UNIX command quick reference

Emma Tonkin & Greg Tourte

December 11, 2006

Contents

1 Getting to know the command line

1.1 What is a shell?

When you type commands on a command line, they are processed and interpreted by an application known as a *shell*. There are many shells available for UNIX systems (many of which are also available for Windows systems using a compatibility layer like Cygwin). Essentially, a shell is a simple scripting language. A set of shell commands can be typed into a shell manually, or they can be saved into a file and run at a later time – just like any other interpreted scripting language, such as Perl or Python (in fact, it is possible to use both of these languages as rather overcomplicated shells).

The most popular shell in the Linux/Unix world these days is *bash* ; on some very old or very cut-down systems you might find yourself using *sh*.

1.2 Typing in shell commands

The majority of command-line applications accept parameters and options. In UNIX, most programs accept parameters using either a verbose "–option-name" syntax, or a rather terser "-o" syntax.

1.3 Inbuilt help text

Some commands come with inbuilt help text. This is usually accessed by typing the command name, plus the string "–help" or, rather more unusually, " -h ".

For example, here is the result of trying this with the small word count application *wc*, which exists on most UNIX systems.

```
em@cyclops:~$ wc --help
Usage: wc [OPTION]... [FILE]...
Print newline, word, and byte counts for each FILE,
and a total line if more than one FILE is specified.
With no FILE, or when FILE is -, read standard input.

  -c, --bytes            print the byte counts
  -m, --chars            print the character counts
  -l, --lines            print the newline counts
  -L, --max-line-length  print the length of the longest line
  -w, --words            print the word counts
```

```
      --help     display this help and exit
      --version  output version information and exit
```

1.4 Man: Reading The Fine Manual

Most UNIX systems come with an application called *man*, which offers a set of documentation for the majority of available applications. *man* suffers from a number of limitations, principally that it tends to be written in an extremely telegraphic style. It is useful if you simply need a little help to jog your memory. Learning to read *man* pages is something of an art, but you will find that it becomes easier with time. The first thing to remember is not to be intimidated!

Here is an example of using *man* :

```
em@cyclops:~$ man wc
WC(1)           User Commands    WC(1)

NAME
       wc - print the number of newlines, words,
   and bytes in files

SYNOPSIS
       wc [OPTION]... [FILE]...

DESCRIPTION
       Print  newline, word, and byte counts for
   each FILE, and a total line if more than
   one FILE is specified.  With no FILE, or
   when FILE is -, read standard input.

...
```

1.4.1 Using man

As you saw above, *man* is easy to query ; just type *man commandname*.

1.4.2 The structure of man pages

Each man page comes with an introductory summary, a synopsis in which the basic use of the application is described, a detailed description of command line options, if any, and a little additional information such as the

name of the application's author(s) and copyright information.

2 Introducing the UNIX file system

Windows file systems are essentially based around the idea that every partition on a hard disk and every "device", which is to say, every CD-ROM, DVD burner, etc, gets its own drive letter. Similarly, if you view a network drive on Windows, it can be mapped to its own drive letter (alternatively, you can view it from the other computer; however, the point remains; drive letters are the way Windows likes to view the world).

UNIX, being much older and designed long before anybody really expected just one or two drives to do the trick, was designed with something else in mind. The UNIX authors wanted the filesystem to be spread over a large number of computers and an extended network, but they did not want this configuration to be obvious for the user – if it was, then changing the network at all would mean that the filesystem would apparently change!

So UNIX is designed with the idea that every computer has precisely one filesystem, and its name is /. /, pronounced "root", as in "the filesystem root", is exactly what it sounds like - the root of your filesystem tree.

In general, Unix/Linux shares a simple basic structure :

2.0.3 /bin

Really necessary programs live in /bin – like the command line shell *bash*, and handy utilities such as *cp* and *mv*.

2.0.4 /etc

System configuration files live in /etc.

2.0.5 /home

The area in which users' home directories are (typically) stored.

2.0.6 /lib

Contains library files (code that is reused within other applications).

2.0.7 /sbin

Where /sbin exists it is used for (more) really necessary programs. However, only programs that are used for system administration purposes live in /sbin – such as shutdown, kernel module manipulation functions and so on.

2.0.8 /tmp

Anybody can write files to /tmp; it's a general-purpose temporary file store.

2.0.9 /usr

Programs for users – not necessarily essential programs – live here in /usr/bin. Similarly, handy but not necessary resources live in /usr/share or /usr/local.

2.0.10 /var

/var is so named because it is the disk partition in which email and news data was traditionally stored. Because this sort of data appears and disappears, it is a file system of **var**iable size.

2.0.11 About . and ..

In UNIX, '.' is your current directory; *cd .* will simply change your directory. . . to the directory you are already in. . .

cd .. will change your directory to the directory above the one you are already in - that is, one directory nearer to the root directory.

2.0.12 About ∼

∼/ is your own home directory. If you type 'cd ∼' you will be returned to your home directory. Similarly, if you type 'cd ∼em/' you will be taken to em's home directory (you probably won't be able to read it, though!)

3 Pipes, stdin and stdout

When you type into the command line, you are introducing input into what is known as **stdin**, otherwise known as "standard input". When an application provides information back to you, it sends information to what is referred to as **stdout**. The important point to recognise here is that both stdin and stdout can be *redirected*.

3.1 Redirection of output to a file with >

To redirect the output of a command to a file, make use of the greater-than sign, ">". Here is an example of using ">".

```
ls > myfilelist.txt
```

Later, when you look into the file called "myfilelist.txt", you will find that it contains a file listing of the directory that you were in when you typed this command.

3.2 Piping output to another application using |

The interesting thing about stdin and stdout is that they can be sent, not only to file, but also to other applications. This is done using the "|" (pronounced 'pipe') redirector.

3.2.1 Example 1: how many files in a directory?

Using the application "ls", we can list all of the files in a directory. However, perhaps we only want to know how many files there are in that directory. Using the pipe redirector, we can accomplish this easily.

```
ls -l lists all of the files in that directory,
with one filename on each line.

wc -l counts all of the lines in a file.

We can chain them together as follows:

em@cyclops:~$ ls -l | wc -l
384
```

Analysing this example, *ls -l* actually gave the following output to stdout :

```
-rw-r--r--   1 em   users   3558095 2006-06-15 10:15 13.ogg
-rw-------   1 em   users    464704 2006-10-19 16:21 1467-8659.pdf
...etc...
```

When the pipe was used to redirect this to *wc -l*, wc read precisely this same text from stdin, counted each line and output a number when it ran out of anything more to count.

This is exactly equivalent to the following :

```
ls -l > temporaryfile.txt
wc -l temporaryfile.txt
```

4 Basic Linux command reference

4.1 A note about your $PATH

Linux makes use of a number of variables to customise your environment. One of them is the PATH variable. This is simply the number of directories into which Bash will look to find whatever application name you have typed on the command line. For example, on my laptop I have put a number of directories in my PATH for convenience, so that I do not have to type the full "address" of the application every time.

```
em@cyclops:~$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin:/usr/games:
/usr/lib/java/bin:/usr/lib/java/jre/bin:/usr/lib/java/bin:
/usr/lib/java/jre/bin:/opt/kde/bin:/usr/lib/qt/bin:
/usr/share/texmf/bin:.:/home/em/tree-tagger/bin:
/home/em/tree-tagger/cmd:
```

To find out what's in your own $PATH, type *echo $PATH*.

If you need to add a directory to your $PATH temporarily, you can do so by typing :

```
em@cyclops:~$ PATH=$PATH:/the/additional/directory/name:
```

4.2 Reading files with cat

The "cat" application is a very simple little application that simply opens a file and prints its contents to stdout.

```
cat myfile.txt
```

4.3 Reading files with less

Less is a very simple little application known as a pager. It succeeds slightly more basic pagers such as "more" (note : UNIX programmers tend to have a very strange sense of humour. Less was named this way because, as the designers said, "less is more"...

The major difference between *less* and *more* is that *less* allows you to page up and down using the Page Up and Down keys, and the arrow keys – *more* only allows you to page downwards. This is still a lot better than trying to read a file with *cat* as it flashes at super-speed down the screen, but

you will find that *less* is relatively painless to use. Sometimes, though, older installations or those more challenged for space will not have *less* installed at all, and you will have to use *more* after all.

4.4 Listing files with ls

For one of the simplest-sounding functions on Unix, *ls* has one of the longest man pages imaginable. This is partly because it gets used all over the place, in all sorts of scripts and interfaces; *ls* will produce output in colour, in any format, with any amount of useful or not-so-useful metadata. Nonetheless, the basic application of *ls* is really pretty simple :

4.4.1 Basic listing of all files in directory

```
ls
```

4.4.2 All files that finish in ".txt"

```
ls *.txt
```

4.4.3 A detailed file listing

```
ls -lisa
```

4.4.4 Reading *ls* output with a pager

```
ls -lisa | less
```

4.5 Editing files on the command line with pico

Pico is a relatively user-friendly little command line text editor. To edit a file with pico, simply type:

```
pico filename.txt
```

You will notice that pico provides a little help text along the lines of

```
^X Exit
```

This simply means "press control and X to exit the program".

4.6 Deleting files with rm

rm is a fantastically dangerous application. Apply with care!

Basic use of *rm* looks like the following :

```
rm filename.txt
```

Recursive use (that is, deletion of a directory) looks like :

```
rm -rf foldername
```

Be warned: rm is not as friendly as Windows' delete function, and generally it will not ask you whether you are sure, certain or completely happy with what you have done. There is no undelete function!

4.7 Creating empty files with touch

Touch is a very simple little application; if a file doesn't exist, it creates it. If it does, *touch* does nothing at all. This may sound pretty useless, and it is, until you realise that lots of programs demand files to exist, even if they're empty. You probably won't find this one very useful for your own work at the moment, but it is as well to know what it does in the event that you find it in a script or similar.

4.8 Finding out your login with whoami

You'll probably find this more useful when writing scripts than anywhere else, unless your short-term memory is a little in need of work. However, it's a very simple command :

```
em@cyclops:~$ whoami
em
```

4.9 Finding out your current directory with pwd

```
em@cyclops:~$ pwd
/home/em
```

4.10 File permissions with chmod

UNIX makes use of a permissions system to work out whether files should be considered :

- executable
- readable
- writable

chmod allows you to change these values for a given file or folder.

```
chmod 777 filename
```

```
equivalent:
```

```
chmod a+rwx filename
```

As you can see, there are two equivalent syntaxes. Each one is explained below.

0	no permissions
1	executable
2	writable
3	writable + executable
4	readable
5	readable and executable
6	readable and writable
7	readable, writable and executable

Why, one might ask, are there three 7s in the command *chmod 777 filename*? Because there are three distinct cases; the first is yourself, the owner of the file, the second is your user group, and the third is everybody who ever uses the machine. Similarly, then, if one typed *chmod 754 filename*, the result would be that you yourself see the file as readable, writable and executable; your user group sees it as readable and executable but not writable, and everybody else sees it as readable but not executable.

The alternative syntax for chmod is the following :

u	your own permissions
g	user group
o	the other users
a	all the users
+r	make the file readable
+w	make the file writable
+x	make the file executable

In other words, chmod u+x makes the file executable for the user; chmod o+rw makes the file readable and writable for all the other users (eg not yourself or your user group), chmod g+x makes the file executable for the user group. So u corresponds to the 7 in chmod 754 filename; g corresponds to the 5, and a corresponds to the 4. Which of these syntaxes you choose to use is entirely a question of which you personally feel more comfortable with.

It is worth bearing in mind that the latter syntax does not reset, whilst the former does; in other words, if you type 'chmod 644 *.txt', you will completely replace the permissions previously set. If you type 'chmod u+rw *.txt' you will not affect the group and general permissions at all.

4.10.1 Using chmod on web sites

In general, you want HTML files to be readable and writable for yourself, but you do not want others to be able to write them. So you might set them with the following : *chmod 644 *.html*.

4.10.2 Chmod and directories

Unlike files, directories have to be executable for you to change directory into them. In other words, if a directory was given the following permissions :

```
chmod 644 mydirectory
```

you would no longer be able to *cd* into that directory.

Directories must be set as executable and readable for you to be able to read them!

4.11 File ownership with chown

Every file belongs to somebody, and to some group. To see what user and what group a file belongs to, use *ls -l* :

```
em@cyclops:~$ ls -l
total 84
-rw-r--r--  1 em users  2991 2006-08-24 15:28 Author.java
```

As you can see, the permissions on this file (see *chmod*) are -rw-r–r–, otherwise known as 644; I can read and write the file, my user group and everybody else can only read it. However, you can also see that it is owned by "em" and that its group is "users".

As a user, you will not find yourself using chown. However, as a system administrator you might find yourself needing to do this; you would do so using the following command.

```
chown username:usergroup filename
```

So for example in my example, if I wished to change the owner to "bob", I would do :

```
chown bob:users Author.java
```

Both chown and chgrp have a "recursive" (-R) option. Use with caution!

4.12 Gaining administrator privileges with su

You may not need to do much of this for the time being. However, if you are given the root password of a server and need to do something you can't do as a user, like installing files into /usr/bin, then you will need to change user to root using *su*.

```
em@cyclops:~$ su -
Password:
root@cyclops@~#
```

That little - after su is important – it tells the system to provide a full root environment. If you do not do this you will find that you have very few applications in your $PATH.

4.13 Finding out what's running with ps

ps lists the currently running processes. For example (note that this particular usage of ps does not require a - before the options) :

```
ps ax | less
```

If you can't see the application running in here, the chances are that it has crashed!

4.14 If something goes horribly wrong – using kill

Sometimes applications get confused, and you won't be able to shut them down using ordinary means. In this case you have two options. If you know the exact name of the process, try :

```
killall -9 processname
```

If you do not, look for it using *ps*, and then kill it either by process name (as above) or by process id, using *kill* :

```
kill -9 5987
```

> Be careful with killall – use it only on Linux! It works completely differently on Linux than it does on classic UNIX systems.

4.15 Copying files with cp

cp is fairly simple :

```
cp filename.txt filename2.txt
```

If you need to copy folders, you need :

```
cp -a folder folder2
```

4.16 Moving files with mv

mv is even simpler than *cp*...

```
mv filename.txt filename2.txt
mv foldername foldername2
```

4.17 Date and time with date

date is a surprisingly useful program, especially if you are producing any sort of script; you will want to use it for getting a timestamp with which to log results, for example.

4.17.1 Human-readable date

```
em@cyclops:~$ date
Tue Nov 28 01:54:11 GMT 2006
```

4.17.2 ISO 8601

```
em@cyclops:~$ date --iso-8601
2006-11-28
```

4.17.3 Seconds since the epoch

```
em@cyclops:~$ date +%s
1164679060
```

This last is particularly interesting. The "seconds since the epoch" refers to the way in which UNIX systems internally measure time – the number of seconds that have elapsed since 1970-01-01 00:00:00 UTC. Whilst not very useful for a human, it is very helpful as a simple way of working with time. As a piece of trivia, this value also has the fun characteristic that, on older UNIX systems at least, this value is internally represented using a 32-bit integer. Thus, these systems will suffer a UNIX equivalent of the Millennium Bug in 2038-01-19 03:14:07 UTC, when they will roll back to 1970.

4.18 Compressing and uncompressing with gzip

Instead of PKZIP or similar, UNIX native compression format is gzip. It provides reasonably efficient compression, but can only compress one file at a time (Windows zips can, of course, contain many files). See "tar" for the Unix answer to this problem.

Basic use of gzip :

```
gzip filename.ext
gunzip filename.ext
```

4.19 Compressing and uncompressing with bzip2

Gzip is relatively old, and is designed for computational efficiency (speed) as well as for efficiency in terms of compression. Later, bzip2 was designed to make better use of the power of modern processors for more effective compression. It is used in very much the same way :

```
bzip2 filename.ext
bunzip2 filename.ext
```

4.20 Archiving with tar

Tar is a very old utility designed for use in the days that people still kept all their backups on magnetic tape. From this fact comes the name ; *tar*, the tape archiver. At heart, *tar* does a very simple job – it reads the contents of each file into one simple bytestream, which it then records serially onto tape (or, in this day and age, onto disk). This is a complicated way of saying that tar can turn an entire directory of files into one large lump, which can then be compressed and stored as just one big file.

Gzip-compressing a directory with tar :

```
tar cvfz mydirectory.tgz mydirectory/
```

Uncompressing a gzip compressed directory with tar :

```
tar xvfz mydirectory.tgz
```

Finding out what's in a gzip compressed tar file :

```
tar tvfz mydirectory.tgz
```

Using bzip2, replace the "z" in each of these examples with a "j"; thus :

```
tar cvfj mydirectory.tgz mydirectory/
```

4.20.1 Tar options explained

tar options :

c	create file
t	list files in archive
x	extract files
v	be verbose about what you are doing
f	operate on a file (as opposed to a stream)
z	make use of gzip compression
j	make use of bzip2

4.21 Checksums using md5sum

md5sum calculates and displays md5 hashes. This is generally used to check that the contents of a file have not changed; for example, that a downloaded disk image still contains what it does on the server. For this purpose, md5s are often published along with files. To calculate an md5sum :

```
em@cyclops:~$ md5sum review.doc
bf159362a24dc55a3b02ee5c4117d431  review.doc
```

4.22 Searching for matches using grep

Grep searches for strings and returns matches. Like most applications, it can either work from standard input (stdin) or directly from a file.

A couple of basic uses of grep :

Given that the file filename.txt contains the words "foo, bar, baz, boo, for, fork", the following uses would return :

grep -e 'f.o' filename.txt	foo
grep -e 'fo.' filename.txt	foo, for,fork
grep -e 'r$' filename.txt	for, bar
grep -e '^for$' filename.txt	for

Grep is a reasonably fast way of turning large, irrelevant files into small, useful result sets.

See the wildcard section for further info.

4.23 Finding files using locate

Locate is a database-backed system designed to help you find files on a large filesystem, quickly, by storing information in an centralised database that can be queried in seconds. Whilst this is much faster than searching all the way through the filesystem each time, it is also dependent on regular sweeps through the filesystem to refresh the database. Therefore, you will find locate useful only if you are looking for a file that is older than about a day – on most Linux systems, the process to rewrite the database runs about once a day, generally around 4am.

```
locate A*.java
locate Author*.java
locate *.pm
```

4.24 Finding files using find

Find is the slow way of finding files. You should not use it unless it is really necessary – it is much slower than locate. It is also something of a complex tool; however, here is a simple formula for finding files using find.

```
find . -iname 'Author.java'
```

The . means 'start in the current directory'. The -iname means 'something with a case-insensitive name like this' – so author.java or Author.java or AUThOR.jAVA all match.

4.25 Stream editing using sed

Sed provides you with a way of editing streams on the fly. For example, to change all uses of the word 'Blair' to the word 'Bliar' in a given piece of text, you would use the following command line :

```
cat filename.txt | sed -e 's/Blair/Bliar/g'
```

4.26 Downloading from the Web with wget

wget provides you with a means of downloading files directly from the command line. The simplest use of wget :

```
wget http://some.web.address/filename.txt
```

You can have much more fun with wget than this, however; for example, a surrealist viewpoint on the BBC website's HTML can be gained by doing the following :

```
wget -O - http://news.bbc.co.uk | sed -e 's/\ news/\ frogs/g'|less
```

Note: **wget -O -** causes wget to write directly to stdout.

An excerpt from the resulting HTML :

> <meta name="description" content="Visit BBC News for up-to-the-minute frogs, breaking frogs, video, audio and feature stories. BBC News provid es trusted World and UK frogs as well

as local and regional perspectives. Also entertainment, business, science, technology and health frogs." /> <meta name="keywords" content="BBC, News, BBC News, frogs online, world, uk, international, foreign, british, online, service" />

4.27 Making Bash scripts out of commands

4.27.1 The hash-bang

A Bash script is simply a set of commands, as we have discussed above. However, to execute them, the system needs to know what interpreter it should be using. Therefore, you provide this information using the following code as the first line of your script:

```
#!/bin/bash
```

Here for example is a very simple bash script :

```
#!/bin/bash
# this replaces ls -lisat
ls -lisat $1
```

Similarly, a Perl script has the following line :

```
#!/usr/bin/perl
print "I am a perl script!\n";
```

4.27.2 Example script: My age relative to the Epoch

```
#!/bin/bash
# this little script calculates the difference
# between my age and the current age of UNIX

MYBIRTH=‘date -d "1978-10-30" +%s‘
NOW=‘date +%s‘
echo $((${NOW}-${MYBIRTH}))
```

Instead of hardcoding my age into this script, I can use the $1 variable – which is set to the first argument on the command line.

```
#!/bin/bash
# this little script calculates the difference
# between any age and the current age of UNIX
# Invoke it with something like:
# ./myscript.sh 1982-01-01

MYBIRTH=‘date -d $1 +%s‘
NOW=‘date +%s‘
echo $((${NOW}-${MYBIRTH}))
```

4.27.3 Example script: Creating a wordlist from a file

```
#!/bin/bash
# Note: the following command is essentially one long line
# the \ tells bash to ignore the newline

cat $1 | tr ’A-Z’ ’a-z’ |  sed -e ’s/\ /\n/g’| \
sed -e ’s/[^A-Za-z\].//g’ | sed -e ’s/[\.,?]//g’ \
| sort | uniq | less
```

4.27.4 Example script: A basic spellchecker

Many UNIX systems come with some form of word list; this often lives in /usr/share/dict/words. This script demonstrates a use of the previous example script, combined with a search of /usr/share/dict/words, to identify words that do not appear in /usr/share/dict/words and may therefore be misspelt.

```
for foo in ‘cat $1 | tr ’A-Z’ ’a-z’ \
| sed -e ’s/\ /\n/g’ | \
sed -e ’s/[\.,?]//g’ | \
sed -e ’s/[^a-z\ ].//g’ ‘;

do
        MYRESPONSE=‘grep /usr/share/dict/words -e "^$foo$" | wc -l ‘
        if [ $MYRESPONSE -gt 0 ] ; then
                #echo "Word $foo OK"
                true
        else
                echo "Word $foo misspelt?"
        fi
done
```

Shell scripting (and scripting in general) is a powerful technique, even though it combines extremely simple elements.

5 Configuring and compiling software

Linux comes with a set of tools for the compilation, linking and installing of software. Each flavour of Linux (Red Hat, Slackware and so on) have their own package management system for installation of precompiled software; however, here, we are looking at how to compile software from scratch.

5.1 The compilation process

The process of compiling a piece of software from scratch is complex; software depends on libraries (kept in /lib, incidentally, on UNIX systems – on Windows systems, dlls do a similar job). Because of this, scripts are produced to simplify the compilation process.

Two pieces of software are used for this; *autoconf* and *make*.

5.2 autoconf

You will almost never find yourself using autoconf directly, unless you one day decide to produce and distribute a large piece of software. However, you will often make use of the result of it. Usually, when you download a set of source files to be compiled on Linux, you will find a file in that source directory called 'configure'. This is a script generated by autoconf that, when you call it by means of './configure' will automagically sort out all of the configurable aspects of the software – such as the location of required libraries, the preferred libraries against which to link and where to place the resulting files.

Sometimes you will want to give some command-line preferences to the ./configure script; for example, if you want it to try to install a program in a directory within your own home directory rather than attempting a system-wide install, you will want to tell the ./configure script to do so. For example, in this case you would write the following :

```
./configure --prefix=/home/em/bin/
```

To find out what command-line preferences exist, type *./configure –help*.

5.3 make

Generally, if a file called 'Makefile' is present in a directory, then the process of compiling and installing the software is as simple as :

```
make
make install
```

In practice, all sorts of things can go wrong. Makefiles are human-editable, however, so if the system is using the wrong compiler or any of the other non-fatal annoyances that frequently occur, it is possible to make some changes and try again.

6 Using Perl

6.1 What is Perl?

It is sometimes said to stand for the Practical Extraction and Reporting Language. In fact, this is a backronym.

6.2 Writing simple Perl

A beginner's introduction to Perl is available at :

http://www.perl.com/pub/a/2000/10/begperl1.html

6.3 Perl modules from CPAN

Perl is a nice scripting language, especially for manipulation of text files, web-based work and so on. It is something of a Swiss Army Knife, in that somebody with Perl experience can look at almost any problem in Web development and say something along the lines of "we could try using the left-handed corkscrew/awl attachment, I'm almost sure it'd do the trick" and be right most of the time. This being the case, attachments (known as *modules* in Perl parlance) have been developed for pretty much every task.

Since it is a waste of everybody's time to reinvent the wheel every time the task comes up, modules are generally shared online for everybody to test, reuse and mess around with. The website used for storing Perl modules is known as *CPAN*, which stands for something along the lines of "the Central Perl Archive Network", although I might be wrong about the details.

6.4 Manually installing Perl modules

Perl modules come packaged as a neat little tar.gz. First, extract the data using the following command :

```
tar xvfz filename.tar.gz
```

Then change directory into the directory created by this command (generally called something like module-name-1.1/). You will now have to prepare and install the Perl module. Perl comes with a simple system for doing this, which is based on the following three commands :

```
perl Makefile.PL
```

Perl will respond by writing something like "Writing Makefile for EPrints::QueryBuilder". If you then type 'ls' in this directory, you will see that a new file has been created. Logically, it's called "Makefile". You will recognise this as a configuration file for the GNU Make utility explained earlier in this document. As ever with Make, the sensible thing to do when you see this file is to see what happens when you type :

```
make
make test
make install
```

If any of these stages fail, you will not be able to install the module without working out why (in fact you can install the module even if "make test" fails, but there is not a great deal of point since the result will be that you will have installed a module that is broken in some way). To explain what is going on underneath the surface here, best practice for authors of Perl modules is to provide a set of "unit tests", which is to say, small tests of the behaviour/functionality of the compiled module on your system. If any of it does not work as planned, Perl knows that there is something wrong somewhere.

6.5 Auto-installing Perl modules from CPAN

This is a very easy way of doing what was just discussed in the previous section, automatically and somewhat faster – when it works.

6.5.1 How to auto-install Perl modules

```
perl -mCPAN -e 'install module::name'
```

```
For example:
```

```
perl -mCPAN -e 'install HTML::Mason'
```

6.5.2 What can go wrong?

Sometimes, Perl modules will depend on other modules that are not installed. Generally, the Perl module installer is smart enough to test for this. However, sometimes, if the author of the module has forgotten to place this information in the module configuration files, it will not. When this happens, Perl will not install the module (this would lead to having a broken module!) Instead, it will generally come up with a slightly cryptic-looking message about failed unit tests.

Detailed debugging is beyond the scope of this document. However, there are two things that you may wish to try ; a forced install, and a forced upgrade of installed Perl modules.

6.5.3 Further information

CPAN itself is at `http://www.cpan.net`. Additionally, try typing *man CPAN* for further information about the *perl -mCPAN* commands.

7 Wildcard/Regular expression cheat sheet

7.1 What are wildcards?

Wildcards are special characters interpreted by the shell.

7.2 Basic wildcards

*	matches anything
?	matches a single character
[abc]	matches a,b or c
[a-z]	matches any character from a to z
[0-9]	matches any number from 0-9
.{txt,pdf,doc}	matches any .txt, .pdf or .doc files

7.3 Grep-style regular expressions

You can test these out on /usr/share/dict/words... for example, try :

```
grep /usr/share/dict/words -e 'cat'
grep /usr/share/dict/words -e '^bure'
grep /usr/share/dict/words -e 'ity$'
grep /usr/share/dict/words -e '^fl.ng'
grep /usr/share/dict/words -e '[au]ggy$'
grep /usr/share/dict/words -e '[a-e]lity$'
grep /usr/share/dict/words -e '^[^a-y]'
```

^	matches the beginning of a word
$	matches the end of a word
.	matches any single character
[abc]	matches a, b or c
[a-z]	matches a-z
[A-Z]	matches A-Z
[0-9]	matches 0-9
[^a-y]	matches every line that does not start with letters a-y

Note: Regular expressions within Perl have a slightly different syntax and we will not cover them in this session!

For more about grep-style regular expressions, try :

http://www.robelle.com/library/smugbook/regexpr.html.