1 The Unix shell - bash

The first thing one is likely to encounter in any Unix system is the *shell*.

While it is a program like any other, for many people it is what "makes" a Unix machine. It takes input from you, say, via the keyboard, and can start other programs and in general allows you to interact with the system. It is *the* program that you will use all the time. 99% of my time with computers is spent in shells on various machines.

Why is the shell such a powerful tool?

Imagine you had some nice program (which I'm sure someone has written) that allows you to add a watermark to your vacation pictures, for example, the date and time when the picture was taken.

Now you are sitting in front of, say, 980 such pictures from your last vacation. Maybe that programmer wasn't worth his salt, and you will find yourself repeating 980 times -

- open the picture
- make a few clicks to effect the watermark
- save the picture
- repeat.

Photoshop, while it has a ton of other things you can do, is one example of such a program. Good for 1,2, maybe 10 photos. Hundreds? Maybe not. You want to automate that process.

Ok, let's say that the programmer figured that out while doing his or her own vacation pictures, and provides functionality to "take all the pictures in this folder, extract the date from the Exif data in the picture, and add a watermark in the lower left corner".

That's better, but you see where this is going. Maybe you want the watermark in the lower right corner. Maybe you want the date formatted in the European notation. Maybe you want the watermark in a different color. Maybe you don't want the date at all but something else. For a single monolithic program, however powerful, it is impossible to predict everything that a user might want to do. And the more the programmer lets his or her imagination run wild, the more complex and feature-overloaded that thing becomes, and it is soon impossible to really "get" it.

This is a simple example where the shell is lightyears more versatile.

Here we have a program that is good at image manipulation, such as adding the watermark in myriad ways (or doing anything else, converting to B&W, what have you). One picture at a time. It takes whatever text we give it and puts it in the picture as a watermark.

Then there's a way to go through all pictures. They don't have to be all in one folder, you can finesse a selection such as "all pictures that were taken with my cell phone and have a size 1280x720 or larger, and are geo-tagged, and have been taken in the last 90 days". No problem. The way it's handled is that there is a program that selects by date in myriad ways, another one that figures out the size of a pic, another one that lets you determine the device with which the picture has been taken.

Not one program alone has to be able to do all of this and has to be good at it. Rather, you string the stuff together as you see fit from simple building blocks which are exceptionally good at one thing but don't need to concern themselves with anything else. For example, the watermarking program does not need to know how to format the string that it puts on as the mark, it just takes a ready-made string which is prepared by another utility.

As our adventure here progresses and we learn a few new concepts, I'll show you how I did this, but for now let me show you the end result of such a watermarking project. Here I wasn't dealing with 980 but thousands of pics from getting a new roof on my house. I had set up a webcam in my shed in the backyard that was taking a snapshot every few seconds. I wanted to make a time-lapse video of the entire day, showing the progress of the roofing. That was easy to do and looked nice. But for kicks and laughs, I wanted spice it up a bit and show the progress of the work in terms of the money I had spent on this - $17,585 or so total. Here is how it breaks down:

`https://www.dropbox.com/s/rvh5kxt985zjclo/roof2.avi?dl=0`

You'll see the elements I discussed before. For each frame and time elapsed, I calculated the fraction of the money spent, formatted the string, and added the watermark to the frame. It's all automated by stringing said components together. I am sure that you will not find a monolithic program that would have that as a ready-made function. In the shell, it took me 27 lines of text, including 10 empty lines for improved readability. I also think it's funny. I hope. (If you think that *I* counted the total and non-empty lines, you'd be wrong, by the way).

1.1 Shell Basics

Ok, let's get to some real basics.

First off, basically everything you do in the shell is done by starting a program which does whatever it is you want.

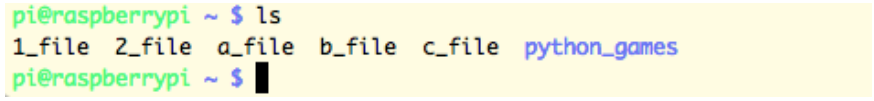
For example, in order to see the list of files in a folder, you'd use the command "`ls`". I'm showing here the entire lines you'll see on the screen. The first

```
pi@raspberrypi ~ $
```

is the *prompt*. As the name suggests, it prompts you to enter some commands.

The prompt can be customized to show different things if you are so inclined. We'll get to this later, but our standard prompt here shows the user name (pi), the hostname (raspberrypi), and the current directory (~). These days, where a genuine

black-and-white monitor is a museum piece, the different parts usually come out in color:



So this is what "ls" shows:

```
pi@raspberrypi ~ $ ls
1_file  2_file  a_file  b_file  c_file  python_games
pi@raspberrypi ~ $
```

So there are 6 entries shown in that folder. Now if you type "ls", followed by enter, what the shell does is invoke another program, which happens to be called `/bin/ls`, which looks at what files are in the folder, and displays their name.

We are getting into a few system things here, but know that there is no real distinction between a system-provided program such as "`/bin/ls`" and something that you develop yourself. The way this works is that there is a definition (known as an environmental variable) called `PATH`. It lists a number of directories (folders) where the system looks for executable programs. If you type a command such as "`ls`", the shell traverses the elements of `PATH`, looks at each area in turn, and runs the first file with that name (and that is marked as an executable, e.g. a program) that it finds. Here is the current value of `PATH`:

```
pi@raspberrypi ~ $ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:/usr/local/games:/usr/games
pi@raspberrypi ~ $
```

The individual folders to search in are separated by a colon. You see "`/bin`" in 6th position, and that's where the shell finds "ls".

How do I know that "`/bin/ls`" is what's being run? There is a utility called "which", which tells you which actual program is chosen as the result of a command:

```
pi@raspberrypi ~ $ which ls
/bin/ls
pi@raspberrypi ~ $
```

You ave already guessed that "which" itself is yet another program, and indeed:

```
pi@raspberrypi ~ $ which which
/usr/bin/which
pi@raspberrypi ~ $
```

Now let's see what happens if we tweak that PATH variable and cut out the `/bin` element from it:

```
$ PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/usr/local/games:/usr/games
```

```
pi@raspberrypi ~ $ ls
-bash: ls: command not found
pi@raspberrypi ~ $
```

Now the shell doesn't find "ls" anymore.

Don't worry about messing anything up in your system by changing environmental variables like this. By design, it is not possible that such changes propagate "up" in the system. So those on-the-fly changes can only affect the current shell and its descendants, if any. If something stops working, just get yourself a new shell and discard the old one (e.g., by logging out and in again, or by making a new window), and all will be working again as before.

Exercise 1

Change the `PATH` variable as shown above so that the `which` program is no longer found.

One side lesson: It is usually bad form to prepend any of your own areas to the `PATH`. The `/bin` and `/usr/bin` and some more areas are protected from tampering by virtue of being owned by the system. You as a standard user can execute programs found in those directories but not actually change them. That is one key element that makes Unix/Linux installation more hacker-proof, that anything that you allow to execute on your system (say, the acrobat reader installer) *cannot* modify any of the good stuff. On Windows, many people run everything with Administrator rights, and anything is allowed to modify anything else. You have tons of sites where you can download something for Windows which in the end is an executable program – you will never know what it does to your system. With that said, you can open doors on Linux by starting your PATH with one or more of your own directories. *Never do that.* Some other program that you might execute is prohibited from tampering with the "ls" program in `/bin`, but it might drop a different version with who knows what side effects into one of your directories. If that comes in the PATH before the system areas, you will unknowingly execute a potential rogue version of "ls". There have been plenty of those – a virus program which modified "ls" in a way that it never showed the virus files themselves, etc etc.

While prepending stuff is possible although frowned-upon, we usually set up the `PATH` in a way that we add some task-specific areas to its end. For example, if you are working with PHENIX raw data, you will routinely use a few utilities that I wrote, called `dlist`, `ddump`, and `dpipe`. If we log into a computer in the PHENIX data acquisition system, we see

```
> purschke@va050: ~> which ddump
> /export/software/oncs/proLinux_SL6/install/Linux.i686-sl6.4.0/bin/ddump
```

The system is set up in a way that whenever you log in to a machine that belongs to the data acquisition system, the

```
/export/software/oncs/proLinux_SL6/install/Linux.i686-sl6.4.0/bin
```

area shows up in the `PATH`. By the same token, if you do the same at the RHIC Computing Facility:

```
> [purschke@rcas2065 ~]$ which ddump
> /afs/rhic.bnl.gov/phenix/PHENIX_LIB/sys/x8664_sl6/64/new.2/bin/ddump
```

You see, a different area, and a different way to set up the PATH, but the same result: the `ddump` utility has become a command that is available to you just as "ls" is.

Command Line Options

Now the "one program - one task" concept is nice, but usually you want to be able to tweak the exact behavior of a given utility. For example, you may want "ls" to show different aspects of files, or more information, or whatever, instead of just listing them as shown before

```
pi@raspberrypi ~ $ ls
1_file  2_file  a_file  b_file  c_file  python_games
pi@raspberrypi ~ $
```

Those behavior-changing things are called command-line options, and are usually either a single-letter option prepended by a minus, or hyphen, or "dash" ("-"), or word-options, by convention denoted with two hyphens or dashes. So I can add a "-l" ("dash-ell") to ls, and get

```
pi@raspberrypi ~ $ ls -l
total 4
-rw-r--r-- 1 pi pi    0 May 20 11:03 1_file
-rw-r--r-- 1 pi pi    0 May 20 11:03 2_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 a_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 b_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 c_file
drwxrwxr-x 2 pi pi 4096 Jan 27 08:34 python_games
pi@raspberrypi ~ $
```

and can see the file permissions, who owns them (the user "pi"), last modification date, size, and name.

In this presentation, you see that the last entry, `python_games`, has a "d" in the very first column. This says that this is not an actual file but another directory. We have a folder in a folder here, which can (and does) in turn contain other files and more folders.

It is virtually impossible to know or remember all options for a given command. "ls" alone has about 100 such options. Most people memorize `-l`, `-l -a`, `-l -t`, and so on. Single-letter options such as `-l -t -r` can be combined into `-ltr`, and that's how I at least remember them. Since no one remembers all of them, if you are looking for something specific, there is a handy "`man`" – short for "manual" – command, and "`man ls`" gives you everything you ever wanted to know about the ls program, and then some.

`ls -ltr` is quite handy. The "-t" instructs ls to sort the files not alphabetically as usual, but by modification date (there is also a different date, the creation date, that can be selected). The "-r" instructs ls to reverse-sort, so `-ltr` shows the youngest files at the bottom of the listing.

```
pi@raspberrypi ~ $ ls -ltr
total 4
drwxrwxr-x 2 pi pi 4096 Jan 27 08:34 python_games
-rw-r--r-- 1 pi pi    0 May 20 11:01 a_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 b_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 c_file
-rw-r--r-- 1 pi pi    0 May 20 11:03 1_file
-rw-r--r-- 1 pi pi    0 May 20 11:03 2_file
pi@raspberrypi ~ $
```

This is quite useful. On almost all systems, files downloaded by the web browser are usually put in the `Downloads` folder. Usually, there are quite a number of files in there. `ls -ltr` will show the just-downloaded file at the bottom of the listing.

Without the -t, the -r reverses the alphabetical order:

```
pi@raspberrypi ~ $ ls -lr
total 4
drwxrwxr-x 2 pi pi 4096 Jan 27 08:34 python_games
-rw-r--r-- 1 pi pi    0 May 20 11:01 c_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 b_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 a_file
-rw-r--r-- 1 pi pi    0 May 20 11:03 2_file
-rw-r--r-- 1 pi pi    0 May 20 11:03 1_file
pi@raspberrypi ~ $
```

There are other ways of sorting the output of ls. Let's say you are running out of disk space all of a sudden, and you suspect that there is some large file that has been created. You see that all the actual files (`a_file`, etc) have a size of 0. The `-S` switch to `ls` sorts the entries by size, largest file first, and `-r` reverses this to show the largest file(s) at the end of the list.

The other kind of option is the "long" option, which is an entire word and supposed to be easier readable, which is true up to a point, with the downside that it is more to type. "ls" has quite a number of long options. Some of them are alternatives to the short options we have already seen. For example, instead of `ls -r` you can type `ls --reverse`.

Generally speaking, most authors of such a utility will try to make the most often used options into short options, and keep the rarely used ones as long options. If you find a command `ls --dereference-command-line-symlink-to-dir`, it might give you an idea what that option effects.

To close out this chapter, let me point out that the one dash and two-dashes style for options is just a convention, and there are tons of programs which do not follow it. For example, there are two image-modification utilities called `convert` and `mogrify`, which take an option that instructs them to strip out any "Exif" data, that is, any information embedded in an image but the image itself. That removes

all geo-tagging, any camera-specific information, camera owner, what have you. I use this, always, for pictures I'm posting on Facebook. Normally the syntax should be

```
mogrify --strip picture.jpg
```

but it is just `-strip`, no two dashes. Oh well. Just the way it is.

Hidden Files

We have seen the `ls` command at work before. I had made it so that there were a number of files in that directory that we could list.

That "first" directory that you see after you login is called the *home directory*, which every user of the system has. In our case as user "pi" on our Raspberry Pi, it is "`/home/pi`", but the exact naming of the home directories are system- and setup-dependent. For example, on my Mac, where my user name is "purschke", my home directory is "`/Users/purschke`".

Unbeknownst to you, in that home directory will usually be a number of additional files which `ls` will not show to you unless explicitly asked. Those are known as *hidden files*, those whose names start with a period, or a dot.

The normal `ls` command will not show them, but add a `-a` option to `ls`, and you can see them:

```
pi@raspberrypi ~ $ ls -al
total 44
drwxr-xr-x 4 pi   pi   4096 May 21 00:51 .
drwxr-xr-x 3 root root 4096 Feb 15 14:05 ..
-rw-r--r-- 1 pi   pi      0 May 20 11:03 1_file
-rw-r--r-- 1 pi   pi      0 May 20 11:03 2_file
-rw-r--r-- 1 pi   pi      0 May 20 11:01 a_file
-rw------- 1 pi   pi    177 May 20 05:25 .bash_history
-rw-r--r-- 1 pi   pi    220 Feb 15 14:05 .bash_logout
-rw-r--r-- 1 pi   pi   3243 Feb 15 14:05 .bashrc
-rw-r--r-- 1 pi   pi      0 May 20 11:01 b_file
-rw-r--r-- 1 pi   pi      0 May 20 11:01 c_file
-rw------- 1 pi   pi     43 May 20 11:56 .lesshst
-rw-r--r-- 1 pi   pi    675 Feb 15 14:05 .profile
drwx------ 2 pi   pi   4096 May 20 11:02 .pulse
-rw------- 1 pi   pi    256 May 20 11:02 .pulse-cookie
drwxrwxr-x 2 pi   pi   4096 Jan 27 08:34 python_games
-rw------- 1 pi   pi     57 May 21 00:51 .Xauthority
```

You see our normal files which we have seen before, and then some hidden ones such as `.bashrc`, `.pulse`, and so on, which the `-a` option now brought to light. As you already know by now, the "d" in the first column denotes a directory, and so that `.pulse` is an entire *hidden directory*. Although they can be used for any purpose just like "visible" files, the hidden ones are often used as configuration files for programs and applications, or to retain some information between sessions. You do not normally want to be bothered with such administrative files, that's why they are hidden.

It is not uncommon to find several dozen hidden files and directories in the home directory of your laptop or desktop. Many programs that you use create hidden files to store some information. The more programs you use, the more such files and directories you will find.

Let me also stress that hidden files and directories can be created anywhere, not just in the home directory.

An example of an information-saving hidden file is `.bash_history`, which keeps a record of the actual commands you typed in the previous session.

You can use the up-arrow on your keyboard (and some other tricks, we get to them later) to recall a previous command; you can also edit a previous command and execute the new version. This is a very useful feature, for example, if you mis-typed a command and got an error. You can simply recall the command, correct the mistake, and execute the proper command. That `.bash_history` files retains that list of commands across your sessions.

There are a few variations of that `.profile` hidden file. They are executed at the time when you log into the machine, and can be used to tweak the functioning or setup of your shell automatically. This would be the place where you can add to your `PATH` variable, and in this way make any addition persistent between sessions. Each time you log in, the `.profile` script would set up your `PATH` in the desired way.

If you are so inclined, you could have your computer greet you properly by adding code to print some "Welcome back!" message to the `.profile` script. Similarly, there is a `.bash_logout` script which kicks in when you end your session, giving you the chance to add code to clean up after yourself is needed. I personally have not had any use for the `.bash_logout` script, ever, but it's good to know that the mechanism exists.

Traversing the directory tree

So far we have not "left" our home directory; we have only looked at files which happened to be there already, hidden or not. We will stick with calling what is known as folders in other operating systems *directories*. People used to shell naming conventions might not know what you mean if we are talking about folders, but more importantly, many commands make only sense if we go with the "directory" name.

And that brings us to our first new concept, the *directory tree*. Under Unix, directories are delimited with a "slash" (`/`). Now it is usually called the directory *tree*, suggesting something that branches upwards, like the branches of an actual tree. However, I myself (and virtually everyone I know) visualizes the "tree" as something that branches downwards. "Go down one directory" virtually always refers to moving into a subdirectory.

The tree starts with the *root* directory, aptly named `/`. That's where everything else is branching off from. For example, the files we were looking at before are in the home directory `/home/pi`. So the "home" area is where often the different users' home directories are kept.

A given tree can have an arbitrary depth.

Unless we set this up to be different, we usually find ourselves in that home directory, `/home/pi` in this case (since we logged in as user "pi").

What does "we find ourselves in" that directory really mean? It means that without adding any other directory information, all file operations get an invisible `/home/pi/` prepended, so the command

```
ls -l
```

really means

```
ls -l /home/pi/
```

That is known as the *current work directory*, which you can change at any time.

It is considered bad form to perform all your work, say, writing a document, writing code, preparing your taxes, etc, all of which involves creating files, in your home directory, or in any one single directory. That would lead to an extremely cluttered work environment which we want to avoid at all costs. You will want to keep files from different projects apart, that is, put them into their own dedicated directory, which often not only means one directory, but a certain hierarchy of directories, not unlike your actual physical filing cabinet.

So you might end up with a directory `/home/pi/work`, in which you find directories for different things, such as `/home/pi/work/customer_X`, `/home/pi/work/customer_Y`, and so on, and maybe `/home/pi/work/designs`, `/home/pi/work/expenses`, and so on, and so on. You could also choose to organize the individual customer directories in yet another `/home/pi/work/customers` directory, so you would have `/home/pi/work/customers/customer_X`, `/home/pi/work/customers/customer_Y`, and so on. The goal is to avoid clutter and establish an environment that matches the structure of your work and your work flow.

In order to create an empty new directory, you use the `mkdir` command. So in order to establish the directory tree as outlined above, you would issue

```
pi@raspberrypi ~ $ mkdir work
pi@raspberrypi ~ $ ls -l
total 8
-rw-r--r-- 1 pi pi    0 May 20 11:03 1_file
-rw-r--r-- 1 pi pi    0 May 20 11:03 2_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 a_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 b_file
-rw-r--r-- 1 pi pi    0 May 20 11:01 c_file
drwxrwxr-x 2 pi pi 4096 Jan 27 08:34 python_games
drwxr-xr-x 2 pi pi 4096 May 21 02:02 work
```

and you see a new directory with that name. Keep in mind that the current work directory adds that stealth `/home/pi/` in front of `work`, so this is equivalent to issuing

```
pi@raspberrypi ~ $ mkdir /home/pi/work
```

Now we make the "customers" directory:

```
pi@raspberrypi ~ $ mkdir work/customers
```

followed by the individual customers' directories:

```
pi@raspberrypi ~ $ mkdir work/customers/customer_X
pi@raspberrypi ~ $ mkdir work/customers/customer_Y
```

So by now that part of the tree looks like

```
/
 home/
      pi/
         work/
              customers/
                       customer_X/    customer_Y/
```

In the same spirit, we add the "designs" and "expenses" directories:

```
pi@raspberrypi ~ $ mkdir work/designs
pi@raspberrypi ~ $ mkdir work/expenses
```

and have now

```
/
 home/
      pi/
         work/
              customers/                              designs/   expenses/
                       customer_X/  customer_Y/
```

Exercise 2

Create the directories in your home directory as shown previously.

Let's assume now that you want to work on files for customer "X", and put all relevant files, say, a spreadsheet file "`project.xls`" into the `work/customers/customer_X` directory. You could at this point refer to the file as `work/customers/customer_X/project.xls`. That's a lot of typing each time, so what we will do now is to change our work directory.

The `cd` and `pwd` commands

The command to change the current work directory is called `cd`, for "change directory". You can have relative or absolute changes – the absolute changes all start out from the file system root, and hence with a slash (`/`). So in order to move down into the "work" directory with an absolute path, you could issue

```
pi@raspberrypi ~ $ cd /home/pi/work
pi@raspberrypi ~/work $
```

and be in the right place. Most Unix/Linux users actually visualize the place on the directory tree like an actual, physical position – "I am *in* the so-and-so directory" is something you'll hear often. Staying with this way of saying it, we are now *in* the `/home/pi/work` directory.

Let's quickly introduce the `pwd` command – it stands for "print work directory" and shows you in which directory you are. At this point:

```
pi@raspberrypi ~/work $ pwd
/home/pi/work
pi@raspberrypi ~/work $
```

You will also note that the prompt has changed and now reflects the current work directory. This is the reason why I always showed the full screen content with the commands together with the prompt so far, which I will stop when we are finished with this chapter.

In most cases one makes just relative moves in the directory tree, which are easier and quite versatile. Since we found ourselves in the `/home/pi` directory initially, we could have made a simpler (with less typing, and less memorizing) relative move:

```
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $ cd work
pi@raspberrypi ~/work $ pwd
/home/pi/work
pi@raspberrypi ~/work $
```

The `cd work` command got us down into the work directory, but how do we get back up with a relative move?

If you go back a bit to where I introduced the hidden files, you may have noticed two hidden files, one with just the period, or dot, and one with two dots:

```
drwxr-xr-x 4 pi   pi   4096 May 21 00:51 .
drwxr-xr-x 3 root root 4096 Feb 15 14:05 ..
```

The single dot refers to the current directory, which comes in handy as a shortcut for "here" – more about that later. The two-dot directory is shorthand for "one directory above", no matter where we are[1]. The above two lines are from our home directory, and you see that "`..`", which refers to `/home`, is owned by the superuser, rather than by us, as it should be.

Now that we have moved, one way or another, down to `/home/pi/work`, the `..` refers to the one directory above that is, `/home/pi`:

```
pi@raspberrypi ~ $ pwd
```

[1] the only exception is the file system root `/`, from where there is no "up" anymore.

```
/home/pi
pi@raspberrypi ~ $ cd work
pi@raspberrypi ~/work $ pwd
/home/pi/work
pi@raspberrypi ~/work $ ls -al
total 8
drwxr-xr-x 2 pi pi 4096 May 21 02:02 .
drwxr-xr-x 5 pi pi 4096 May 21 02:02 ..
pi@raspberrypi ~/work $ cd ..
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $
```

So, using only relative moves, we went down into "work", and back up again.

As a special case, using `cd` without any parameter is taking us back to our home directory, irregardless of where we are:

```
pi@raspberrypi ~/work $ pwd
/home/pi/work
pi@raspberrypi ~/work $ cd
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $
```

Ok, the original plan was to get to the directory for customer "X". We'll stay with the relative moves, and we can go there in one fell swoop:

```
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $ cd work/customers/customer_X
pi@raspberrypi ~/work/customers/customer_X $ pwd
/home/pi/work/customers/customer_X
pi@raspberrypi ~/work/customers/customer_X $
```

Again, the prompt reflects the current work directory. Let me point out that this feature is setup-dependent, and can be customized, as virtually anything in the shell, and in Unix/Linux in general.

Starting out from our home directory, we could have arrived at our destination in different ways. For example, I could have broken up the above cd command into 3 steps:

```
pi@raspberrypi ~ $ pwd
/home/pi
pi@raspberrypi ~ $ cd work
pi@raspberrypi ~/work $ cd customers
pi@raspberrypi ~/work/customers $ cd customer_X
pi@raspberrypi ~/work/customers/customer_X $ pwd
/home/pi/work/customers/customer_X
pi@raspberrypi ~/work/customers/customer_X $
```

Just as I went down to `customer_X` from the home directory with one `cd` command before, you can navigate up more than one level by issuing

```
pi@raspberrypi ~/work/customers/customer_X $ pwd
/home/pi/work/customers/customer_X
pi@raspberrypi ~/work/customers/customer_X $  cd ../../
pi@raspberrypi ~/work $ pwd
/home/pi/work
pi@raspberrypi ~/work $
```

You can also combine up and down-again moves in one command:

```
pi@raspberrypi ~/work/customers/customer_X $ pwd
/home/pi/work/customers/customer_X
pi@raspberrypi ~/work/customers/customer_X $  cd ../../designs
pi@raspberrypi ~/designs $ pwd
/home/pi/designs
pi@raspberrypi ~/designs $
```

This command takes you 2 levels up and one level down again to the `designs` directory.

Exercise 3

Make a number of relative and other moves in the previously created directory tree. Each time, verify with `pwd` that you got it right.

- issue `cd`, which brings you to your home directory.
- make a relative move to work.
- make a relative move to customer "X" with one `cd` command.
- make a relative move from customer "X" to customer "Y" with one `cd` command.
- from there, move to "expenses" with one `cd` command.
- use a single cd to get back to the home directory.

Shell Amenities

Now that we are getting proficient with the shell, let's talk about a few time- and keystroke-saving things. Obviously, many computer experts have spent and are spending a lot of time using the shell, so there's a lot of things built in that will allow you to work a lot faster.

The tab key is your friend

The tab key, located usually somewhere at the very left of your keyboard, will expand a file or in general a path either fully or at least to the last unambiguous place. So if we are in our home directory, where at this point we only have one file and directory starting with "w" (work), typing `cd w` and pressing the tab key will

expand this to the full `cd work/`. This is a short command, but you can continue this. Say you want to go to our `customer_Y` directory. Type `cd w`, tab, c, tab, c, tab. That takes you to `work/customer/customer_` where the choices stop being unambiguous. If you hit tab one more time, the shell presents you with the choices. So this saves you a lot of typing. If you observe seasoned users interact with the shell, they hit tab all the time. This also works with commands - just type the first few letters of a command, and the shell expands it to the best of its abilities.

For example, I happen to have a file named

`tracking_configuration_study_August1_2014.pdf`

here on my Mac. I only need to type "tra" and the tab will expand the long-ish filename.

Repeating or modifying previous commands

You can use the up-arrow (and down-arrow) to navigate in the list of previous commands. This also allows you to edit a previous command.

You can use `!$` as shorthand for the last word from the most recent command. That allows you to quickly get the last parameter, or, if that command had no parameters, the command itself. I mentioned the

`tracking_configuration_study_August1_2014.pdf`

file, which I previously expanded with the tab key. Say I did

```
ls  tracking_configuration_study_August1_2014.pdf
```

and now want to see more, I can type

```
ls -l !$
```

Similarly, the `!^` shortcut gives you the first parameter on the previous command line.

There are many other shortcuts such as `!:1`, which is the same as `!^`. This sequence will give you the n^{th} parameter, e.g. `!:3` will give you the third, and `!:0` will give you the actual command.

However, I myself find that I do not use them frequently enough that they became part of my daily routine, since I have to concentrate on what I need to type, but that's just what I am using most.

Another shortcut you should know about is `!some_string`, which recalls the last command that matches `some_string`, *and executes it.* That latter part is why I almost never use this unless I am really certain that this command recall is getting me what I want. I typically use this feature with `CTRL-R`, which does the same but gives me a chance to review what is about to happen.

Other keystroke shortcuts can come in handy and can save a lot of time.

- if you think of the letter "T" to mean "transpose", it is easy to remember the `ESC-T` sequence (hit the "escape" ("esc") key usually located at the upper left

corner of your keyboard), followed by "T", will switch the last two parameters on the command line. Let's say you just executed `mv myfile oldfile` and realize that was the wrong thing to do, recall the command and hit `ESC-T`, and the command changes to `mv oldfile myfile`. Similarly, `CTRL-T` (hit "T" while holding down `CTRL`) transposes the last two characters.

- `CTRL-A` takes you to the begin of the command line
- `CTRL-E` takes you to the end of the command line
- `CTRL-K` and `CTRL-U` clear the text from the cursor position to the end of of the command line, and to the begin of the command line, respectively.

There are many more such shortcuts that you can google; I am listing the ones that I use somewhat frequently.

Parking a command without executing it

We already learned abut the possibility to recall commands, edit them as needed, and re-executing them.
Sometimes it happens that I have already typed a lengthy command, for example

```
wget -q -O - http://some.server.net/a_program.tar.gz | tar xvz
```

only to remember that I am not in the "right place" for that command to put the files where I want them. Rather than erasing the already-typed line, I go back to he beginning of the command line (by using the CTRL-A shortcut), and change the command into a non-existing one, and make the command fail, deliberately, for example:

```
xxwget -q -O - http://some.server.net/a_program.tar.gz | tar xvz
```

In this way, I "store" this command in the history. I cd then to the right place and then recall the command, remove the "xx" I added to make it fail, and finally execute it.

More commands

Let's move on to a few more commands which will be useful in what is about to follow.

echo

First is the `echo` command, which "echoes" all the parameters that follow, no matter what it is. So if you type

```
echo hello
```

you get "hello" back. I can hear you wonder how lame that is; after all, you knew the answer up front. So why is that so useful?

First, you do *not* always know what the answer will be. Remember our `PATH` environmental variable that we played with before? If you type `$PATH` anywhere on the command line, the shell (and not the program you run) will substitute the variable with its value. In this way,

```
echo $PATH
```

will show you the value of that variable as shown before on page 3. In the same way, you can look at (and preserve!) the value of `!$` (the last word):

```
echo !$
```

since it is again the last word, this allows you to verify that what you are about to do acts on the right thing, and then use `!$` again.

grep

This is the command to search for the occurrence of some text in a file or a selection of files. Let's say I want to know if I added the name of a particular person to a list of names, I can use

```
grep Martin names.txt
```

and will see all lines that contain my first name.

`grep` has a rich set of options to modify its behavior. The ones I use most frequently is the `-v` switch, which selects all lines which do *not* contain that string.

The other option is `-l`, which only lists the files that contain the string without showing the contents. Very often you only want to see that list. For example, if I am debugging a program and want to know in which source files the variable `momentum_correction` occurs, I can use

```
grep -l momentum_correction *.h *.cc
```

and see a list of all those files.

Let me point out that `grep` also supports *regular expressions*, which are beyond the scope of this introduction. However, if you are a programmer or a scientist using a computer for the day-to-day work, proficiency with regular expressions is a must.

find

This is an extremely powerful command that allows you to find files with an abundance of search criteria. For example, it allows you to find files which are larger than 5KB, contain the string ".cc" in their name, and are normal files. It also allows to act on each file found with some arbitrary command if you so choose.

```
$ cd
$ find . -name "*.cc"
```

will find *all* C++ source files in my entire directory tree (which can be quite a few).

I often wonder "where did I put that" - let's say I'm looking for the manual of my Samsung cable box to figure out some setting, and I don't recall where in my directory tree I put that. I also don't know the exact name of the file, but I do recall that it was in Acrobat Reader format, a PDF file.

```
$ cd
$ find . -name "*amsung*.pdf"
```

I only specify `amsung` to match file names `Samsung.pdf` as well as `samsung.pdf`, or any file such as `samsung_cablebox.pdf` with this.

With the `-type` you can select files which are a regular file, a directory, or any number of types, such as a block special file (which we won't get into), and so on.

```
$ cd
$ find . -name "*amsung*.pdf" -type f
```

searches for files which not only match that file name pattern, but are also a regular file. It is unlikely that `samsung.pdf` would be the name of a directory, but a file name `analysis` might be a file or a directory.

```
$ cd
$ find . -name "*analysis*" -type d
```

will show me all files which contain the string "analysis" anywhere in the name, and are also drectories.

bc

You will likely find some kind of calculator program on your system, with the appearance of an actual pocket calculator. I'm not disputing the usefulness of such a program, but I find that `bc` is vastly more useful.

It is a command-line based calculator *with arbitrary precision*. That is right, it has the ability to calculate things with an arbitrary number of significant digits. I always invoke it as `bc -lq`, which loads some additional math libraries but also pre-sets some more meaningful precision, and prevents a copyright message from being printed.

Here is the command without the `-q` option:

```
$ bc -l
bc 1.06
Copyright 1991-1994, 1997, 1998, 2000 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type ‘warranty’.
6751 * 33445
225787195
```

This is usually much faster than firing up a dedicated graphical calculator program.

To show off the "unlimited precision" aspect of `bc`, let's calculate π to 1000 digits precision. The `-l` pre-loads some math libraries, where the `a(x)` function calculates the `arctangent` of x. `a(1)` will give $\pi/4$, so `4*a(1)` will give π. We set the `scale` variable to set 1000 digits precision:

```
$ bc -lq
scale=1000
4*a(1)
3.14159265358979323846264338327950288419716939937510582097494459230\
78164062862089986280348253421170679821480865132823066470938446095505\
82231725359408128481117450284102701938521105559644622948954930381964\
42881097566593344612847564823378678316527120190914564856692346034861\
04543266482133936072602491412737245870066063155881748815209209628292\
54091715364367892590360011330530548820466521384146951941511609433057\
27036575959195309218611738193261179310511854807446237996274956735188\
57527248912279381830119491298336733624406566430860213949463952247371\
90702179860943702770539217176293176752384674818467669405132000568127\
14526356082778577134275778960917363717872146844090122495343014654958\
53710507922796892589235420199561121290219608640344181598136297747713\
09960518707211349999998372978049951059731732816096318595024459455346\
98302642522308253344685035261931188171010003137838752886587533208381\
42061717766914730359825349042875546873115956286388235378759375195778\
18577805321712268066130019278766111959092164201988
```

Input and Output Re-Direction

Among the most powerful features of the shell is what you can do with the input and output of – well, about anything. Take any program that produces some output, such as the `ls` program we used so much before. You can capture and otherwise manipulate the output to an unbelievable degree.

To begin with, you can *re-direct* the output to another file with the "re-direction" operator `>`:

```
$ ls -l > out
```

You will not see any output, because it has gone to a new file called "out".

This is also a convenient way to make a really short file, for example, a configuration file. Say you need a file that has the line

```
ENTRIES=100
```

as a configuration file for a (fictitious) program, you could just do

```
$ echo "ENTRIES=100" > config.file
```

which is, for just one or two lines, much faster than firing up your favorite editor. Of course, for longer files, or if you want to modify the contents of such a text file, you need to run the editor, but often you get by with the output redirection shown above.

1.2 stdout and stderr

Before we go on, lets explain that most programs have two distinct output streams by default, the "standard output" ("`stdout`") and the "standard error" ("`stderr`") stream. Usually, if you type commands interactively as we have so far, both of those streams put their output on your screen, and you will not really see a difference. The various forms of the `ls` command that we used before did not give any errors. Let's make one which actually does. We are using `ls` on a directory where we do not have read permission for everything:

```
 $ ls -l /proc/1/*
ls: cannot read symbolic link /proc/1/cwd: Permission denied
ls: cannot read symbolic link /proc/1/exe: Permission denied
ls: cannot read symbolic link /proc/1/root: Permission denied
-rw-r--r-- 1 root root 0 Mar 18 10:10 /proc/1/autogroup
-r-------- 1 root root 0 Mar 18 10:10 /proc/1/auxv
-r--r--r-- 1 root root 0 Mar 18 09:11 /proc/1/cgroup
--w------- 1 root root 0 Mar 18 10:10 /proc/1/clear_refs
-r--r--r-- 1 root root 0 Mar 18 09:11 /proc/1/cmdline
..... many lines deleted ....
```

You can see a number of "Permission denied" lines among the other output lines of `ls`. Those go to the `stderr` stream, while the "normal" lines go to `stdout`.

Let's say you do not care about the areas that you don't have access to. You can redirect all error messages to somewhere else, e.g. "nowhere" – the "null" device called `/dev/null` – by

```
 $ ls -l /proc/1/* 2>/dev/null
-rw-r--r-- 1 root root 0 Mar 18 10:10 /proc/1/autogroup
-r-------- 1 root root 0 Mar 18 10:10 /proc/1/auxv
-r--r--r-- 1 root root 0 Mar 18 09:11 /proc/1/cgroup
--w------- 1 root root 0 Mar 18 10:10 /proc/1/clear_refs
-r--r--r-- 1 root root 0 Mar 18 09:11 /proc/1/cmdline
-rw-r--r-- 1 root root 0 Mar 18 09:11 /proc/1/comm
-rw-r--r-- 1 root root 0 Mar 18 10:10 /proc/1/coredump_filter
```

with `2>` being the output re-direction for `stderr`.

If you want to see only the errors, redirect the regular output to `dev/null`. The `>` operator is the same as `1>`.

```
$ ls -l /proc/1/* >/dev/null
ls: cannot read symbolic link /proc/1/cwd: Permission denied
ls: cannot read symbolic link /proc/1/exe: Permission denied
ls: cannot read symbolic link /proc/1/root: Permission denied
ls: cannot open directory /proc/1/fd: Permission denied
ls: cannot open directory /proc/1/fdinfo: Permission denied
ls: cannot open directory /proc/1/ns: Permission denied
```

1.3 Pipes

The most powerful redirection of output (or input) are pipes, which are usually made with the "pipe" (`|`) symbol. Let me demonstrate this with the `wc` utility,

which can count characters, words, and lines of the input it receives. Let's see how much effort writing this very document has cost me so far (I am still adding to it, obviously):

```
$ wc shell.tex
   1012 6310 39827 shell.tex
```

So if have typed 1012 lines, 6310 words, and 39827 characters up to this point. The main power, however is that it will read its standard input. Let's say I want to know how many latex files I find on my computer:

```
$ find . -name ‘‘*.tex’’ | wc -l
     449
```

So the find command outputs one line per file it finds, and we pipe that output into `wc -l`, which then counts the lines, which equals the number of files found.

1.4 Dissecting a complex command

In 2014 I ran a huge physics simulation project on the "Open Science" computing grid. I simulated 5 trillion heavy-ion collisions and ended up using about 250 years of CPU time. I was required to give regular status updates throughout the 6-weeks long project, usually stating how many CPU-years I had used so far.

The project was divided up into about a half million individual *jobs* that simulate 10 million collisions each.

The easiest way to get the total CPU usage is to look at the log files of each job, which towards the end lists the CPU time actually used, among other statistics:

```
 (1) Normal termination (return value 0)
  Usr 0 19:35:42, Sys 0 02:10:44  -  Run Remote Usage
  Usr 0 00:00:00, Sys 0 00:00:00  -  Run Local Usage
  Usr 0 19:35:42, Sys 0 02:10:44  -  Total Remote Usage
  Usr 0 00:00:00, Sys 0 00:00:00  -  Total Local Usage
 1749596  -  Run Bytes Sent By Job
 5651231  -  Run Bytes Received By Job
 1749596  -  Total Bytes Sent By Job
 5651231  -  Total Bytes Received By Job
```

We are looking for the "Total Remote Usage", that is, the CPU time that was used on the remote system where the job actually executed.

I no longer have the original log files from that project, but will show the output from a similar, albeit much smaller, project.

So how do we go about extracting that time? We are starting with `grep` to extract that line in question:

```
$ grep "Total Remote Usage" pos_20_03.log
  Usr 0 19:35:42, Sys 0 02:10:44  -  Total Remote Usage
```

You may notice that there is a comma after the "Usr" time listing, which will be getting in the way for what's to come, so we eliminate it using the stream editor `sed`:

```
$ grep "Total Remote Usage" pos_20_03.log | sed -e 's/,//g'
  Usr 0 19:35:42 Sys 0 02:10:44  -  Total Remote Usage
```

Next we use `awk` twice in a row to extract the hour, minute, and second values. The "19:35:42" field is the third parameter, and "02:10:44" is the 6th. I checked that there was no file that indicated more than a day of CPU usage, so we can ignore the "days" fields which is always 0. We use `awk` to produce output with the 6 numbers in question delimited by a colon:

```
$ grep "Total Remote Usage" pos_20_03.log | sed -e 's/,//g' | awk '{print $3":"$6}'
19:35:42:02:10:44
```

The second `awk` statement uses those colons as the field delimiters, so the parameters are `$1`, `$2`, and so on. I have to start breaking the single command line across multiple lines here to show it properly:

```
$ grep ``Total Remote Usage'' pos_20_03.log | sed -e 's/,//g' | awk '{print $3'':''$6}' \
  | awk -F: '{X += ($1*3600 + $2*60 + $3 + $4*3600 + $5*60 +$6 )/3600} END {print X}'
21.7739
```

So the `($1*3600 + $2*60 + $3 + $4*3600 + $5*60 +$6 )` calculates the total number of seconds, and the division by 3600 yields hours.

That `awk` command is designed to take an arbitrary number of lines with those `19:35:42:02:10:44` entries and sum them up. That's what the `END` construct is for, have `awk` process all lines, and only at the end print that sum (the variable `X`).

So how do we get all those lines processed? We use the `find` command to get us all the files in question:

```
$ find . -name "*.log"
./pos_-01_-01.log
./pos_-01_-20.log
./pos_-01_-02.log
./pos_-01_-21.log
./pos_-01_-03.log
... -- lines deleted --
```

and execute our series of commands on each file found using the `-exec` construct:

```
$ find . -name "*.log" -exec grep "Total Remote Usage" {}  \; | sed -e 's/,//g' | awk '{print $3":"$6}'
11:37:44:01:03:55
12:42:52:00:50:45
11:13:14:01:46:36
17:00:27:02:09:53
11:19:38:01:44:59
08:41:46:00:47:50
08:37:59:00:46:26
... -- lines deleted --
```

and can now feed this result to the final `awk` command. The `{}` is, one by one, replaced with the name of the file that `find` turns up. The escaped semicolon (`\;`) delimits the command given to `find`, but it needs to be protected from being interpreted by the shell itself, hence the "`\;`".

And here is the full command, broken across multiple lines to make it more readable:

```
$ find . -name "*.log" -exec grep "Total Remote Usage" {}  \; | sed -e 's/,//g' \
  | awk '{print $3":"$6}' \
  | awk -F: '{X+=($1*3600 + $2*60 + $3 + $4*3600 + $5*60 +$6 )/3600} END {print X}'
14305.7
```

At the time, at a stage about 60% towards the project completion, I got

```
$ find . -name "*.log" -exec grep "Total Remote Usage" {}  \; | sed -e 's/,//g' \
  | awk '{print $3":"$6}' \
  | awk -F: '{X+=($1*3600 + $2*60 + $3 + $4*3600 + $5*60 +$6 )/3600} END {print X}'
1.34523e06
$ bc -lq
1345230 / 24
56051.25000000000000000000
1345230 / 24/365
153.56506849315068493150
```

So that was 56051.25 days, or 153.56 years of CPU usage at the time.

You see, string together different shell commands can make a daunting task rather easy.