
```

and have fun. The file does not have to exist – it will be created if necessary when you save it.

Pipes and streams

Most shell commands are capable of taking an input and producing an output – `cat` takes a file and prints it out, `echo` prints its arguments (unless you invoke it with none!), `wc` prints out line, word, and byte counts for its input, `grep` searches for matching text in a stream, and so on.

So, of course, you can chain these commands together.

What is a stream?

A **stream** is a queue of data. Stuff (usually text, for our purposes) is written to the stream at one end and can be read from the stream at the other.

The two streams that matter most to shell users are the standard-input and standard-output streams, known as **`stdin`** and **`stdout`**. (There is a third, **`stderr`**, which is mostly for error handling.) For example, `cat` just writes the contents of its input file to `stdout`.

Pipes

Sometimes you want the output of one command to go to the input of another command. This is done using a **pipe**, denoted with a vertical bar (|):

```
user@host:~$ ls -a1 # all files, one per line
.
..
garbage
my_stuff
user@host:~$ ls -a1 | grep my # only MY stuff
my_stuff
user@host:~$ _
```

The | tells the shell to pipe the output of `ls -a1` (its `stdout` stream) to the input of `grep my`.

More stream destinations

Streams can also come from and go to files. The operators are as follows:

<	take `stdin` from file
>	send `stdout` to file
>>	same as >, but append

All of these operators are followed by a file name, indicating where to read from/write to. Several of these may be used in one command, along with various pipes, to accomplish whatever you're trying to do.

More stream destinations

Say you want to include a list of files in a story you're writing. You could use ls and a redirect to append a readout of your current directory to it:

```
user@host:~$ cat story.txt
All I had to my name was:
user@host:~$ ls -F >> story.txt
user@host:~$ cat story.txt
All I had to my name was:
garbage/
my_stuff/
story.txt
user@host:~$ nano story.txt
_
```

More stream destinations

If you wanted to get a list of all text and data files on your system and save them, you could run

```
find / | grep -E '(.txt|.dat)$' > files.list
```

which would pipe the output of `find` to the input of `grep`, and the output of `grep` to the file called `files.list`, overwriting whatever content it had before. You could then run

```
grep my < files.list | less
```

to view a list of all the files you just found whose paths contain “my”.

Handy tricks

There are certainly more, but here are my top three:

- Pressing `Tab` will often auto-complete command names, filenames, arguments, etc. I usually just type the first few letters of whatever I want and have the shell write the rest.
- Pressing ↑ will recall the last command you typed (useable multiple times!).
- Pressing `Ctrl-r` will allow you to search your command history for something you did before.

Finally: a brief incomplete summary

`man`	get info about command
`ls`	list files
`pwd`	print working directory
`cd`	change directories
`cp`	copy file
`mv`	move file
`rm`	remove file
`nano`	edit file
`cat`, `less`	view file

Table 1: Useful shell commands.

Finally: a brief incomplete summary

|	pipe `stdout` to `stdin`
<	take `stdin` from file
>	send `stdout` to file
>>	same as >, but append

Table 2: Stream redirects.

↑	recall last command
`Tab`	autocomplete text under cursor
`Ctrl-r`	reverse history search

Table 3: Trick reminders.

Also see Jacob’s shell cheatsheet! He put a lot of work into it.