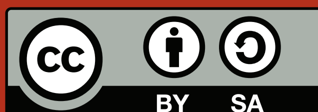
What is Computer Science?

An Information Security Perspective

Daniel Page ⟨dan@phoo.org⟩ and Nigel P. Smart ⟨csnps@bristol.ac.uk⟩

git # 148def3 @ 2018-07-11



A SHORT BASH TUTORIAL

Throughout this book, we have tried to use examples that can be typed into a computer and experimented with. The argument for adopting this approach should be obvious: actively *doing* things, rather than simply reading about them, offers a more interesting way to learn *plus* allows more in-depth investigation beyond the limited examples presented in print. A number of challenges complicate matters however. First, a vast number of different types (and configurations) of computer and operating system are available; meaning a single set of examples will, inevitably, be less than ideal for one set of people or another. Second, we need to find a careful balance between the advantages above and some potential disadvantages. Specifically, if we assume too much background knowledge or introduce too many new things to remember, there is a danger this confuses you or even discourages you. Much the same could be said of the Mathematics used, where we intentionally limited the notation for example.

To cope with these problems, we make two compromises. First, we focus the examples on use of a UNIX-based [22] operating system via the BASH shell [1]. Such a platform is provided by a variety of freely available options:

1. Use of a Linux (of whatever flavour or origin [6]) Live CD [11] perhaps represents the best choice: this allows use of a working operating system at no cost (bar the download) and without needing to actually install it. Virtually all distributions now offer a Live CD, with Ubuntu

   http://www.ubuntu.com/

   representing a popular and easy to use example

2. The fantastic Raspberry Pi project

   http://www.raspberrypi.org/

   has produced low-cost, complete computer (and associated Linux distribution) which is ideal for the examples in this book: beyond this, it has huge potential for exploring Computer Science more generally.

3. Cygwin [4] is a system that allows use of BASH on *other* operating systems, specifically versions of Windows:

   http://www.cygwin.com/

   The default installation of Cygwin provides the BASH shell itself, plus a basic set of commands; these alone are enough to support most of the examples.

A complete BASH reference guide is beyond our scope; this could easily fill another book entirely! There are plenty of such resources available, for example a free online books relating to the use of BASH and UNIX can be found at

http://en.wikibooks.org/wiki/Bourne_Shell_Scripting/

and

http://en.wikibooks.org/wiki/Guide_to_Unix

respectively, with the Computing At School (CAS) backed Raspberry Pi Educational Manual also containing highly relevant content. To address the second challenge above however, this Appendix presents an easy to digest (but more limited) introduction to the main concepts and commands used within the examples. We have tried to make it quite short: the goal is simply to demonstrate what each command does and how it can be used, not give a definitive list of every possible option.

To get the most out of what follows, the recommended approach is to first read through Section 1 which introduces important background concepts and terminology. Then, Section 2 should be used as a reference when or if you need to, rather than read from start to finish: it probably makes sense simply to refer to this latter section when you encounter an unfamiliar command elsewhere.

1 Some basic, background concepts and terminology

The natural place to start is from first principles, assuming little or no prior knowledge. As such, the goal of what follows is to define just enough concepts and terminology to support the rest of this Chapter. Some aspects of this might seem familiar, others overly simple or obvious: keep in mind that the Chapter overall is introductory, and intended to be a tool to explore the examples elsewhere rather than a definitive guide.

- A physical **computer** is comprised of many hardware components. These will almost certainly include a **Central Processing Unit (CPU)** [2], or **processor**, some short-term storage in the form of **Random Access Memory (RAM)** [15], and some long(er)-term storage such as a **hard disk** [8].

- The general-purpose hardware **executes** software **programs**, represented by a static set of **instructions**. When executed, the program becomes an separate, active instance of what is termed a **process** [14]. The concept of a process includes, for example, an environment in which the program executes; very abstractly, you could think of a program as a sort of template, and execution as taking the template and "using" it to form a concrete process.

  This means that several *distinct* processes could be executed based on the *same* program, each with their own *distinct* execution context. If you execute a web-browser three times for example, there is only one web-browser program but three separate processes representing each execution: what you do with one of them does not necessarily effect the others.

- A special, privileged program called the **Operating System (OS)** manages the underlying hardware and the processes executing on it. The OS is often described as the combination of

  1. a **kernel** [10], which represents the innards of the OS, and
  2. a **shell** [17], which is how users interact with the kernel and hence the computer as a whole.

  Put another way, the shell is basically a **user interface** while the kernel is tasked with duties such as (but certainly not limited to)

  – responding to **commands** issued by the user via the shell (e.g., in order to execute a given program),
  – ensuring each resulting process has access to **resources** it needs to execute (e.g., a large enough, distinct allocation of memory),
  – management of **multi-tasking** [12], so that more than one process can be executed concurrently,
  – performing operations, or **system calls** [20] on behalf of processes (e.g., perform low-level operations using the hard disk, or reading and writing data to and from the network).

  Depending on the type of shell, some commands might be real programs and some might be provided by the shell itself; these are sometimes called a **built-in** [18], but we ignore any distinction otherwise.

- The execution of a process is said to **terminate** (or finish) either

  1. normally (e.g., the computation it was executed to perform is complete),
  2. abnormally (e.g., due to an error during the computation), or
  3. forcibly (e.g., the user decides the computation is not required).

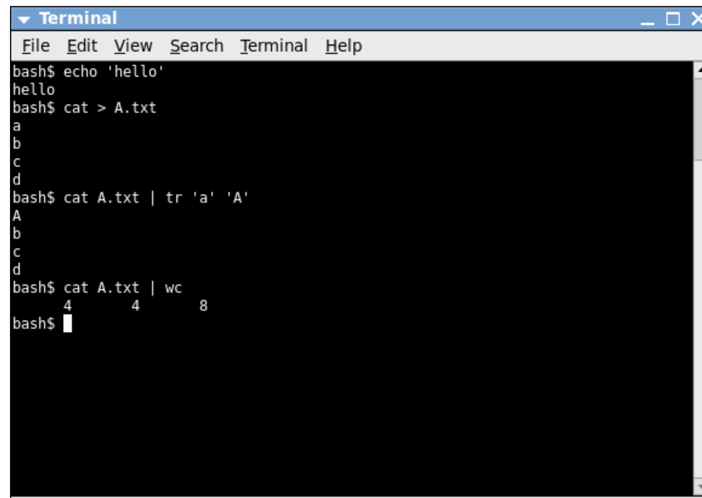


Figure 1: *An example BASH session within a UNIX-based terminal window.*

Arguably the most common way to interact with modern computers is via a **Graphical User Interface (GUI)** [7]. Their exact form and function might differ, but the basic idea is to offer an analogue to how humans interact with the world around them: using a mouse to point and click on icons and therefore issue commands is analogous to seeing a physical button and pressing it with a finger for example. In part, this style of interaction is motivated by the types of computer, user and also program that are most common. For example mobile computers with touch screens lend themselves better to use of a GUI than typed input (due to the lack of physical keyboard), non-expert users can quickly understand metaphors presented by icons (rather than remember text-based commands), and some of the most popular classes of application are naturally suited to graphical presentation of the data they process (web-browsing and word processing are central examples). Such advantages are evident from the first GUI-style shells pioneered [9] at Xerox PARC, through to modern equivalents in Apple and Microsoft products for example.

The taste and general requirements of different users will of course differ however, as will the tasks they undertake and hence programs they execute. Therefore, different non-GUI styles of interaction and hence shell are also useful. The **Command Line Interface (CLI)** [3] is an important example, where users interact with the computer by typing commands into a text-based terminal [21]. Both GUI and CLI have advantages and disadvantages, but since there is no perfect choice a common compromise is to expose a CLI *within* a GUI via a terminal *window*: the idea is that the CLI executes within same framework as other GUI-based programs, giving a compromise between the two styles of interaction.

Our choice of using CLI-based examples is motivated mainly by practicality: they are easier to present in a typeset book, and make it easier to focus on, reproduce and explain concepts we are interested in. The default CLI-based shell within a UNIX-based OS is almost always BASH [1], which we use exclusively.

1.1 Issuing commands to execute processes

Consider an example BASH session, which mirrors the screen capture of a real terminal window in Figure 1:

```
bash$ echo 'hello'
hello
bash$
```

Here, the `bash$` part of the first and third lines represents a **prompt**: this shows the user has control, and specifically that the OS is waiting for their input. The user does not type the prompt: it is included simply to make the examples match what you see in the terminal. Rather, the user types the (rest of the) first line (i.e., the text `echo "hello"` then return) as input, and BASH produces the second line (i.e., `hello`) as output. Following the same terminology as above, the first line is a command instructing the OS to execute a program called `echo`. A process is created and managed by the kernel, in this case producing output represented by the second line before terminating naturally and returning control to the user.

1.2 Controlling execution using options and arguments

In order to control what a given command does and how it does it, we can provide a list of extra information when we invoke it. One way to think of this information is as a form of input: the process might use some items in the list to make decisions about what operation to perform, and other items as data to operate on.

All elements in the list are supplied after the command name, separated by spaces, but there are two different types:

1. A command line **option** is some *optional* information provided as part of the command. An option is specified using an identifier, and (optionally) some form of parameter associated with it (e.g., a string, or an integer). Where no parameter is necessary, an option is sometimes termed a **flag** or **switch** (since it either turns on or turns off some behaviour).

2. A command line **argument** is some *compulsory* information provided as part of the command. Unlike an option, the order of arguments is important: since there is no identifier involved, the only way one argument can be distinguished from another is by their position in the ordered list.

Normally the argument list comes after any options, but this rule is not always true. Even so, a set of examples should make things clearer:

```
bash$ echo    'hello'
hello
bash$ echo -e 'hello' 'goodbye'
hello goodbye
bash$ ls -a --format=long
total 164K
drwx------  2 page page 4.0K Jun 22 09:44 ./
drwxrwxrwt 15 root root 156K Jun 22 09:44 ../
bash$ ls --format=long -a
total 164K
drwx------  2 page page 4.0K Jun 22 09:44 ./
drwxrwxrwt 15 root root 156K Jun 22 09:44 ../
bash$
```

Notice for instance that

- in the first command, echo is provided no options, but one argument 'hello',
- in the second command, echo is provided one option identified by -e, which has no parameter, and two arguments 'hello' and 'goodbye',
- in the third command, ls is provided one option identified by -a but with no parameter, one option identified by --format with the parameter long, but no arguments, while
- in the fourth command, ls is provided one option identified by --format with the parameter long, one option identified by -a but with no parameter, but no arguments.

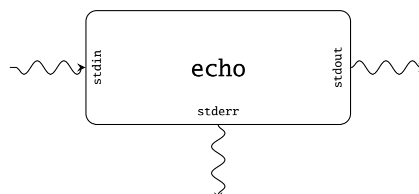
In the former two commands the differing options and arguments provoke different behaviour (which is clear from the output), whereas the later two commands produce the same output: this illustrates the fact that the order of options (in this case at least) is irrelevant.

1.3 Controlling input and output streams

Each process has an associated set of standard **streams** [19] which it can use to interact with the environment it is executed within. Specifically, each process can access

- standard input (or stdin), which is used to read input into the process,
- standard output (or stdout), which is used to write output from the process, and
- standard error (or stderr), which is used to produce error messages and avoid having them mixed up with actual output.

Consider a process representing an execution of echo, as in the example above. We can picture it (in isolation) as follows:



The question is, where do the (currently dangling) streams come from and go to? By default, when we execute a program and a resulting process is created, the associated streams are connected to the terminal. This means any input typed into the terminal by a user via the keyboard can be read by the process on standard input; any output produce by the process on standard output can be viewed by the user via the monitor. We already saw this above for example, where the command `echo "hello"` produced output: `echo` wrote `hello` to standard output, which we were able to see because standard output was connected to the terminal. However, we can, rather handily, *choose* where they are connected to via two important mechanisms outlined below.

1.3.1 Input and output redirection

Having just one stream for input and one for output might seem limiting: for example if a web-browser is used to open some web-page it must have to access numerous *different* resources (e.g., the HTML source code, images and so on), right? The solution provided is to allow each process to supplement the standard input and output streams with access to files; the process can open and close, read and write to files at will, vastly extending the range of ways it can interact with the environment.

To make the task easier, the OS manages streams and files used by a given process in the same way. A **file descriptor** [5], basically a number, is assigned to each: the standard input, output and error streams are numbered 0, 1 and 2, with open files numbered 3 and onward. Each time the process wants to do something related to a file descriptor, is asks the OS to do it via an appropriate system call: "open the file `A.txt` and give me a file descriptor for it" or "read data from file descriptor 0" or "write data to file descriptor 3" say. There is (at least) one major advantage to this approach: if they are managed the same way internally, it should seem natural that one of the standard streams can be connected to a file just as easily as it can the terminal. This is termed **redirection** [16]. If we want to connect the standard output stream of a process to a file, for example, the idea is that the OS pulls a slight-of-hand by replacing the entry for file descriptor 0 with one for the file. The process is unaware and still simply produces output using file descriptor 0, asking the OS to "write data to file descriptor 0". Behind the scenes however, the OS causes the output to end up somewhere else (i.e., in the file, not on the terminal). A simple example should make this clearer:

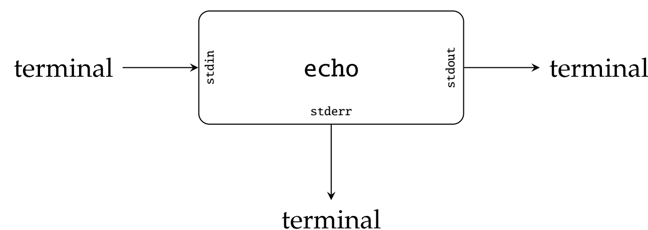
```
bash$ echo 'foo bar baz'
foo bar baz
bash$ echo 'foo bar baz' > A.txt
bash$ cat A.txt
foo bar baz
bash$ tr 'a' 'b' < A.txt
foo bbr bbz
bash$ tr 'a' 'b' < A.txt > B.txt
bash$ cat B.txt
foo bbr bbz
bash$
```

Within the various commands, < and > are used to control how redirection should work:

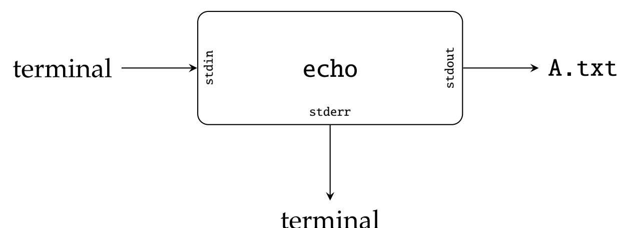
- The first group of three commands illustrate the difference between standard output being connected to the terminal versus a file. The addition of `> A.txt` in the second command should be read as "redirect standard output into the file `A.txt`".

  As a result, when the command is executed we no longer see the same output: this output is stored into `A.txt`, a fact then illustrated by the third command (which is used to show the content of `A.txt`.

  Diagrammatically for example, the first command in this group produces



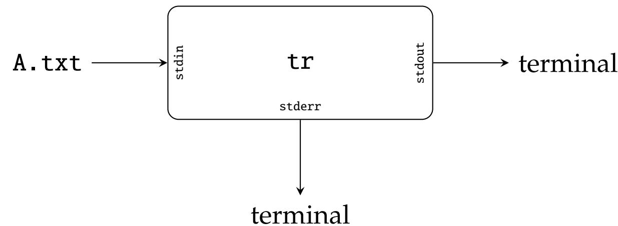
  while the second produces



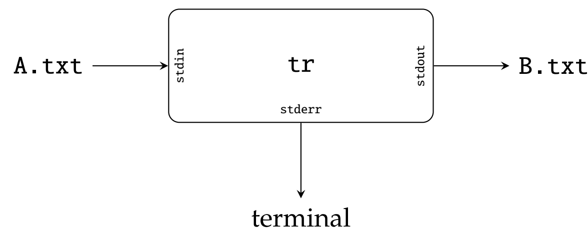
- The second group of three commands illustrate the difference between standard input being connected to the terminal versus a file. The addition of < `A.txt` in the first command should be read as "redirect standard input from the file `A.txt`".

  Now `tr` is fed input from `A.txt`, meaning we no longer need to type anything but do see the output (since standard output is still connected to the terminal). The second command uses both forms of redirection, so we get standard input redirected from `A.txt` and at the same time standard output redirected into `B.txt`. The third command shows the content of `B.txt`, which as expected matches what we saw directly via the terminal.

  Diagrammatically for example, the first command in this group produces



  while the second produces



There are various more advanced ways to extend this ability, although their use is less common in the examples presented. For completeness, just note that the following are possible:

- As shown above, input and output redirection impacts on the behaviour of standard input and output respectively. However, identifying the stream we want to redirect gives some more control. For example,

  `echo 'foo bar baz' 2> B.txt`

  means standard error is redirected into the file `B.txt` with standard output operating as normal, while

  `echo 'foo bar baz' &> C.txt`

  means both standard output *and* error are redirected into the file `C.txt`.

- By default, redirecting a stream into a given file overwrites the file content: the previous content lost. This behaviour can be changed by using

  `echo 'foo bar baz' >> D.txt`

  or

  `echo 'foo bar baz' 2>> E.txt`

  to mean append what appears from standard output or error to the end of files `D.txt` and `E.txt` respectively. Here, previous content in `D.txt` and `E.txt` is added to rather than overwritten.

1.3.2 Command pipelines

Revisiting the example above, the two commands

```
bash$ echo 'foo bar baz' > A.txt
bash$ tr 'a' 'b' < A.txt
foo bbr bbz
bash$
```

have a slightly annoying feature: the first command writes output into the file `A.txt` so it can be read as input by the second command, but then we probably just delete it. So if we already know we are going to delete `A.txt`, surely there is a way to avoid using it at all? BASH offers a neat solution to this question, which is termed a **command pipeline** [13]. In short, we can rewrite the two commands as one command pipeline

```
bash$ echo 'foo bar baz' | tr 'a' 'b'
foo bbr bbz
bash$
```

which produces the same output using two parts:

1. a left-hand part is represented by the command

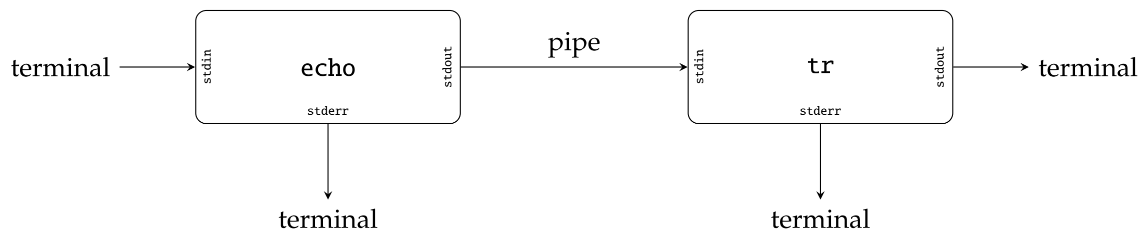
   `echo 'foo bar baz'`

   and

2. a right-hand part is represented by the command

   `tr 'a' 'b'`

When the single command pipeline is issued, each part creates a separate process that then execute in parallel (i.e., at the same time). Crucially, the standard output stream of the left-hand process is connected to the standard input stream of the right-hand process via a **pipe**, i.e., we have



meaning any output produced by `echo` on standard output can be read by `tr` on standard input.

2 A limited BASH command reference

2.1 alias

The `alias` command is usually a BASH built-in. The idea is to give a new name to an existing command, or an entire command pipeline: it literally creates an alias that we can later use as if it were a normal command we had typed in. When used with no arguments, `alias` prints a list of current aliases:

```
bash$ alias
alias cls='clear'
alias emacs='emacs -nw'
alias less='less -X'
alias ls='ls --color=auto --classify --human-readable'
alias md='mkdir'
alias rd='rmdir'
alias tidy='rm -f *~ .*~'
alias top='htop'
alias windows='xfreerdp /w:1024 /h:768 /d:uob /u:csdsp /v:staffremotedesktop.cse
.bris.ac.uk'
bash$
```

Notice for example that `ll` is not a real command, but an alias: BASH translates a use of `ll` into `ls -l`, so `ll` acts as a shorthand for `ls` with a particular option.

To create a new alias, we supply a name (or identifier) and a value that details what the name should mean (i.e., what it should translate into):

```
bash$ alias oneperline="fold -1"
bash$ cat > A.txt
abcd
efgh
bash$ oneperline A.txt
a
b
c
d
e
f
g
h
bash$ cat A.txt | oneperline
a
b
c
d
```

```
e
f
g
h
bash$
```

In this example, we create a new alias whose name is `oneperline` and whose meaning is `fold -1`: every time BASH sees the command `oneperline`, it translates it into `fold -1`. As the subsequent uses show, `oneperline` works just like a normal command with respect to standard input and output.

2.2 cat

The `cat` command (short for "concatenate") is one of the most simple listed here, but also one of the most useful. The command has a modest set of features: given a list of file name arguments, `cat` reads each file in turn and copies the content to standard output. Since each file is read in turn, the content on standard output is essentially a concatenation of all the files. Perhaps even more usefully, when no file names are specified `cat` reads from standard input (i.e., takes input from the user). Crucially, these features enable us to use `cat` to

- provide input to other commands in a command pipeline, or
- create files whose content is supplied by user input.

For example, imagine we want to construct a file called `A.txt` whose content is supplied by user input. We can invoke `cat` with no file name arguments so input is read from standard input, and then redirect standard output at the file:

```
bash$ cat > A.txt
a
b
c
d
bash$
```

To mark the end of the input, the user types the EOF character (i.e., Ctrl-D) which is not shown above. Now imagine we want to check that `A.txt` has the expected content; `cat` again supplies a solution. We can invoke `cat` with one file name argument, i.e., `A.txt`, and have it copy the file content to standard output for us to see:

```
bash$ cat A.txt
a
b
c
d
bash$
```

Now imagine we make another file called `B.txt` in much the same way as above:

```
bash$ cat > B.txt
e
f
g
h
bash$
```

Invoking `cat` now with two file name arguments, i.e., both `A.txt` and `B.txt`, shows how their content is concatenated and copied to standard output:

```
bash$ cat A.txt B.txt
a
b
c
d
e
f
g
h
bash$
```

2.3 umask and chmod

New file system entries are created with a default set of permissions controlled by a mask. The mask is subtractive: it disables, i.e., takes away, specific permission types from a starting point where every permission type is enabled. The `umask` command is used to set and inspect this mask:

```
bash$ umask 022
bash$ umask -S
u=rwx,g=rx,o=rx
bash$
```

The first command sets the mask using the shorthand `022`; the three numbers in this shorthand should be interpreted as follows:

- 0 means no permission types for the user permission set should be disabled,
- 2 means the writable permission type for the group permission set should be disabled, and
- 2 means the writable permission type for the other permission set should be disabled.

With the `-S` option, `umask` reports this in an easier to read format by telling us which permissions a new file will have: notice that the writable permission type is missing from the group and other permission sets, whereas no permission types are missing from the user permission set. We can trial this using an example:

```
bash$ umask 022
bash$ umask -S
u=rwx,g=rx,o=rx
bash$ cat > A.txt
a
b
c
d
bash$ ls -l A.txt
-rw-r--r-- 1 page page 8 Jun 23 09:07 A.txt
bash$
```

The final output of `ls` confirms the discussion above: the new file `A.txt` is not writable by either members of the group, or by other users.

Of course, we can change this behaviour to whatever suits us. Imagine we are paranoid, and want a default that prevents anyone other than the owner of a new file from reading it:

```
bash$ umask 077
bash$ umask -S
u=rwx,g=,o=
bash$ cat > B.txt
e
f
g
h
bash$ ls -l B.txt
-rw------- 1 page page 8 Jun 23 09:10 B.txt
bash$
```

The new mask `077` implies the following meaning

- 0 means no permission types for the user permission set should be disabled,
- 7 means all permission type for the group permission set should be disabled, and
- 7 means all permission type for the other permission set should be disabled

so when `B.txt` is created, the effect is that the new file cannot be read by anyone other than the owner.

Clearly it can be useful to change the permissions of a file system entry after it is created; this is the job of `chmod`. The basic idea is that we specify the name of the file system entry as an argument, along with the change to the permissions we want to make. This change can be specified in two ways. To make an update to the permissions, we can use `chmod` as follows:

```
bash$ cat > C.txt
i
j
k
l
bash$ ls -l C.txt
-rw-rw-r-- 1 page page 8 Jun 23 09:13 C.txt
bash$ chmod g+w C.txt
bash$ ls -l C.txt
-rw-rw-r-- 1 page page 8 Jun 23 09:13 C.txt
bash$ chmod o-r C.txt
bash$ ls -l C.txt
-rw-rw---- 1 page page 8 Jun 23 09:13 C.txt
bash$
```

The two uses of `chmod` specifies the changes

- `g+w`, read as "add the write permission type to the group permission set", and
- `o-r`, read as "remove the read permission type from the other permission set"

where checking the result with `ls` confirms the changes worked as expected in both cases. However, this style only allows us to specify one change per invocation of `chmod`; for several of changes, as above, this can quickly get tedious. The good news is that we can describe the permission set we want to end up with using a similar shorthand as above:

```
bash$ cat > D.txt
m
n
o
p
bash$ ls -l D.txt
-rw-rw-r-- 1 page page 8 Jun 23 09:17 D.txt
bash$ chmod 777 D.txt
bash$ ls -l D.txt
-rwxrwxrwx 1 page page 8 Jun 23 09:17 D.txt*
bash$ chmod 664 D.txt
bash$ ls -l D.txt
-rw-rw-r-- 1 page page 8 Jun 23 09:17 D.txt
bash$
```

This time, the mask is additive: it enables, i.e., adds, specific permission types from a starting point where every permission type is disabled. In the first case, the mask `777` implies the following meaning

- 7 means all permission types for the user permission set should be enabled,
- 7 means all permission type for the group permission set should be enabled, and
- 7 means all permission type for the other permission set should be enabled

That is, we want all of the readable, writable and executable permission types enabled in all of the user, group and other permission sets. In the second case the mask `644` means

- 6 means the readable and writable permission type for the user permission set should be enabled,
- 4 means the readable permission type for the group permission set should be enabled, and
- 4 means the readable permission type for the other permission set should be enabled

That is, we want the readable and writable permission types enabled in the user permission set, but only the readable permission type enabled in the group and other permission sets.

While `chmod` can alter permissions, `chgrp` and `chown` can be used to alter the ownership of a file system entry. For example, the `chgrp` command takes a group name and a file system entry as arguments and sets the group accordingly:

```
bash$ ls -l D.txt
-rw-rw-r-- 1 page page 8 Jun 23 09:17 D.txt
bash$ chgrp dialout D.txt
bash$ ls -l D.txt
-rw-rw-r-- 1 page dialout 8 Jun 23 09:17 D.txt
bash$ chgrp plugdev D.txt
bash$ ls -l D.txt
-rw-rw-r-- 1 page plugdev 8 Jun 23 09:17 D.txt
bash$
```

Of course, the combination of user name and group name must be valid in the sense that the user must actually be a member of the specified group.

Note that `chmod`, `chgrp` and `chown` all recognise the `-R` option; this instructs the command to recursively process the specified file system entry, and can be useful when changing the ownership or permissions of an entire directory tree for example.

2.4 cut

The `cut` command is roughly the opposite of `paste`. There are a number of ways to use `cut` but the general idea is to consider each line of input, by default read from standard input, as a number of columns (or fields) separated by a delimiter character; use of `cut` constructs output by extracting specific columns from the input, writing them by default to standard output.

Consider creating an example file called `A.txt`; the delimiter character in this case is a comma (i.e., ',') meaning `A.txt` represents a so-called **Comma Separated Value (CSV)** file:

```
bash$ cat > A.txt
a,b,c
d,e,f
g,h,i
bash$
```

There are three lines, and each line has three fields; imagine we want to extract just the first and second fields. We can invoke `cut` as follows:

```
bash$ cut -d ',' -f 1,2 A.txt
a,b
d,e
g,h
bash$ cat A.txt | cut -d ',' -f 1,2
a,b
d,e
g,h
bash$
```

The `-d` option sets the delimiter character to match the one used in the file, and the `-f` option specifies the fields we want in the output; note that fields are counted from one rather than zero. Using other delimiter characters is obviously possible:

```
bash$ date
Thu 22 Jun 11:21:40 BST 2017
bash$ date | cut -d ' ' -f 1
Thu
bash$ date | cut -d ' ' -f 2
22
bash$ date | cut -d ' ' -f 3
Jun
bash$ date | cut -d ' ' -f 4
11:22:40
bash$ date | cut -d ' ' -f 5
BST
bash$ date | cut -d ' ' -f 6
2017
bash$
```

However, in this case specifying the delimiter and character is perhaps overkill since `date` always uses the same format (e.g., the day field is always the first three characters) unless told otherwise. Rather than rely on the use of a delimiter character, it can be useful to have `cut` extract fields based on character position; this is achieved by using the the `-c` option to specify the position to start and finish at:

```
bash$ date | cut -c 1-3
Thu
bash$ date | cut -c 5-7
22
bash$
```

Again note that the character number is counted from one rather than zero; if we leave out the finish position, `cut` just produces the rest of the line. Returning to our original example, we could therefore extract the second field with an alternative usage of `cut`

```
bash$ cat A.txt | cut -c 3-3
b
e
h
bash$
```

or, by leaving out the finish position, get the second and third fields:

```
bash$ cat A.txt | cut -c 3-
b,c
e,f
h,i
bash$
```

2.5 diff

Imagine we want to know if the content of two files is identical or not; we might also want to know where the content differs if it does so. Perhaps you are working on a program or document with someone: they make a change, and you want to know where that change is in the new version relative to the old version you have got. For short text files we might just inspect the files visually, but for longer text files or binary files this is problematic. Fortunately two commands, namely `diff` and `cmp`, can help; they operate on textual and binary content respectively.

Both commands take two file name arguments, and compare their content. The `-s` and `-q` options can be used to silence `cmp` and `diff` respectively, meaning they set the exit code to 0 if the files are identical, or 1 if there are difference found:

```
bash$ cat > A.txt
a
b
c
d
```

```
bash$ cat > B.txt
a
e
c
f
bash$ diff -q A.txt A.txt
bash$ echo ""

bash$ diff -q A.txt B.txt
Files A.txt and B.txt differ
bash$ echo ""

bash$ cmp  -s A.txt A.txt
bash$ echo ""

bash$ cmp  -s A.txt B.txt
bash$ echo ""

bash$
```

However, some information about what and where differences were found is often very useful. This is particularly true when the file content is textual and we use `diff`. Consider an example were the `-q` option is removed:

```
bash$ diff A.txt B.txt
2c2
< b
---
> e
4c4
< d
---
> f
bash$
```

The cryptic tokens `2c2` and `4c4` tell us that two differences between `A.txt` and `B.txt` were found. Each token can be broken into three components:

1. a line number in the first file,
2. a character that describes what type of difference was found,
3. a line number in the second file.

Consider the first token, i.e., `2c2`, as an example: the difference type is `c`, meaning "change"; line two of `A.txt` has been changed and created a difference in line two of `B.txt`. Looking again at the output, each token is followed the actual text, i.e. the lines from `A.tex` and `B.tex` which have been changed.

Here are some more examples:

```
bash$ cat > B.txt
a
b
e
c
d
bash$ diff A.txt B.txt
2a3
> e
bash$
```

This time the difference type is `a`, meaning "addition"; relative to line two of `A.txt`, there is an addition at line three of `B.txt`. We might also get the opposite:

```
bash$ cat > B.txt
a
b
d
bash$ diff A.txt B.txt
3d2
< c
bash$
```

where now the change type is `a`, meaning "deletion".

2.6 du

The `du` command (short for "disk usage") allows a direct way to inspect the size of entries in the file system. Invoked with no arguments, `du` starts in the working directory and then recursively inspects all files and directories from; it prints the cumulative size of each directory to standard output:

```
28      ./fonts/Type1
8       ./fonts/100dpi
60      ./fonts
104     ./Xsession.d
536     .
bash$ du -k | tail -n 5
28      ./fonts/Type1
8       ./fonts/100dpi
60      ./fonts
104     ./Xsession.d
536     .
bash$ du -h | tail -n 5
28K     ./fonts/Type1
8.0K    ./fonts/100dpi
60K     ./fonts
104K    ./Xsession.d
536K    .
bash$ du -c | tail -n 5
8       ./fonts/100dpi
60      ./fonts
104     ./Xsession.d
536     .
536     total
bash$
```

The four uses of du demonstrate different options. Note that the output is restricted to a few lines (using `last`) since there is a lot of it, and that

- the `-k` option reports sizes in kilobytes,
- the `-h` option reports sizes in "human readable" units,
- the `-c` option includes a total size at the end of the output.

Including one or more path names as arguments instructs du to process them, in turn, rather than the working directory. When the path name specifies a file, du simply prints the size of that file; when the path name specifies a directory, du processes it recursively as above. For example:

```
bash$ du -hc /etc/ssh/
344K    /etc/ssh/
344K    total
bash$ du -hc /etc/ssh/*.pub
4.0K    /etc/ssh/ssh_host_dsa_key.pub
4.0K    /etc/ssh/ssh_host_ecdsa_key.pub
4.0K    /etc/ssh/ssh_host_ed25519_key.pub
4.0K    /etc/ssh/ssh_host_rsa_key.pub
16K     total
bash$
```

Notice that by inspecting the output and identifying the largest entries, it is easy to find out which ones are using the most space. This can be useful when trying to clean up in the event that you run out of disk space: removing the largest entries (which are not useful) will give the most benefit.

2.7 echo

The echo command has the modest goal of producing output; it takes a string argument and prints the string, potentially after some minor translation, to standard output.

```
bash$ echo 'hello world'
hello world
bash$
```

Using the `-e` and `-E` options, one can enable or disable the translation process which converts escaped characters within the string; by default this behaviour is disabled. For example, imagine the string argument contains a backspace character Ctrl-H escaped as '`\b`':

```
bash$ echo -e 'hello\b world'
hell world
bash$ echo -E 'hello\b world'
hello\b world
bash$
```

In the first case, where translation is enabled, when the backspace is printed on standard output it deletes the previous character (i.e., the 'o' character at the end of "hello"). However, where translation is disabled in the second case, the backspace is printed on standard output as-is (i.e. as the escaped character '`\b`').

Finally, note that by default echo prints an EOL character at the end of the string. This behaviour can be turned off by using the `-n` option, and can be useful when producing binary output for example:

```
bash$ echo -n 'hello world'
hello worldbash$
```

2.8 fold

The `fold` command is an example of something very simple, and perhaps seemingly useless, but which turns out to be very useful in solving all sorts of other problems. The command reads lines of input from standard input (or the content of a file named as an argument if provided) and "wraps" them so they fit a certain length; the output is written to standard output. You can think of this as similar to what happens when you use a word processor: when you write a very long line, it typically moves you onto the next line automatically.

The `-w` option sets the maximum length of each line, i.e., where to wrap each line; the default is 80 characters. As such an example shows how `fold` works:

```
bash$ cat > A.txt
ab cd ef gh
ijkl
mnop qr
st
bash$ fold -w 4 A.txt
ab c
d ef
 gh
ijkl
mnop
 qr
st
bash$ cat A.txt | fold -w 4
ab c
d ef
 gh
ijkl
mnop
 qr
st
bash$
```

A specifically important application of `fold` stems from use of `-w 1` as an option: this splits a 1-line input, of say n characters, into an n-line output each of which has just one character. Since many other commands are line-oriented, i.e., process each line of input in turn, this can allow them to operate on the characters within a single line.

Finally, when processing text it can be useful to only wrap a line at a point where there is a space rather than at a specific length; this prevents splitting words over two lines for example. This is possible using the `-s` option. Returning to the example above, notice how this alters the output:

```
bash$ cat A.txt | fold -w 4 -s
ab
cd
ef
gh
ijkl
mnop
 qr
st
bash$
```

The first line is now split at the space between "ab" and "cd" because, with the same line length of four, we saw above that "cd" was originally split over two lines.

2.9 grep

The `grep` command is possibly the greatest invention ever, bar perhaps the wheel and sliced bread; used as a verb, "to grep" is something you might actually hear in conversations. Perhaps the best way to understand what `grep` does is to start by thinking of it as a stream filter: it reads lines of input, checks whether they match some pattern (described using a regular expression), and only outputs lines that match. Consider a single example: first we construct a file called `A.txt`, then try to filter out all the lines that contain an 'a' character:

```
bash$ cat > A.txt
a
aba
cac
add
eea
f
bash$ grep a A.txt
a
aba
cac
add
eea
bash$ cat A.txt | grep a
a
aba
cac
```

```
add
eea
bash$
```

Notice that by default grep reads input from standard input and writes output to standard output; by supplying a file name argument we can have grep read the file content instead. Of course, more complicated regular expressions are possible. Imagine we try to filter the same file using regular expressions that match

1. lines containing characters from the set [af], i.e., the characters 'a' and 'f',
2. lines that start with the character 'a',
3. lines containing the sequence 'a', then any character then another 'a'.

The grep usage and resulting output is as follows:

```
bash$ cat A.txt | grep [af]
a
aba
cac
add
eea
f
bash$ cat A.txt | grep
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
bash$ cat A.txt | grep a.a
aba
bash$
```

One can also turn the filter the opposite way and have grep only output a line if it does not match the pattern. This is achieved by using the -v option:

```
bash$ cat A.txt | grep -v [af]
bash$ cat A.txt | grep -v
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
bash$ cat A.txt | grep -v a.a
a
cac
add
eea
f
bash$
```

It is common to execute grep several times within a single command pipeline in order to narrow down (or "thin") the input until it contains only those lines which are of interest. Imagine we want to end up with just the line containing "cac". Among many options for this particular file, one way to filter out all other lines would be to use

```
bash$ cat A.txt | grep .a.
cac
bash$
```

where the pattern selects all lines that contain any character, then an 'a' then any character. Alternatively we could do the same thing with the following command pipeline:

```
bash$ cat A.txt | grep a | grep -v a$ | grep -v
Usage: grep [OPTION]... PATTERN [FILE]...
Try 'grep --help' for more information.
bash$
```

For this particular file this works by first selecting all lines containing 'a', then eliminating all of those lines ending with 'a' and finally eliminating any remaining lines that start with 'a'.

Finally, grep can act as more than just a stream filter. When provided with one or more file name arguments, grep acts to search those files for the given pattern. Imagine we are writing a C program and litter the source code with "todo list" style comments that indicate where work is needed. We might want to see where in a particular file such comments exist, or gather a list of files that include such comments; in both cases grep provides a solution. Imagine we write three source code files:

```
bash$ cat > B.c
/* TODO: finish main function */
int main( int argc, char* argv[] ) {
  return 0;
}
bash$ cat > C.c
/* TODO: fix foo function, it returns the wrong value */
int foo( int x, int y ) {
  return ( x < y ) ? ( x ) : ( y );
}
bash$ cat > D.c
```

```
int bar( int x, int y ) {
  return ( x > y ) ? ( x ) : ( y );
}
bash$
```

Providing the same file names to `grep` as arguments, we can search for the string `TODO` as follows:

```
bash$ grep TODO B.c C.c D.c
B.c:/* TODO: finish main function */
C.c:/* TODO: fix foo function, it returns the wrong value */
bash$
```

Notice that `grep` has produced the file names and the lines within those files which match the pattern. If you just want to know the file names which match the pattern, use of the `-l` option will suppress output of the actual lines:

```
bash$ grep -l TODO B.c C.c D.c
B.c
C.c
bash$
```

2.10 bzip2, gzip and zip

Most UNIX-based operating systems offer a wide range of commands that perform compression and decompression of data. Their common role is simple: some input data is compressed into a smaller version (e.g., in order to communicate or store it) which can, at a later date, be decompressed to recover the original. Example commands include:

- `gzip` (and `gunzip`),
- `bzip2` (and `bunzip2`), and
- `zip` (and `unzip`).

The difference between these examples is (very roughly) how effectively they compress data (i.e., how small an output they produce given some input), and how efficiently they perform said compression. However, note that `zip` represents a combination of archival and compression tasks: it can collect many files together into an archive then compress the result. In contrast, `gzip` and `bzip2` perform compression only; they rely on a separate command such as `tar` to deal with archives and hence operate with more than one file.

The commands are all used in roughly the same way, and have similar sets of options. Since the exact command does not matter in most cases, we focus only on `bzip2` and `bunzip2` since they represent a more modern solution. By default `bzip2` (resp. `bunzip2`) reads input from standard input, compresses (resp. decompresses) it and copies the result to standard output. Alternatively, a file name argument can be provided: the file content is compressed (resp. decompressed) and the result is written to a new file. The new file is named by taking the original file name and adding (resp. removing) the `.bz2` extension.

Consider an example where we construct a file called `A.txt` and then compress it using `bzip2`:

```
bash$ cat > A.txt
a
a
a
a
b
b
b
b
bash$ ls -l A.txt
-rw-rw-r-- 1 page page 16 Jun 22 09:59 A.txt
bash$ bzip2 -z A.txt
bash$ ls -l A.txt.bz2
-rw-rw-r-- 1 page page 43 Jun 22 09:59 A.txt.bz2
bash$
```

The `-z` option instructs the command to compress, but this can be left out since `bzip2` compresses the input by default. Either way, the result for this small example input is a bit disappointing; in terms of compression, we have actually made `A.txt.bz2` larger! However, for larger and less contrived inputs you can hopefully believe that `bzip2` is more effective. We can test the integrity of `A.txt.bz2` (i.e., whether the file can be decompressed correctly) using the `-t` option:

```
bash$ bzip2 -t A.txt.bz2
bash$ echo ""

bash$
```

In this case the lack of output and the exit code indicate success.

The command bunzip2 is actually just a shorthand for use of bzip2 command with the -d option; this instructs bunzip2 to decompress the input rather than compress it:

```
bash$ ls -l A.txt.bz2
-rw-rw-r-- 1 page page 43 Jun 22 09:59 A.txt.bz2
bash$ bzip2 -d A.txt.bz2
bash$ ls -l A.txt
-rw-rw-r-- 1 page page 16 Jun 22 09:59 A.txt
bash$
```

Note that a slew of alternates to commands such as cat exist that will accept compressed input. For example, given a file compressed with gzip, zcat will copy the decompressed file content to standard output:

```
bash$ cat > B.txt
a
b
c
d
bash$ gzip B.txt
bash$ zcat B.txt.gz
a
b
c
d
bash$
```

Usually a similar result can be achieved just by using the right options for the command; for example, bzip2 and gzip will both produce output on standard output (given a file as input) if used with the -c option.

2.11 head and tail

The purpose of the head (resp. tail) command is to output the first (resp. last) part of some input. We can specify the amount of input to retain in various ways: one might want to output the first *n* lines for example, or the first *n* bytes. This is a simple remit, but deceptively powerful. Given A.txt, a short example file

```
bash$ cat > A.txt
a
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
bash$
```

consider some examples of specifying the number of lines to output using the -n option:

```
h
i
j
k
l
m
n
o
p
bash$ cat A.txt | tail
g
h
i
j
k
l
m
n
o
p
bash$ tail -n 2 A.txt
o
p
bash$
```

Notice that the input taken from a file named as an argument, or from standard input by default, and the output is written to standard output. When -n is used, head and tail output the first and last two lines of A.txt respectively; when it is not used, the default is ten lines.

In the case of tail, when the number of lines is preceded by a plus sign it signals a slightly different behaviour. For example, if used as follows

```
bash$ cat A.txt | tail -n +2
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
bash$
```

tail starts at the second line in A.txt meaning the first line containing 'a' is skipped. By using a combination of head and tail, we can therefore split a file into parts at a given line; to split the same A.txt at the second line for example, we could use

```
bash$ cat A.txt | head -n  1
a
bash$ cat A.txt | tail -n +2
b
c
d
e
f
g
h
i
j
k
l
m
n
o
p
bash$
```

Or, to extract a specific number of lines from the middle of the file, we could combine head and tail in a single command pipeline such as:

```
bash$ cat A.txt | tail -n +3 | head -n 3
c
d
e
bash$
```

To specify a number of bytes instead of a number of lines, we can use the -c option instead of -n. By first making a new file called B.txt

```
bash$ cat > B.txt
abcd
efgh
ijkl
mnop
bash$
```

the following examples demonstrate how this option behaves:

```
bash$ cat B.txt | head -c 10
abcd
efgh
bash$ cat B.txt | tail -c 10
ijkl
mnop
bash$
```

At first the output seems odd. However, remember that EOL characters are included: the output from head, for example, includes eight printable characters plus two EOL characters, making ten in total.

2.12 ls

The ls command (short for "list") allows inspection of file system entries: it will list information about the path names given as arguments, or the working directory if none are provided. Imagine we are interested in contents of the root directory of the file system. We might change directory there and issue an ls command to list the content:

```
bash$ cd /
bash$ ls
bin/    dev/        initrd.img.old@  lost+found/  proc/  snap/  usr/
boot/   etc/        lib/             media/       root/  srv/   var/
cdrom/  home/       lib32/           mnt/         run/   sys/   vmlinuz@
core    initrd.img@ lib64/           opt/         sbin/  tmp/   vmlinuz.old@
bash$
```

By default `ls` lists the names of entries in the new working directory, writing to standard output; at the moment it is not clear if these entries are directories or files, we only get their names. The `ls` command permits a huge range of options, but a select few are perhaps the most important. The example

```
lrwxrwxrwx   1 root root   32 Jun  7 12:12 initrd.img.old -> boot/initrd.img-4.4
.0-79-generic
drwxr-xr-x  27 root root 4.0K Jun 20 07:39 lib/
drwxr-xr-x   2 root root 4.0K Jun 20 07:39 lib32/
drwxr-xr-x   2 root root 4.0K Jun 20 07:39 lib64/
drwx------   2 root root  16K Oct 31  2014 lost+found/
drwxr-xr-x   3 root root 4.0K Nov  4  2014 media/
drwxr-xr-x   2 root root 4.0K May  3  2016 mnt/
drwxr-xr-x  18 root root 4.0K Feb 28 12:20 opt/
dr-xr-xr-x 305 root root    0 Jun 21 16:48 proc/
drwx------  18 root root 4.0K Feb 22 13:52 root/
drwxr-xr-x  31 root root 1020 Jun 22 07:36 run/
drwxr-xr-x   2 root root  12K Jun 20 07:39 sbin/
drwxr-xr-x   2 root root 4.0K Apr 19  2016 snap/
drwxr-xr-x   2 root root 4.0K Jul 22  2014 srv/
dr-xr-xr-x  13 root root    0 Jun 21 16:48 sys/
drwxrwxrwt  16 root root 156K Jun 22 12:52 tmp/
drwxr-xr-x  16 root root 4.0K May  3  2016 usr/
drwxr-xr-x  14 root root 4.0K May  3  2016 var/
lrwxrwxrwx   1 root root   29 Jun 20 07:40 vmlinuz -> boot/vmlinuz-4.4.0-81-gene
ric
lrwxrwxrwx   1 root root   29 Jun  7 12:12 vmlinuz.old -> boot/vmlinuz-4.4.0-79-
generic
bash$
```

allows us to describe them as follows:

- The `-1` option instructs `ls` to format the list using one entry per-line. This can be useful if the output of `ls` is used within a line-oriented command pipeline.
- The `-a` option instructs `ls` to show all entries including, for example, those hidden by default. In the example, as well as the original entries we now get the special current directory and parent directory entries (i.e., `.` and `..`).
- The `-l` option instructs `ls` to produce a "long" listing which includes a range of details about each entry. In the example, the columns of output detail

  1. the entry type (e.g., file or directory) and permissions,
  2. the number of links to the entry,
  3. the user who owns the entry,
  4. the group which owns the entry,
  5. the entry size (in bytes),
  6. the modification timestamp for the entry, and
  7. the entry name.

  The summary line at the beginning of the output gives the total size of all the entries listed.

This is perhaps obvious, but although we have so far used `ls` in the default mode to list entries in the working directory, we can also specify absolute or relative paths to other file system entries:

```
bash$ ls -l  /etc/passwd
-rw-r--r-- 1 root root 2.5K Jun 13  2016 /etc/passwd
bash$ ls -l  /etc/ssh/
total 340K
-rw-r--r-- 1 root root 294K Apr 16  2016 moduli
-rw-r--r-- 1 root root 1.8K Apr 16  2016 ssh_config
-rw------- 1 root root  668 May  5  2015 ssh_host_dsa_key
-rw-r--r-- 1 root root  603 May  5  2015 ssh_host_dsa_key.pub
-rw------- 1 root root  227 May  5  2015 ssh_host_ecdsa_key
-rw-r--r-- 1 root root  175 May  5  2015 ssh_host_ecdsa_key.pub
-rw------- 1 root root  399 May  5  2015 ssh_host_ed25519_key
-rw-r--r-- 1 root root   95 May  5  2015 ssh_host_ed25519_key.pub
-rw------- 1 root root 1.7K May  5  2015 ssh_host_rsa_key
-rw-r--r-- 1 root root  395 May  5  2015 ssh_host_rsa_key.pub
-rw-r--r-- 1 root root 2.5K May  3  2016 sshd_config
```

```
-rw-r--r-- 1 root root 2.5K May  5  2015 sshd_config~
bash$ ls -ld /etc/ssh/
drwxr-xr-x 2 root root 4.0K May 15 15:29 /etc/ssh//
bash$
```

Note that in this context, the `-d` option is useful: if the file system entry we specify is a directory, it tells `ls` we are interested in the directory itself rather than the contents. The result above is that with `-d`, we can inspect the permissions and so on for `/etc/ssh/` but without it `ls` lists the contents of `/etc/ssh/`.

Finally, a note on permissions: imagine we create a directory called `A/` using `mkdir` then create a file in it called `B.txt`. Changing the permissions of both file and directory using `chmod` demonstrates some important facts about how we access to them:

```
bash$ mkdir -p A/
bash$ cat > A/B.txt
a
b
c
d
bash$ ls -ld A/
drwxrwxr-x 2 page page 4.0K Jun 22 13:00 A//
bash$ chmod 300 A/
bash$ ls -ld A/
d-wx------ 2 page page 4.0K Jun 22 13:00 A//
bash$ ls -l  A/
ls: cannot open directory 'A/': Permission denied
bash$ chmod 700 A/
bash$ ls -ld A/
drwx------ 2 page page 4.0K Jun 22 13:00 A//
bash$ ls -l  A/
total 4.0K
-rw-rw-r-- 1 page page 8 Jun 22 13:01 B.txt
bash$
```

First we create `A/` and `A/B.txt`, and then change the permissions of `A` so the user permission set includes only the write and execute permission types; as a result, the first time we try to use `ls` to list the contents of `A/`, the command fails. After altering the permissions of `A/` again to now include the read permission type, the second attempt to list the contents of `A/` succeeds.

2.13 od

The `od` command (short for "octal dump") is a brilliant tool. The main use-case is visualisation of binary data. The idea is that we provide some input (either by supplying an optional file name argument or on standard input by default), and `od` produces an unambiguous representation on standard output. This is a fancy way to say that it inspects the binary input and produces the corresponding "human readable" output that is not complicated by the existence of non-printable characters and so on.

When used with no options, the output from `od` is somewhat difficult to interpret:

```
bash$ cat > A.txt
a
b
c
d
bash$ od A.txt
0000000 005141 005142 005143 005144
0000010
bash$ cat A.txt | od
0000000 005141 005142 005143 005144
0000010
bash$
```

The left-hand column is an index (or address); starting at each index is some content, which is listed in columns to the right. By default however, both the index and content are displayed in octal. Rather than try to interpret this, a better approach is typically to control `od` using one or both of two options:

- `-A` allows us to change the index format using a 1-character code to specify the base (or radix):
  - `o` for octal, which is the default,
  - `d` for decimal, or
  - `x` for hexadecimal.
- `-t` allows us to change the content format using a 1- or 2-character code to specify the base (or radix) *and* size, e.g.,
  - `c` for ASCII characters,
  - `a` for "named" ASCII characters,

- o for octal integers,
- u for unsigned decimal integers,x
- d for signed decimal integers,
- x for hexadecimal integers,
- u1 for 1-byte unsigned decimal integers,
- d2 for 2-byte signed decimal integers, or
- x4 for 4-byte hexadecimal integers.

Using decimal for the index *and* the content, `A.txt` now makes more sense for example:

```
bash$ cat A.txt | od -Ad -td1
0000000   97   10   98   10   99   10  100   10
0000008
bash$
```

This shows that starting at index 0 (through to 7, since there is no content on the second line starting at index 8) the content includes the eight 8-bit unsigned decimal values 97, 10, 98, 10, 99, 10, 100 and 10. This should not be surprising since, for example, 97 is the ASCII value for the character 'a' and 10 is the ASCII value for EOL. Switching to hexadecimal for the content, i.e.,

```
bash$ cat A.txt | od -Ad -tx1
0000000 61 0a 62 0a 63 0a 64 0a
0000008
bash$
```

should still make sense in that $61_{(16)} = 91_{(10)}$, and $0A_{(16)} = 10_{(10)}$ for example.

Two other options are often useful: `-w` and `-N` respectively specify the number of entries in the content to display on each line and in total. In a final example using `A.txt`, these are used to inspect only the first four bytes with two bytes on each line:

```
bash$ cat A.txt | od -Ad -ta -w2 -N4
0000000   a  nl
0000002   b  nl
0000004
bash$
```

This example also uses a "named" ASCII for the content format: each character is translated into a name (e.g., EOL into `nl`) rather than a character code as it would be in the alternative

```
bash$ cat A.txt | od -Ad -tc -w2 -N4
0000000   a  \n
0000002   b  \n
0000004
bash$
```

2.14 paste

The `paste` command is roughly the opposite of `cut`; it takes a list of n file name arguments as input, and pastes the files content as columns along side each other to form output. That is, the i-th line of the output is constructed by reading the i-th lines of each n inputs and placing them in order along side each other. The end result is perhaps explained more clearly by example:

```
bash$ cat > A.txt
a
b
c
d
bash$ cat > B.txt
e
f
g
h
bash$ paste A.txt B.txt
a	e
b	f
c	g
d	h
bash$
```

By default, as above, the output is written to standard output; note that the tab character is used to separate the parts of each line. One can alter this behaviour using the `-d` option to specify an alternative delimiter character; for example to separate the parts using a comma one might use:

```
bash$ paste -d ',' A.txt B.txt
a,e
b,f
c,g
d,h
bash$
```

It is worth pointing out that `paste` is similar to other commands in that it can accept input from standard input rather than from the content of files. In this case however, there is only one standard input whereas above we specified multiple file name arguments. By using the `-s` flag, `paste` uses just one input source: the i-th line of the output is again constructed by reading lines from the input and placing them in order along side each other. You can think of this as serialising the multi-line input into a single-line output:

```
bash$ paste -s A.txt
a       b       c       d
bash$ cat A.txt | paste -s
a       b       c       d
bash$
```

2.15 sort

It should come as no surprise that the `sort` command reads lines of input and sorts them to produce output. The sorting procedure is controlled by the sorting "key". The idea is to imagine each line split into a number of fields by a delimiter (e.g., a space or comma); sorting the input essentially means comparing the field in each line indicated by the sorting key.

The input is taken from a file named as an argument, or from standard input by default, and the output is written to standard output. To demonstrate the command we first need to construct `A.txt`, a short example file:

```
bash$ cat > A.txt
a t 300
A t 300
b s 456
h m 74
d q 0
g n 5999
f o 623
c r 13
bash$
```

By supplying the file to sort without any options, the whole line is used as the sorting key and so the output is:

```
bash$ sort A.txt
A t 300
a t 300
b s 456
c r 13
d q 0
f o 623
g n 5999
h m 74
bash$ cat A.txt | sort
A t 300
a t 300
b s 456
c r 13
d q 0
f o 623
g n 5999
h m 74
bash$
```

However, imagine we want to sort the file using the second field as the sorting key. This is easy: we simply use the `-t` option to specify that the delimiter character in this case is a space, and the `-k` option to specify that the second column should be used as the sorting key:

```
bash$ cat A.txt | sort -t ' ' -k 2
h m 74
g n 5999
f o 623
d q 0
c r 13
b s 456
A t 300
a t 300
bash$
```

Trying the same trick with the third field probably does not produce the output one might expect intuitively:

```
bash$ cat A.txt | sort -t ' ' -k 3
d q 0
c r 13
A t 300
a t 300
b s 456
g n 5999
f o 623
h m 74
bash$
```

The reason is simple: unless otherwise specified, the sorting procedure is based on alphabetic order. In order to use the third field as the sorting key, we need a numerical order and hence the `-n` option:

```
bash$ cat A.txt | sort -t ' ' -k 3 -n
d q 0
c r 13
h m 74
A t 300
a t 300
b s 456
f o 623
g n 5999
bash$
```

Controlling `sort` via numerous other options is possible: two of the more useful are `-r`, which reverses the sort order (i.e., backwards rather than forwards, or decreasing rather than increasing), and `-c`, which checks if the input is sorted (rather than actually sorting it). These options are demonstrated in the following example:

```
bash$ cat A.txt |        sort -r
h m 74
g n 5999
f o 623
d q 0
c r 13
b s 456
a t 300
A t 300
bash$ cat A.txt |        sort -c
sort: -:2: disorder: A t 300
bash$ echo ""

bash$ cat A.txt | sort | sort -c
bash$ echo ""

bash$
```

Notice that in the second and third cases, the `-c` option tells us whether the input is sorted via the exit code; in addition, when the input is not sorted, it tells us where the first "out of order" line of input is.

2.16 tr

The `tr` command (short for "translate") reads standard input, transforms characters in the input based on rules supplied by the user, and then writes the result to standard output. The command can operate in three modes, meaning it either

1. translates instances of one character into instances of another,
2. delete every instance of a character, or
3. deleting duplicate instances of a character (i.e., "squeeze" them).

The translate mode is the default; `-d` and `-s` options instruct `tr` to operate in delete or squeeze modes respectively. As well as the operating mode, `tr` needs to know the rules that control how the the input is transformed. The rules are specified using one more more character sets (depending on the mode).

An example will make this easier to explain. For translation, two sets need to be specified; imagine `tr` reads characters of input one at a time. If the i-th character it reads is not found in the first set, it is written to the output unchanged; if it matches the j-th character of the first set then the output is the j-th character in the second set. This is a very formal way to describe something very simple:

```
bash$ cat > A.txt
a
b
c
d
bash$ cat A.txt | tr 'a' 'e'
e
b
c
d
bash$
```

First we construct a file called `A.txt`, and then use `tr` to translate every instance of the character 'a' into the character 'e'; described as such, the output is as you might expect.

In this case single characters control the translation process, but the sets are more general. For example, imagine you want to turn all lower-case characters into their upper-case equivalents. We could build a command pipeline of twenty six invocations of `tr`, one for each character, but this would be awkward to say the least! Instead, one single invocation suffices which we can write in (at least) two ways:

```
bash$ cat A.txt | tr [a-z]     [A-Z]
A
B
C
D
bash$ cat A.txt | tr [:lower:] [:upper:]
A
B
C
D
bash$
```

In both cases, `tr` translates instances of 'a' into 'A', 'b' into 'B' and so on:

- The first usage is arguably the most intuitive, and describes the sets concretely in terms of their members: `[a-z]` and `[A-Z]` represent the lower- and upper-case characters respectively.
- The second usage arguably requires the least work, and describes the sets abstractly in terms of their properties: `[:lower:]` and `[:upper:]` represent the lower- and upper-case characters respectively.

Note that the sets do not need to be the same size:

```
bash$ cat A.txt | tr [a-c] 'e'
e
e
e
d
bash$
```

When this happens, `tr` pads the second set by repeating the last character as many times as necessary; in this case, the character 'e' is repeated three more times. Excess characters in the second set are simply ignored.

The translation mode is the most common use of `tr`, but remember it also allows one to delete and "squeeze" characters. Deletion is a simple concept: imagine we want to remove the end of line characters from `A.txt`. As a first attempt we could try

```
bash$ cat A.txt | tr '
> ' ' '
a b c d bash$
```

but this has simply translated the end of line characters into spaces, whereas we want to delete them. To do this, we use the `-d` option which instructs `tr` to delete characters that occur in the first set. As such, we can accomplish our goal using:

```
bash$ cat A.txt | tr -d '
> '
abcdbash$
```

Finally, "squeezing" characters basically means replacing a sequence of some duplicate character that occurs in the first set with a single copy of that character. Imagine we create a new file, this time called `B.txt`:

```
bash$ cat > B.txt
aaa
bbb
ccc
ddd
bash$
```

Now we have a sequence of 'a' characters for example. To squeeze this sequence into one copy of 'a' we use:

```
bash$ cat B.txt | tr -s 'a'
a
bbb
ccc
ddd
bash$
```

Or, to squeeze all the duplicates, in this specific case we could use

```
bash$ cat B.txt | tr -s [a-d]
a
b
c
d
bash$
```

noting the set of characters to squeeze is 'a' through 'd'. An alternative would be to use use a similar simplification as above, i.e.,

```
bash$ cat B.txt | tr -s [:lower:]
a
b
c
d
bash$
```

A final trick is provided by the `-c` option which complements (or inverts) the first set before it is used. Imagine we want to delete all characters except lower-case characters; this is now a little awkward if we follow the examples above, because we need to specify a set which has many members. However, the `-c` option makes it much easier:

```
bash$ cat > A.txt
a
b
c
d
bash$ cat A.txt | tr -dc [:lower:]
abcdbash$
```

2.17 `uniq`

The idea of the `uniq` command (short for "unique") is simple: imagine the input split into lines, the command compares the i-th line with successive lines (i.e., j-th lines for $j > i$) and removes all but the first identical instance. So for example, if line one is the same as line two then only line one appears in the output.

The input taken from a file named as an argument, or from standard input by default, and the output is written to standard output. To demonstrate the command we first need to construct `A.txt`, a short example file:

```
bash$ cat > A.txt
a
b
a
c
d
d
d
bash$
```

Using `A.txt`, it is easier to see what is going on:

```
bash$ uniq A.txt
a
b
a
c
d
bash$ cat A.txt | uniq
a
b
a
c
d
bash$
```

Notice that there are two 'a' characters on the first and third lines but these duplicates are not removed since the lines are not successive; the three lines at the end of `A.txt` which contain the 'd' character are successive and also identical however, so only the first appears in the output. Modern implementations of `uniq` have numerous other options. For example, if we make a new file called `B.txt`

```
bash$ cat > B.txt
a
b
a
c
D
d
D
bash$
```

the `-i` option can be used to ignore case:

```
bash$ cat B.txt | uniq -i
a
b
a
c
D
bash$
```

Another nice trick is that `uniq` can be instructed to print a count of how many identical copies of a line were encountered:

```
bash$ cat B.txt | uniq -i -c
      1 a
      1 b
      1 a
      1 c
      3 D
bash$
```

This starts to allow for some useful command pipelines; imagine we want to know how many lines in a given file start with the character 'a' (ignoring case). One solution would be to first use `cut` to extract the first character of each line, then use `sort` to sort the characters into order, then finally use `uniq` to count how many identical copies of the characters exits:

```
bash$ cat /usr/share/dict/words | cut -c 1-1 | sort | uniq -i -c | grep a
   4667 a
bash$
```

By completing the pipeline using `grep` to get the count for 'a' alone, the output we want is produced.

2.18 `wc`

The `wc` command counts the number of lines, words and characters (by default all three) in some input. In this context, a "word" is roughly defined as a string of non-space characters surrounded by space characters. The input taken from files named as arguments, or from standard input by default, and the output is written to standard output:

```
bash$ wc    /usr/share/dict/words
 99171  99171 938848 /usr/share/dict/words
bash$ wc -l /usr/share/dict/words
99171 /usr/share/dict/words
bash$ wc -w /usr/share/dict/words
99171 /usr/share/dict/words
bash$ wc -c /usr/share/dict/words
938848 /usr/share/dict/words
bash$
```

Note that the `-l`, `-w` and `-c` options instruct `wc` to output only the count of lines, words and characters; in this case the number of lines and words is the same because `/usr/share/dict/words` is essentially a list of words, one per-line.

The other useful trick `wc` can perform is to output the length of the longest line in the input. Consider an example where `wc` is supplied with input by `cat`, and the `-L` option is used:

```
bash$ cat > A.txt
a
bb
ccc
dddd
bash$ cat A.txt | wc -L
4
bash$
```

The output is 4 which is sane since the longest line in `A.txt`, namely `dddd`, has four characters in it.

In isolation `wc` might not seem useful, but when combined with other commands we can use it to solve real problems. Imagine we want to know how many lines in a given file start with the character 'a' (ignoring case). One solution would be to first use `grep` to filter out all lines except those starting with 'A' or 'a', and then `wc` to count the number of lines that remain:

```
bash$ cat /usr/share/dict/words | grep a] | wc -l
0
bash$
```

2.19 `wget`

To call `wget` a command is underselling it. This command has so many different options and uses, it should really be classed as an application: put simply, `wget` is like a web-browser, but one we use to automate tasks rather than use interactively.

Rather than explore the vast range of tasks `wget` can be used for, we focus on a single use-case: imagine we want to download a web-page and, instead of rendering it on the screen, process the content somehow using other commands (e.g., as if it were a file held locally). In order to accomplish this, we use three options

1. the `-q` option tells `wget` to operate quietly (i.e., not print out any progress information, which can be useful within a command pipeline),

2. the `-U` option allows `wget` to masquerade as a real web-browser by mimicking the style of commands and information supplied to the web-server (which can be useful to ensure a download is not blocked), and
3. the `-O` option tells `wget` where to store the output,

plus an argument that specifies the web-page via a URL. Imagine we want to download content from

http://www.gutenberg.org/dirs/etext97/1ws1810.txt

which is the text for *The Merchant of Venice by Shakespeare*, and save it into a file called `A.txt`. We can accomplish this with the deceptively simple command

```
<A.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ ls -l A.txt
-rw-rw-r-- 1 page page 0 Jun 22 17:00 A.txt
bash$
```

noting that here the `-U` option means `wget` acts like a Chrome web-browser, and `-O` instructs it to save the content into `A.txt`. We also told `wget` to be quiet, and it certainly was! It offers no output on standard output, but a subsequent `ls` output shows that `A.txt` represents was download successfully:

```
bash$ head -4 A.txt
bash$
```

Imagine we wanted to count the number of lines in the content downloaded in the example above; at least two approaches are possible:

1. download the content, save it into a file and then count the lines in the file content using `wc`, or
2. download the content, copy it to standard output and then feed this via a command pipeline to `wc`.

By replacing the file named after the `-O` option with a dash character (i.e., '-'), we can adopt the second approach:

```
< B.txt 'http://www.gutenberg.org/dirs/etext97/1ws1810.txt'
bash$ wc B.txt
0 0 0 B.txt
<       'http://www.gutenberg.org/dirs/etext97/1ws1810.txt' | wc
      0       0       0
bash$
```

References

[1] *Wikipedia: BASH*. http://en.wikipedia.org/wiki/Bash (see pp. 3, 5).

[2] *Wikipedia: Central Processing Unit (CPU)*. http://en.wikipedia.org/wiki/Central_processing_unit (see p. 4).

[3] *Wikipedia: Command Line Interface (CLI)*. http://en.wikipedia.org/wiki/Command-line_interface (see p. 5).

[4] *Wikipedia: Cygwin*. http://en.wikipedia.org/wiki/Cygwin (see p. 3).

[5] *Wikipedia: File descriptor*. http://en.wikipedia.org/wiki/File_descriptor (see p. 7).

[6] *Wikipedia: GNU/Linux naming controversy*. http://en.wikipedia.org/wiki/GNU/Linux_naming_controversy (see p. 3).

[7] *Wikipedia: Graphical User Interface (GUI)*. http://en.wikipedia.org/wiki/Graphical_user_interface (see p. 5).

[8] *Wikipedia: Hard disk drive*. http://en.wikipedia.org/wiki/Hard_disk_drive (see p. 4).

[9] *Wikipedia: History of GUIs*. http://en.wikipedia.org/wiki/History_of_the_graphical_user_interface (see p. 5).

[10] *Wikipedia: Kernel*. http://en.wikipedia.org/wiki/Kernel_(computing) (see p. 4).

[11] *Wikipedia: Live CD*. http://en.wikipedia.org/wiki/Live_CD (see p. 3).

[12] *Wikipedia: Multi-tasking*. http://en.wikipedia.org/wiki/Computer_multitasking (see p. 4).

[13] *Wikipedia: Pipe*. http://en.wikipedia.org/wiki/Pipeline_(Unix) (see p. 8).

[14] *Wikipedia: Process*. http://en.wikipedia.org/wiki/Process_(computing) (see p. 4).

[15] *Wikipedia: Random Access Memory (RAM)*. http://en.wikipedia.org/wiki/Random-access_memory (see p. 4).

[16] *Wikipedia: Redirection*. http://en.wikipedia.org/wiki/Redirection_(computing) (see p. 7).

[17] *Wikipedia: Shell*. http://en.wikipedia.org/wiki/Shell_(computing) (see p. 4).

[18] *Wikipedia: Shell built-in*. http://en.wikipedia.org/wiki/Shell_builtin (see p. 4).

[19] *Wikipedia: Standard streams*. http://en.wikipedia.org/wiki/Standard_streams (see p. 6).

[20] *Wikipedia: System call*. http://en.wikipedia.org/wiki/System_call (see p. 4).

[21] *Wikipedia: Terminal*. http://en.wikipedia.org/wiki/Computer_terminal (see p. 5).

[22] *Wikipedia: UNIX*. http://en.wikipedia.org/wiki/Unix (see p. 3).

I have a question/comment/complaint for you. Any (positive *or* negative) feedback, experience or comment is very welcome; this helps us to improve and extend the material in the most useful way.

However, keep in mind we are far from perfect; mistakes are of course possible, although hopefully rare. Some cases are hard for us to check, and make your feedback even more valuable: for instance

1. minor variation in software versions can produce subtle differences in how some commands and hence examples work, and
2. some examples download and use online resources, but web-sites change over time (or even might differ depending on where you access them from) so might cause the example to fail.

Either way, if you spot a problem then let us know: we will try to explain and/or fix things as fast as we can!

Can I use this material for something ? We are distributing these PDFs under the Creative Commons Attribution-ShareAlike License (CC BY-SA)

http://creativecommons.org/licenses/by-sa/4.0/

This is a fancy way to say you can basically do whatever you want with them (i.e., use, copy and distribute it however you see fit) as long as a) you correctly attribute the original source, and b) you distribute the result under the same license.

Is there a printed version of this material I can buy? Yes: Springer have published selected Chapters in

http://www.springer.com/computer/book/978-3-319-04041-7

while *also* allowing us to keep electronic versions online. Any royalties from this published version are donated to the UK-based Computing At School (CAS) group, whose ongoing work can be followed at

http://www.computingatschool.org.uk/

Why are all your references to Wikipedia? Our goal is to give an easily accessible overview, so it made no sense to reference lots of research papers. There are basically two reasons why: research papers are often written in a way that makes them hard to read (even when their intellectual content is not difficult to understand), and although many research papers are available on the Internet, many are not (or have to be paid for). So although some valid criticisms of Wikipedia exist, for introductory material on Computer Science it certainly represents a good place to start.

I like programming; why do the examples include so little programming? We want to focus on interesting topics rather than the mechanics of programming. So even when we include example programs, the idea is to do so in a way where their meaning is fairly clear. For example it makes more sense to use pseudo-code algorithms or reuse existing software tools than complicate a description of something by including pages and pages of program listings.

But you need to be able to program to do Computer Science, right? Yes! But only in the same way as you need to be able to read and write to study English. Put another way, reading and writing, or grammar and vocabulary, are just tools: they simply allow us to study topics such as English literature. Computer Science is the same. Although it *is* possible to study programming as a topic in itself, we are more interested in what can be achieved *using* programs: we treat programming itself as another tool.