Unix is a Four Letter Word... and Vi is a Two Letter Abbreviation

Christopher C. Taylor

August 1993

Corrections and suggestions are welcomed by the author. He can be reached by electronic mail at *c.c.taylor@ieee.org* or by phone at 1–800–IDIOT–IQ.

1 Introduction

1.1 What is this thing?

> *In Samoa, when elementary schools were first established, the natives developed an absolute craze for arithmetical calculations. They laid aside their weapons and were to be seen going about armed with slate and pencil, setting sums and problems to one another and to European visitors. The Honourable Frederick Walpole declares that his visit to the beautiful island was positively embittered by ceaseless multiplication and division.— R. Briffault*

This document was not written to cause you to relive the experience of native Samoans or cause those around you to better empathize with Frederick Walpole. Rather, it was prepared to help acquaint new users with Unix and *vi* and provide a quick reference for me in case I forgot a how to do something. Although much of the material contained within has the same tantalizing appeal of arithmetic, please try to contain your excitement. This document is a chronicle of my adventures in the proverbial wonderland of Unix. Admittedly, minimal effort was made to accommodate a more diverse audience. It focuses on items which were of particular interest to me.

As our world draws closer and closer to its date with total randomness[1] things will change. It should not shock you to find that some of the things contained in this paper are no longer true. Try to deal with it.

1.2 Why read this thing?

I'm sure it is clear to the lazy person why a thorough knowledge of this document is ideal, but I will explain the rational for you hard workers. In my "many" years of experience it has become increasingly clear to me that the more you know the easier a given task becomes. A lazy person would benefit from reading this because they could perform a given task with less effort. A hard working person would benefit from reading this because they could perform a greater number of tasks in a given time period. Let's look at an example: suppose you wanted to move a file from one directory to another. Although this could be done by copying it to the new directory and then deleting it from the old, knowledge of the Unix command *mv* would allow you to accomplish the same task with only one command. Cutting your workload in half like this gives you the freedom to do half as much work as the idiot next to you[2] or get twice as much done. Please don't short change yourself by just skimming this document. Make a decision now to learn why every word in this document is here and what it means. Know it so well that you don't even need to think about it. I cannot begin to describe

[1] See the Second Law of Theromdynamics.
[2] Assuming you are not sitting by yourself.

the pleasure you will derive from this accomplishment. Consider the words of Alfred North Whitehead:[3]

> *It is a profoundly erroneous truism, repeated by all copy-books and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilization advances by extending the number of important operations which we can perform without thinking about them. Operations of thought are like cavalry charges in a battle—they are strictly limited in number, they require fresh horses, and must only be made at decisive moments.*

[3] For those of you who are easily persuaded, please don't take this paragraph too seriously.

Contents

2 Unix

2.1 Overview

Unix is an operating system designed at AT&T for their own personal use. The following electronic mail message from Dennis Ritchie may help explain who was responsible for Unix.

```
From: dmr@alice.att.com (Dennis Ritchie)
Subject: re: UNIX
Message-ID: <11613@alice.att.com>
Date: 14 Nov 90 05:53:03 GMT
Organization: AT&T Bell Laboratories, Murray Hill NJ

I read,

 > Looks like folks are now beginning to credit the development
 > of UNIX to Kernighan and Ritchie, but I thought the principal
 > investigators were *Thompson* and Ritchie.  Did something change?

The differences between Kernighan Ritchie Thompson are real
but very subtle. We all look alike (middle aged with scruffy
graying beards).  Note these distinctions:

-- Kernighan is slimmest, Ritchie middlest, Thompson heaviest
   in body build
-- Ritchie got contacts a couple of years ago and so is the
   only current non-glasses wearer
-- Thompson wouldn't touch netnews with a pole, Kernighan
   secretly gets misc.invest and misc.taxes mailed to him,
   Ritchie reads it more than is good for him and occasionally
   contributes
-- Ritchie is the only one who has met five people who have
   appeared on David Letterman (Penn, Teller, Rob Pike, Mayor Koch, and
   the guy who raised the biggest hog in Ohio)
-- Kernighan has written ten times as much readable prose as has
   Ritchie, Ritchie ten times as much as Thompson.  It's tempting
   to say that the reverse proportions hold for code, but
   in fact Kernighan and Ritchie are more nearly tied
   and Thompson wipes us both out.

        Dennis
```

Through a wild[4] series of events, Unix has become a standard operating system for many. Why else would you be reading this?

2.1.1 Case sensitivity

Unix is case sensitive. This means that Unix distinguishes between uppercase and lowercase letters, i.e. *Biff* and *biff* don't mean the same thing to Unix.

[4] Actually, I have no idea if it was wild or not, this is just a guess.

2.1.2 The shell

There are a number of different "flavors" of Unix available today. By different "flavors" I mean different command interpreters (called shells) which handle your input in their own unique way. This manual covers the C shell only. Many of the things found here will be identical with other shells, but don't count on it. It is possible to determine which shell is in use by typing `echo $SHELL`. The response for the C shell is */bin/csh* which is what you should get. One other popular shell is the Bourne shell which would respond with */bin/sh*.

2.1.3 Command syntax

Unix commands begin with a command name, often followed by flags and arguments some of which are optional. The generic syntax is:

```
command [flags] argument1 argument2 ...
```

Normally the flags are preceded by a hyphen to prevent them from being interpreted as a filename. For example, in the command line

```
ls -l avhrr
```

ls is the program called, *-l* is the flag, and *avhrr* is the argument. This command tells the computer to list (in long format) the file called *avhrr* or, if *avhrr* is a directory, to list all the files in the directory *avhrr*.

2.1.4 Correcting typos

There are three tools to destroy typos that occur on the command line. [5]

```
<DEL>       Erases the previous character.
<ctrl-W>    Erases the previous word.
<ctrl-U>    Erases the whole command line.
```

2.1.5 Controlling your terminal output

If output scrolls up on your terminal screen faster than you can read it, you can suspend it by typing <*ctrl*-`S`>. To resume the display, type <*ctrl*-`S`>. Again section 2.4.6 will discuss how to pass output through a paging program that will automatically display only one screen at a time. While I'm at it, <*ctrl*-`C`> will abort a process, and <*ctrl*-`O`> will discard the output until another <*ctrl*-`O`> is entered. Be sure to note that although the output doesn't appear, the process is still running. <*ctrl*-`Z`> suspends the current program. You can see its job number by typing *jobs*. You can resume the suspended program by giving the *fg* (foreground) command, or resume it in the background with *bg*.

[5] A couple notes on notation here. The delete key varies from keyboard to keyboard. Your delete key may be labeled *DEL, DELETE, BACKSPACE, RUB,* or *RUBOUT*. Also, <*ctrl*-`W`> means holding down the <*ctrl*> key and pressing `W`. Control <*ctrl*> character commands are not case sensitive, i.e. <*ctrl*-`w`> is equivalent to <*ctrl*-`W`>.

2.2 Files and directories

When you start a Unix session on a computer, you are placed in a directory that contains your files. This directory is called your *home* directory. My home directory is */home/cernan/taylor*. You can create, copy, move, and remove files as well as create subdirectories from here (see section 2.4).

2.2.1 Pathnames

There are a number of methods for specifying which directory and file you are interested in. Pathnames (the directory specification) can be relative or absolute. Absolute pathnames begin with a slash, `/`, and start at the root directory. Successive directories down the path are also separated by a slash. In the previous paragraph I gave the absolute pathname of my home directory. Each subdirectory is a branch in the directory tree.

A relative pathname begins with the directory you are in (commonly referred to as *working* directory) and moves downward to a lower directory. Relative pathnames begin with the name of the first directory below the working directory. Each lower directory down the path should have a slash in front of it. Assuming I was in the */home* directory, *cernan/taylor* would be the relative pathname to my home directory. A "." indicates the working directory, while ".." indicates the directory one level up (known as the *parent* directory). If I were in my home directory, the relative pathname for the */home* directory would be *../..* which says go to the "grandparent" directory two directories higher than you are now.

2.2.2 Naming files and directories

In general, file and directory names should be composed only of letters of the alphabet, digits, "." 's, and "_" 's. Be aware that files that begin with a "." do not appear in the directory list unless a special flag has been set when doing the list command.

The period is often used to add a suffix on to a base filename. For example, the source code for C programs have a *.c* suffix added to them, e.g. *prog.c*. Separating a filename by a "." is particularly useful when using wildcard selections (see section 2.7.1).

2.3 Online manual

All of the commands in Unix are described online in a collection of files. They are known as *man pages* because they were originally pages of the *UNIX Programmer's Manual*. There are eight sections of the man pages:

1. Commands
2. System calls

3. Library functions
4. Devices and device drivers
5. File formats
6. Games
7. Miscellaneous
8. System maintenance

If you know the name of a command, you can view its man pages by typing[6]

```
man [section] name
```

A program called *apropos*[7] is available for those who don't know the name of the command they want. The *apropos* program searches through the header lines of the man pages for whatever keyword you supply and displays a list of the man pages containing it. For example,

```
apropos copy
```

produces a list of all the man pages that contain *copy* in their header lines.

The list will contain commands followed by a number in parentheses, i.e. *cp (1)*. The number in parentheses is the section number. If the section number is omitted when doing a *man* command, the *man* program searches through each section until it finds the named man page. This works fine if the name is unique, but a few names exist in more than one section. One example of this is *intro*. There is an *intro* man page for each section. Typing `man intro` would get you the *intro* man page for the first section, but the only way to get the *intro* man page for section 5 is to type

```
man 5 intro
```

When the man pages are being displayed on your terminal, it pauses after each screen full and displays a `--More--` on the bottom line. This give you a chance to read the information before you go on to the next screen full. Press the space bar to scroll an entire screen forward.

2.4 Basic commands

The following few sections are devoted to many of the commands available in Unix. The descriptions are by no means complete. The most useful commands (at least to me) have descriptions that should suffice for the average user. However, less useful commands have rather terse summaries. If more information is desired for any of the commands, check the man pages. See section 2.3 on how to use the man pages.

[6] Portions of commands that are in square brackets, [ ], are optional.
[7] Typing `apropos` or `man -k` do the same thing.

2.4.1 Logging on (rlogin)

In order to use a computer operating under Unix you need to "log on". This attempts to protect against unauthorized use of the computer equipment. It also lets each user define their own personalized working environment on the same computer and even work on the same computer at the same time. The basic Unix command for remotely logging onto a computer is *rlogin* (remote login). To log onto a computer type `rlogin computername`. You are then asked to enter your account name, password, and then your charge code. Workstation consoles, as well as x-terminals, are ready for your account name, password, and charge code. The *rlogin* command is not needed. The characters do not appear when you type your password to promote confidentiality.

2.4.2 Changing your password (passwd)

Passwords are an important security measure. Don't neglect creating a "good" password. A good password should be easy to remember for you but hard for others to guess. Words in the dictionary, nicknames, and common chemical compound names are poor choices for a password. One way of generating a password is to use the first letter of each word in a strange yet memorable sentence. For example, *fatIwrnf* could be my password based on the sentence: *For a time I would recommend no forgery.*[8]

When you first receive your account you will probably be given a temporary password. You should change your password to something else. This is done with the *passwd* command. After typing `passwd`, you will be prompted first for your current password and then twice for a new password. Please note that this only effects the computer you are logged onto. You will need to repeat this ritual on every computer you have an account on.

2.4.3 Getting out (exit)

The command for ending a Unix session is *exit*. Another way to log out is to type <*ctrl*-D>. To avoid accidentally ending your Unix session with an inadvertent <*ctrl*-D>, type the command `set ignoreeof` at the beginning of each Unix session. Most lazy, or should I say efficient, people don't like doing this every time they log in. In section 2.6 we will discuss how to get around this threat to our slothfulness.

2.4.4 Listing files (ls)

The names of files and subdirectories can be displayed with the *ls* (list) command. Typing `ls` lists the files and subdirectories located in the working direc-

[8] This sentence is especially interesting because the number of letters in each word make up the constant pi to eight significant digits.

tory that don't begin with a ".". To see all the names, use the all flag, i.e. `ls -a`.

Other interesting flags for the *ls* command are:

```
-F      Marks directories with a trailing slash and executable files with
        a trailing asterisk.
-l      Lists in long format.  Gives all sorts of information.
-R      Recursively lists subdirectories encountered.
-s      Gives the size of each file.
-t      Sorts by time modified instead of by name.
```

It is possible to limit the scope of the files and subdirectories by using the wildcard characters discussed in section 2.7.1. For example, I would type

```
ls q*
```

if I wanted to list all the files and subdirectories that began with a *q*.

Note that the *ls* command lists files in the working directory only, unless you include the pathname to another directory whose filenames you want to list.

2.4.5 Catenate (cat)

Catenate means "to connect in a series." The *cat* command displays the contents of a file. If more than one file is placed in the command line, i.e. `cat yellow blue`, the files are displayed in succession. It is here that *cat* derives its name. With the use of the redirection operator (see section 2.7.2) two files can be placed into a single file. Typing

```
cat yellow blue > green
```

will cause *green* to contain the contents of *yellow* followed by the contents of *blue*.

2.4.6 Paging display system (more)

The *more* command provides a convenient alternative to displaying text on your terminal. The *more* program takes the input text and displays one screen full worth. The last line of the screen contains `--More--`. To scroll an entire screen forward, press the space bar. To scroll forward one line at a time, press <*return*>. To enter the *vi* text editor (see section 3), type `v`. To quit reading, type `q`.

More can be used on a text file by typing `more filename` or can be used to display the output from another program with the use of the pipe symbol (see section 2.7.3) by typing `command | more`.

2.4.7 Copying files (cp)

The *cp* (copy) command lets you duplicate a file of choice. Here is an explanation by examples:

```
cp cocoon butterfly
```

makes a duplicate of the file *cocoon* and gives it the name *butterfly*. Note that the filenames can include pathnames as well.

```
cp /home/cernan/taylor/tex/contract ../contract.bak
```

makes a copy of the file *contract* found in the */home/cernan/taylor/tex* directory and places it one directory level above the working directory in a file called *contract.bak*.

If */home/cernan/taylor/tex* is a directory, then

```
cp report /home/cernan/taylor/tex
```

will place a copy of *report* in the */home/cernan/taylor/tex* directory with the name *report*.

```
cp /home/cernan/taylor/tex/headlines .
```

will copy the file *headlines* in the */home/cernan/taylor/tex* directory into the working directory. The name will remain unchanged.

```
cp /home/cernan/taylor/tex/* .
```

will copy all the files (but not the subdirectories) in */home/cernan/taylor/tex* into the working directory. You can copy all the subdirectories in the directory and files contained in them by using the *-r* (recursive) flag as follows:

```
cp -r /home/cernan/taylor/tex/* .
```

Another useful flag is the *-i* (interactive) flag which prompts you if you are about to overwrite an existing file.

2.4.8 Removing files (rm)

The *rm* (remove) command deletes files that you no longer want. Just type `rm filename` to remove the file *filename*. If more than one filename is on the command line, i.e. `rm archaeologists date anything`, then the files *archaeologists*, *date*, and *anything* are removed.

2.4.9 Renaming and moving files and directories (mv)

The *mv* (move) command moves a file from the first argument to the second argument, e.g.

```
mv neatguy tidyguy
```

moves the contents of *neatguy* to the contents of *tidyguy*. This command reminds me of the time Chicago Bulls forward Stacey King said:

> I'll always remember this as the night that Michael Jordan and I combined to score 70 points.

after scoring one point in a game in which Jordan scored sixty-nine. All that really happened was that the file's name was changed. The reason it is called the *move* instead of *rename* or something like that is that you can include pathnames (just like you have done before in *cp* and *rm*). Including pathnames allows you to move a file into a different directory, hence the name. The *mv* command works on both filenames and directory names exactly the same way.

The *-i* flag works here just like it worked with *cp*. Setting this flag will prompt you before it moves a file on top of one that already exists.

WARNING: for ($i = 0$; $i <= 50$; i++) printf("Don't"); don't use wildcards (see section 2.7.1) with the *mv* command unless the destination is a directory. The *mv* command doesn't know what to do if you tell it to move a bunch of files into a single filename and so it will move all the files you selected on top of each other.

2.4.10 Navigating the directory tree (cd)

The *cd* (change directory) command does just what it says. It changes your working directory. The command syntax is

```
cd pathname
```

where the *pathname* can be either relative or absolute.[9]

2.4.11 Directory creation and destruction (mkdir and rmdir)

New directories are created with the *mkdir* (make directory) command and removed with the *rmdir* (remove directory) command. The syntax is

```
mkdir directory
```

and

[9] If these words don't make sense to you, you are either not paying attention or aren't reading this in order. I don't have a problem if you aren't reading this sequentially if you are willing to deal with some of this terminology ambiguity, but if you are struggling with the first problem, go back and reread section 1.2.

```
rmdir directory
```

The *pwd* (print working directory) command displays the absolute pathname of your working directory.

2.5 Additional commands

A number of additional commands are listed in this section. If you have forgotten what is in section 2.3, see section 1.2 and then section 2.3 for advice on how to get more information about the commands in this section.

2.5.1 alias

The *alias* command allows you to define shortcuts to save yourself time. In a sense, *alias* creates a link between a requested set of keystrokes and another set of keystrokes. For example, to use the *rm* command in interactive mode I would type

```
rm -i
```

By typing

```
alias rm 'rm -i'
```

the alias command would allow me to avoid typing the interactive flag, *-i*, every time a called the *rm* command.

The *alias* command defines a link between the first and the second arguments following the command. Whenever the first argument is entered at the command prompt, the Unix shell substitutes it with the second argument. An *alias* link stays in effect until the Unix session is ended or the link is "*unalias*ed". To destroy the link in my previous example I would type `unalias x`. The power of this command is more easily realized when used in a login file (see section 2.6).

2.5.2 ap

The *ap* (auto pilot) command has a deceptive name. It doesn't actually place the computer on auto pilot. The *ap* command reads your mind and attempts to perform the commands you want done. For example, thinking "I really wish I had a backup copy of the *tanana* image." will cause *ap* to input

```
cp tanana.* tanana_bak.*
```

to the Unix shell. Preceding a thought with "ignore" will cause *ap* to ignore your next thought. Although, with enough practice, the *ap* command can be a significant time saver, there are a few unresolved problems with this command.

1. I often change my mind while in the thinking process. In the previous example I may have decided later that I wanted to call the backup copy something else. No big deal here, *ap* just changes the filename but it isn't the most efficient use of computer resources.

2. All of the commands are echoed to the screen so that you know exactly what is going on. This is great as long as you remember to think "ignore" before you read each command. If you forget, the command will be executed again. This will continue until you remember to include the "ignore" flag or you think, "What is going on here?" which will cause the man pages for the particular command you are repeating to be displayed.

3. The *ap* command reads the strongest mind waves (known as *grey waves*) that it finds. If you have weak grey waves or your monitor is closer to someone else in your office, *ap* may listen to someone else's mind instead of yours. Also, walls do not provide insulation from grey waves, so if your monitor is near a wall, be prepared for some grey waves from minds on the other side of the wall to occasionally sneak in.

4. As you probably know, humans (you included) don't use their brains to their highest potential. In fact, many believe that we use as little as 5% of our brain's capacity. The problem here is that *ap* is only able to read around 80% of your mind. Unfortunately, many people use the 5% of their mind that *ap* can't read. When *ap* is called it scans your mind for activity, if none is found it prints the following cryptic error message:

   ```
   ap: Command not found.
   ```

   This indicates that it couldn't find a command in your head. Don't worry, this doesn't mean that you aren't thinking, it just means that you use the part of your brain that *ap* can't access.

2.5.3 biff

The *biff* command runs in the background and lets you know when electronic mail arrives. It was named after a dog at Berkeley that was known for barking at the mailman. To tell *biff* to bark at the mailman, type `biff y`. To tell *biff* not to bark at the mailman, type `biff n`.

2.5.4 chmod

Your files and directories have a number of attributes that are set when they are created. Listing the files with the *-l* flag, i.e. `ls -l`, displays the attributes of each file and directory in the working directory. Here is an example listing:

```
total 3
drwxr-xr-x  2 taylor        512 Aug  2 08:41 .
drwxrwxr-x 12 taylor       1024 Aug  2 08:41 ..
-rw-r--r--  1 taylor          5 Aug  2 08:41 blue
-rw-r--r--  1 taylor         12 Aug  2 08:41 green
-rw-r--r--  1 taylor          7 Aug  2 08:41 yellow
```

To the far left of each file or directory name are ten characters which show the attributes. The first column indicates whether the entry is a directory (`d`) or not (`-`). The other nine characters are organized into three groups of the three. The first group pertains to the owner (that would be you for your files). The second group pertains to people in your group, if you are in a group. The third group pertains to everyone else. Within each group of three are three characters. The first indicates read (`r`) permission. The second indicates write (`w`) permission. The third indicates execute (`x`) permission. If the permission is not present, a "-" will replace the *r*, *w*, or *x*.

The *chmod* (change mode) command lets you change the attributes on a file or directory. There are a number of forms, but I have chosen to cover the following syntax because of its similarity with *umask*.

```
chmod mode filename
```

where *mode* is a three digit octal number. The first digit pertains to the owners privileges. The second pertains to the groups privileges, and the third pertains to everyone elses privileges. Each octal digit is composed of the addition of three components. The read component is worth 4, the write component worth 2, and the execute component worth 1. Suppose I wanted the owner to have read, write, and execute privileges, the group to have read and write privileges, and everyone else to have read privileges only. The octal number I would use with *chmod* would be `764`.

2.5.5 compress/uncompress

The *compress* and *uncompress* commands compress a selected file using adaptive Lempel-Ziv coding to help conserve disk space. This technique almost always does a better job than the Huffman coding technique used by the *pack/unpack* commands. Typing `compress edison` would create a compressed file called *edison.Z* which could be resorted to its original condition by typing `uncompress edison` or `uncompress edison.Z`.

2.5.6 df

The *df* (disk free) command displays the amount of free disk space. This is often quite handy when determining if there is enough space to store an image on a particular hard drive. A quick glance at the man pages should indicate what flags should be set for the computer you are on.

2.5.7 diff

This program is useful in determining differences between two files or directories. It produces a list of lines that must be changed (c), appended (a), or deleted (d) to make the first file match the second. Lines from the first file are prefixed by "<" and lines from the second are prefixed by ">".

The *-b* option ignores trailing blanks and treats all other strings of blanks as equivalent. The *-i* option removes case sensitivity so that uppercase and lowercase letters match.

2.5.8 du

The *du* (disk usage) command displays the number of kilobytes consumed by each file and recursively provides results on directories. This can be useful for determining who the big disk space hogs are when you need more room. Typing

```
du -s *
```

from the parent directory of your home directory, e.g. */home/cernan*, gives a grand total of the kilobytes consumed by each user.

2.5.9 echo

The *echo* command echos a string to the terminal. One use for this command is in determining the contents of environment variables. Environment variables are variables that Unix keeps track of at the shell level. Two common examples are TERM and PATH. The TERM variable identifies what kind of terminal you are using. The PATH variable contains a list of pathnames to search through when looking for commands. More information on environment variables can be found in section 2.6.

To see the contents of the TERM variable type `echo $TERM`.

2.5.10 find

The *find* command recursively descends through the directory tree looking for files that match a logical expression. The *find* command has many options and is very powerful. Rather than go into detail here, I encourage you to take a look at the man pages for *find*. The *find* command does have a rather contorted syntax which is not easily mastered, and if truth be written, that's why I'm not spending more paper on it here.

2.5.11 finger

The *finger* command displays information about users. It can be used both locally and across the internet. For example,

```
finger taylor@en.ecn.purdue.edu
```

will display information about me from my computer account at Purdue University.

2.5.12 ftp

The easiest way to copy files from one disk to another is to use the *cp* command. However, often I am interested in copying files from one computer to another. The *ftp* command uses the File Transfer Protocol (FTP) to transfer data over a network connection.

To use *ftp* you open a connection to a remote computer and log onto that computer that can't access each others hard drives. The remote computer runs its own version of *ftp*, but you are in control of it. Within the *ftp* program you can list the files in your remote computer's directory, get copies of files on the remote computer, put copies of files from your computer onto the remote computer, and even delete files on the remote computer.

Here is an example of a FTP session:

```
ftp baboon                                                  (1)
Connected to baboon.ecn.purdue.edu.
220 baboon.ecn.purdue.edu FTP server (Version 4.178 Tue Jun 18 13:30:39
EST 1991) ready.
Name (baboon:taylor): taylor                                (2)
331 Password required for taylor.
Password: xxxxx
230 User taylor logged in.
ftp> cd tex/manual                                          (3)
250 CWD command successful.
ftp> get chap1.tex chap1.tex.bak                            (4)
200 PORT command successful.
150 ASCII data connection for chap1.tex (8612 bytes).
226 ASCII Transfer complete.
local: chap1.tex.bak remote: chap1.tex
8848 bytes received in 0.45 seconds (19 Kbytes/s)
ftp> quit                                                   (5)
221 Goodbye.
```

1. This starts the *ftp* program and tells it to open a connection with the computer called *baboon*.[10]

2. Here you need to type in the name of your account on the remote computer. If the name of your account on the remote computer is the same as the account on your local computer, you don't need to type in the account name but can just hit <*enter*>.

3. The *cd* command works like it does in Unix with one exception that we won't go into here.

[10] If you are attempting to open a connection with a computer outside of the Engineering Computer Network (ECN), you will need to include the entire internet address. In this case it would be *baboon.ecn.purdue.edu*.

4. This copies the file *chap1.tex* from the remote computer to *chap1.bak* in your local working directory. If no destination file is given the *get* command gives the file the same name on the local computer. The *put* command will send a file from the local computer to the remote computer. The *get* and *put* commands don't like wildcards. (See section 2.7.1 for a discussion of wildcards.) If you want to copy a number of files that have similar names but don't have the energy to type in all the names individually the suggestion of *mget* and *mput* may make you very happy that you read this manual.

5. Typing `quit` gets you out of the *ftp* program.

A short explanation of the available commands can be coaxed onto your screen by typing `help` at the `ftp`> prompt.

2.5.13 grep

The *grep* (get regular expression) program searches for an expression in a file or group of files. There are three versions: *grep*, *egrep* (extended grep), and *fgrep* (fixed-string *grep*). The *grep* program expands wildcard characters in the given expression. The *egrep* program searches for the expression including alternations. The *fgrep* program searches for fixed-strings only and does not expand wildcard characters. The *egrep* program has more sophisticated internal algorithms, and is usually faster than *grep* or *fgrep*. The syntax for all three versions is:

```
command [options] expression [file] ...
```

I have found these Unix commands to be very useful when programming. Suppose I had a C program with a number of subroutines and a global variable labeled *chuck_wivell*. Suppose further that Chuck found out about this and didn't like it. Of course I would change it immediately.

```
egrep chuck_wivell *.c
```

would give me a list of all files where the offensive variable manifested itself. By placing a `-n` option in the command line I could also obtain the line numbers of the offenses.

The wildcard characters that *grep* handles are

```
\ [ ] .  ^ * $
```

and a delimiter used to mark the beginning and end of an expression. Delimiters are necessary only if the expression contains blanks or wildcard characters. Here are a few examples to help solidify this potential mumbo-jumbo:

```
grep 'Nostalgia is not what it used to be' fft.c
```

searches through the file *fft.c* for the expression *Nostalgia is not what it used to be*.

The wildcard character "." matches any character. Therefore,

```
grep 'eur.' fft.c
```

would find *eureka*, *amateur*, *chauffeur*, etc... in the file *fft.c*.

Characters placed inside square brackets are each compared when searching.

```
grep '[cm]an' fft.c
```

would find any words with the sequence *can* or *man*, but would not locate sequences like *ran* or *and*. More can be found on the wildcard characters in section 2.7.1.

Preceding a wildcard character by a "\" turns off the wildcard character feature and the character is treated normally, i.e. the expression `eddie\.` would yield all the *eddie.*'s but not *eddies* or *eddieboy*.

Here are some useful options for all three of the *greps*:

```
-f    Matches all the expressions in a given file as opposed to
          the one typed in the command line.
-i    Removes case sensitivity so that uppercase and lowercase
          letters match.
-n    Displays the line numbers containing a match.
-l    Displays the names of the files that contain a match but
          not the lines that contained a match.
-v    Displays the lines that don't match as opposed to those that do.
```

2.5.14 history

The *history* command displays a list of commands you have previously typed. For this command to work correctly you must first type `set history=n` where *n* is the letter before *o* and the number of commands that should be saved. A peek at section 2.7.5 may help explain this.

2.5.15 kill

At times you may find that you have a job running that you don't want to continue. It is at this point that your thoughts may turn to murder. *Kill* is the hitman of choice for Unix users. *Kill* is quick and cheap (roughly 13 keystrokes). To put *kill* to work just type

```
kill -9 processid
```

where the *processid* can be found with the *ps* command.

If the process was created by the current interactive shell, you can type

```
kill -9 %n
```

where *n* is the process index indicated by the *jobs* command.

2.5.16 look

The *look* command searches through the system dictionary or lines in some other sorted list for a word. I often use *look* to check my spelling of a word. Suppose I want to know if *inoculate* is spelled correctly. I would type `look inoculate`. If *inoculate* is in the system dictionary (which it is) it is echoed to the terminal, and I know that the spelling is correct. If it is not in the system dictionary, it is not echoed to the terminal.

2.5.17 mail

Most users with access to computer accounts in a higher education setting and many in a corporate environment have access to internet. Your email address is *your_account_name@hostname.domain* where *hostname* is the name of the local computer and *domain* is the name of the "system" you are on. For example, *taylor@sunp.cr.usgs.gov* was my email address this past summer. In this case, *taylor* was my account name, *sunp* was the local computer name, and *cr.usgs.gov* was the name of the "system" I was on. I can read mail sent to me by logging on to the sunp computer and typing `mail`. Mail is sent to others by typing:

```
mail internet_address
```

where *internet_address* is the address of the person you wish to send a message to. You are then thrown into a very crude line editor that lets you type your message. Remember to hit <*return*> at the end of each line because it can't handle word wrapping. Typing a "`.`" or a <*ctrl*-D> on a line all by itself will signal the computer that you are finished with the message. The computer will then send the message you just wrote. If you wish to send a file rather than typing the message, use the following command:

```
mail internet_address < filename
```

where *filename* is the name of the file containing the message you wish to send. Section 2.7.2 covers the redirection (<) operator in more detail.

2.5.18 ps

The *ps* (process status) command displays the status of current processes. If no flags are set, the command displays only your processes. Take a look a the man pages to see what flags might be of interest to you. I usually use *-aux*.

2.5.19 script

The *script* command records, in a specified file, everything you type and every response you receive during your terminal session. To save the contents of your session in a file called *logsession*, type

```
script logsession
```

2.5.20 setenv

The *setenv* (set environment variable) command assigns values to environment variables. Many environment variables are used by different Unix programs. We will see some of these in section 2.6. It is also possible to define your own variables. To either define a new environment variable or change the value of an existing variable type

```
setenv variablename newvalue
```

For example, `setenv TERM vt100` assigns *vt100* to the variable *TERM*.

2.5.21 source

The *source* command sends the contents of a text file to the Unix shell. Suppose I have (and I do) a number of *alias* commands that I want typed in. Rather than typing them all in, I keep them stored in a file called (oddly enough) *.alias*. All I need to do is type

```
source .alias
```

and I have all my *alias* commands executed as if I had typed each one in separately.

2.5.22 spell

The *spell* command checks the spelling of all the words in a desired file against those in the system dictionary or some other file and outputs all the words that it couldn't find in the system dictionary. To check the spelling of the file *holy_cow* type

```
spell holy_cow
```

2.5.23 tar

The *tar* (tape archiver) program is useful for storing a bunch of files in one file (usually on a magnetic tape, but it doesn't have to be). The syntax for this command is

```
tar [key] [name ...]
```

where *key* is specified by a plethora of options (see abridged list below and unabridged list in the man pages) and *name* is either the file name or device name.

Here are some of the more commonly used keys:


```
c    Creates a new tape.
f    Used for taring to a tape.
t    Lists the contents of a tar file.
v    Turns verification on.
x    Extracts selected files.  If no file argument is given,
     the entire contents of the tar file is extracted.
```

Here is the syntax I use to create and read tar files:

```
tar cvf /dev/drivename directoryname   <-- creates
tar xvf /dev/drivename directoryname   <-- reads
```

2.5.24 telnet

The *telnet* command opens a connection to another computer via the internet network. This command allows you to log onto machines around the world that you have accounts on or that allow public access. For example, the University of Michigan offers public telnet access to weather information. To access this information type

```
telnet madlab.sprl.umich.edu 3000
```

2.5.25 umask

The *umask* command displays or sets the creation mask setting. The creation mask setting defines the default attributes for new files (see section 2.5.4). If no argument is included, *umask* displays the current setting. To change the creation mask setting type

```
umask value
```

where *value* is a three digit octal number similar to the one defined in section 2.5.4. It is important to note that this is a mask. This means that a *umask* setting of `022` would give the owner full privileges while the group and all others would not have write privileges. This is exactly opposite of what we saw in section 2.5.4 on *chmod*.

2.5.26 who

The *who* command simply tells you who is on the computer. Just type `who`.

2.5.27 A bunch more commands

The rest of the this subsection is a terse description of a few more Unix commands that you may find occasion to use.

awk — A pattern scanning and processing language.

bar — Creates a tape archive. (Similar to *tar*) *bg* — Moves a job into the background.

cal — Displays a calendar.

cc — Compiles C code.

chfn — Changes finger information.

clear — Clears your terminal's screen.

cmp — Preforms a byte-by-byte comparison of two files.

cut — Removes selected fields from each line of a file.

date — Displays or sets the date.

ed — The most basic line editor.

ex — A simple line editor. Also know as *e* or *edit*.

fg — Moves a job into the foreground.

file — Determines the type of a file by examining its contents.

fmt — Formats text.

hostname — Sets or prints the name of the current host computer.

jobs — Lists the current jobs in the shell.

make — Maintains, updates, and regenerates related programs and files.

mesg — Permits or denies messages on your terminal.

mt — Provides magnetic tape control.

od — Dumps octal, decimal, hexadecimal, or ascii representations of files.

pack/unpack — Similar to compress/uncompress, but uses Huffman coding.

paste — Joins corresponding lines of several files.

rev — Reverses the order of characters in each line.

rcp — Copies a file from a remote computer to the local computer.

rsh — Execute a remote shell command.

sed — A stream editor–quite powerful.

sort — Sorts and collates lines.

split — Splits a file into pieces.

stty — Sets or alters the options for a terminal.

tr — Translates characters.

uname — Displays the name of the current system.

units — Converts a number into different units.

uuencode/uudecode — Encodes/decodes a binary file into strictly ascii characters. (Useful for transmission via electronic mail)

write — Write a message to another user.

xget/xsend — Commands for sending/receiving secret mail.

2.6 Login files

Every time you log in, the Unix shell searches your home directory for certain files and executes the commands in them. This allows you to customize your Unix session. There are two initialization files that I will discuss here. The *.cshrc* (pronounced 'dot-see-shirk') file and the *.login* (pronounced 'dot-login') file. The *.cshrc* file is executed every time a new C shell is started. The *.login* is executed after the *.cshrc* file only when you initially log in. Generally, environment variables should be set in the *.login* file, and *alias* and *set* commands should be in the *.cshrc* file so that every new copy of the C shell will be able to use them.

2.6.1 The .cshrc file

The following is an example *.cshrc* file. The "#" character at the beginning of a line tells the C shell to ignore the rest of the line. I don't expect you to understand every command in this file or in the example *.login* file found in the next section, but I don't care to explain them all either. This document is getting too long as it is.

```
################################################################################
#                                                                              #
#         Example .cshrc file                                                  #
#                                                                              #
#         by Chris Taylor                                                      #
#                                                                              #
################################################################################

# Set path shell variable
# (See description of path in the paragraph followin this example .cshrc)
set path = ( /usr/bin /usr/local /usr/local/bin /usr/bin/X11 \
             /usr/ucb /usr/opt/bin ~ )

# Don't overwrite existing files with the redirection character ">"
set noclobber
```

```
# Don't create core dump files when a program blows up.
limit coredumpsize 0

# Check to see if this is an interactive shell.
# If not, skip the rest of this file.
if ($?USER == 0 || $?prompt == 0) exit

## Set C shell variables
# Remember my 40 most recent events
set history=40

# Save the most recent 40 events when I log out
set savehist=40

# Substitute the filename to be completed when I type an <ESC> at
# the command line
set filec

# Tells the shell to ignore .o files when trying to complete filenames
# when filec is set.  (This doesn't hold if the .o file is the only
# on that could be completed.
set fignore=.o

# Tells "filec" not to cry if it can't complete a file.
set nobeep

# Notify me when the status of background jobs change
set notify

# Don't let me log out by pressing <ctrl-d>
set ignoreeof

# Set TTY shell variable equal to the current terminal name
set TTY=`who am i | awk '{print $2}'`

# Allow others to send messages directly to my terminal
mesg y


# Set prompt to have the following form: [cmd#]cpu[directory]:
set cpu=`hostname | awk '{FS = "."; print $1}'` # set cpu = computer name
alias sp      set prompt='\[!\]$cpu\[$cwd\]:\ ' # set sp to set the prompt
alias cd      'chdir \!* ; sp'                  # redefine cd command
alias pd      'pushd \!* ; sp'                  # redefine pd command
alias pp      'popd \!* ; sp'                   # redefine pp command
sp                                              # set the prompt

# Shortcut aliases
alias c       'clear'
alias dict    'vi /usr/dict/words'
alias gv      'ghostview'
alias h       'history !* | head -39 | more'
alias laser   'lpr -Pmsa13 -h'
alias line    'lpr -Ped3'
alias ll      'ls -la'
```



```
alias ls      'ls -x'
alias mine    'chmod og-rwx'
alias pwd     'echo $cwd'       # This is faster than executing the pwd command
alias safe    'chmod a-w'
alias tmp     'cd /tmp/taylor'

# end of .cshrc file
```

A number of commands, i.e. *history*, *set*, etc... are built in commands. The rest of the commands must call an external program to execute it. Not all of these other commands are stored in the same directory. They are spread into a bunch of different directories. The *path* variable is a shell variable that tells the shell where to look for these commands. In the example *.cshrc* file, the *path* variable is set to

```
( . /usr/bin /usr/local /usr/local/bin /usr/bin/X11 /usr/ucb /usr/opt/bin ~ )
```

This tells the shell to look first in the working directory, then in the */usr/bin* directory, next in the */usr/local* directory, and so on until the file has been found or all directories have been looked at.

2.6.2 The .login file

The following is an example *.login* file. The same rules apply here as did with the *.cshrc* file.

```
################################################################################
#                                                                              #
#         Example .login file                                                  #
#                                                                              #
#         by Chris Taylor                                                      #
#                                                                              #
################################################################################

# Set erase, kill, and interrupt keys
stty crt erase '^H' kill '^U' intr '^C'

# Set the creation mask setting so that everyone can read my files
# but can't write to them
umask 022

## Set environment variables
# Set my terminal type to xterm
setenv TERM xterm

# Select vi as my editor of choice
setenv EDITOR /usr/ucb/vi

# Show the path to my mailbox
setenv MAIL /usr/spool/mail/$USER

# Set mail program
```

```
setenv MAILER /usr/ucb/mail

# Set paging program
setenv PAGER more

# Set default printer
setenv PRINTER hp

if (-f /bin/sun != 0) then
# Using a Sun
if ("`tty`" == "/dev/console") then
# Using console
setenv DISPLAY $cpu":0.0"
                # Ask if I want to start X11
echo ""; echo -n "Start up X11? "
set ans = $<
if ("$ans" != "n" && "$ans" != "N") then
# Start X11
setenv DISPLAY $HOST\:0
stty -tostop
exec xinit .xstartup ; kbd_mode -a
clear
endif
unset ans
else
setenv DISPLAY `last | grep $USER | head -1 |          \
                awk '{print $3}' | awk '{FS=".";print $1 "." $2 ":0" }'`
endif
endif

# end of .login file
```

2.7 Special characters

2.7.1 Wildcards

A number of characters are interpreted by the Unix shell before any other action takes place. These characters are known as wildcard characters. Usually these characters are used in place of filenames or directory names.

```
*     An asterisk matches any number of characters in a filename,
           including none.
?     The question mark matches any single character.
[ ]   Brackets enclose a set of characters, any one of
           which may match a single character at that position.
-     A hyphen used within [ ] denotes a range of characters.
~     A tilde at the beginning of a word expands to the name
           of your home directory.  If you append another user's login
           name to the character, it refers to that user's home directory.
```

Here are some examples:

1. `cat c*` displays any file whose name begins with *c* including the file *c*, if it exists.

2. `ls *.c` lists all files that have a *.c* extension.
3. `cp ../rmt?.` copies every file in the parent directory that is four characters long and begins with *rmt* to the working directory. (The names will remain the same.)
4. `ls rmt[34567]` lists every file that begins with *rmt* and has a *3*, *4*, *5*, *6*, or *7* at the end.
5. `ls rmt[3-7]` does exactly the same thing as the previous example.
6. `ls ~` lists your home directory.
7. `ls ~hessen` lists the home directory of the guy[11] with the user id *hessen*.

2.7.2 Redirecting output

A program that normally reads its input from the terminal (standard input) or normally writes its output to the terminal (standard output) may become annoying if you would rather send the input from a file instead of the keyboard or send the output to a file instead of the terminal. This annoyance can be avoided if you happen to be swift with the redirection operators. The redirection operators are "<", ">", and ">>". The first is used to send input to a command. The second is used to create a file and send the output to it. The third is used to append the output to an existing file.

An example of the first redirection operator was already given in section 2.5.17 on electronic mail. Suppose you wanted to put a list of all the people logged on into a file called *neatguys* with the current time listed at the top of the file.

```
date > neatguys
```

would create a file with the date and time in it, and

```
who >> neatguys
```

would append the list of users logged on.

2.7.3 Pipes

A pipeline is a convenient way to channel the output of one command into the input of another without creating an intermediate file. Let's say we wanted to get an alphabetical listing of the current processes. From a thorough study of the previous section and the man pages for *ps* and *sort*, we already know how to do this:

```
ps -aux > processes
sort processes
```

[11] Throughout this paper *guy* is assumed to be gender neutral unless otherwise stated.

This works, but it gives us a file (namely *processes*) which we don't want. The pipe symbol, "|" lets us bypass this intermediate file. The above two commands can be replace with the following:

```
ps -aux | sort
```

It is possible to connect a series of commands by additional pipe symbols. We could pass our previous output through the *more* paging program to obtain a more pleasing display of the results. This is a accomplish by typing

```
ps -aux | sort | more
```

One important point to recognize is that if a command isn't capable of reading from standard input, it cannot be placed to the right of a pipe symbol.

2.7.4 Quote characters

Sometimes it is necessary to place wildcards in the command line without having the shell treat them as special characters. This can be done by either preceding a single wildcard character with a backslash, \, or enclosing a sequence of wildcard characters in apostrophes, `' '`.

For example, if you wanted to set your C shell prompt to a question mark and typed

```
set prompt=?
```

the question mark would be expanded to be the first single-character filename in the working directory. If one exists it will be your prompt. If no single-character filenames exist, you will get a "set: No match" error. You should have typed

```
set prompt=\?
```

2.7.5 Other special characters

If you have set the history option (see section 2.6), you can use special characters to repeat those commands without retyping them. Here are some of them:[12]

```
!!          On a line by itself will repeat the most recent event.
!com        Will repeat the most recent event that begins with "com".
!?string    Will repeat the most recent event that contained "string".
!-n         Will repeat the nth previous event.
!n          Will repeat the nth event.  Type "history" to see the events
            numbered.
^old^new^   Will substitute "new" for the first occurrence of "old" in the
            most recent event, and repeats that event.
:           Will select specific words from an event line so you
            can repeat parts of an event, e.g.
```

[12] By event I mean one command line. This may be a single command, or it may include a number of commands in a pipeline, or whatever.


```
!?adam:s/adam/eve/

        will substitute "eve" for "adam" and repeat the last event
        with "adam" in it.
```

The semicolon, ";", separates commands. Typing

```
clear ; ls
```

is equivalent to typing each command on a separate command line.

The "`&`" symbol tells the shell to execute the command in the background. For example, typing `xid &` would execute XID in the background and give my Unix command line back so I could continue to use it even while XID was running.

The C shell also finds special meaning in the following:

```
"  `  {  }  #
```

Rather than explain the uses of these special characters, I caution you to avoid using them in filenames.

2.8 Miscellaneous tips

2.8.1 Removing files with strange names

There may come a time that you will discover that you have somehow created a file with a strange name that cannot be removed through conventional means. This section contains some unconventional approaches that may aid in removing such files.

Files that begin with a dash can be removed by typing

```
rm ./-filename
```

A couple other ways that may work are

```
rm -- -filename
```

and

```
rm - -filename
```

Now let's suppose that we an even nastier filename. One that I ran across this summer was a file with no filename. The solution I used to remove it was to type

```
rm -i *
```

This executes the *rm* command in interactive mode. I then answered "yes" to the query to remove the nameless file and "no" to all the other queries about the rest of the files.

Another method I could have used would be to obtain the *inode* number of the nameless file with

```
ls -i
```

and then type

```
find . -inum number -ok rm '{}' \;
```

where *number* is the inode number.

The *-ok* flag causes a confirmation prompt to be displayed. If you would rather live on the edge and not be bothered with the prompting, you can use *-exec* in place of *-ok*.

Suppose you didn't want to remove the file with the funny name, but wanted to rename it so that you could access it more readily. This can be accomplished by following the previous procedure with the following modification to the *find* command:

```
find . -inum number -ok mv '{}' new_filename \;
```

2.8.2 Wildcards beyond the working directory

Let's say we want to perform some command on a set of files in the working directory and all the directories below it. What if there was a Hewlett-Packard advertisement that asked, "What if I had a slew of subdirectories containing mounds of C source code, and I wanted to copy all of the library files (files with a *.h* extension) into a separate directory called *library*. How could I do it?" If you had read the next line, you would respond immediately with the following:[13]

```
cp `find . -name '*.h' -print` library
```

2.8.3 Terminal input in a shell script

To input text from your terminal into a C shell script use the following syntax:

```
while ( 1 )
    set line = "$<"
    if ( "$line" == "" ) break
    ...
end
```

Also, be advised that the C shell has no way of distinguishing between a blank line and an end-of-file.

[13] That is, if you talk to your television.

2.8.4 Remote shell trick

Here is the proper syntax to use the *rsh* (remote shell) command without having the remote shell remain active until the remote command is completed.

```
rsh machine -n 'command >&/dev/null </dev/null &'
```

where *machine* is the name of the remote computer and *command* is the remote command to be performed.

This works because the *-n* flag attaches the *rsh*'s standard input to */dev/null* so you can execute the complete *rsh* command in the background of the local computer. Also, the input/output redirections on the remote computer (the stuff inside the single quotes) makes *rsh* think the session can be terminated since there is no data flow. In all truth, you don't have to use */dev/null*. Any filename will work.

2.8.5 Loops in scripts

Here is an example of a simple loop in a script. I use it to send out my biweekly junkmail messages.[14]

```
#!/bin/sh
for i in `cat $HOME/jm/list`
do
mail -s 'Junkmail message number '$1 $i < jm.$1
done
```

The script takes one line at a time from the file *$HOME/jm/list* and executes the command

```
mail -s 'Junkmail message number '$1 $i < jm.$1
```

where *$1* the the first argument on the command line calling the script and *$i* is the line from the file *$HOME/jm/list*.

2.8.6 More tricks

Every word of a file can be placed on a separate line by typing

```
cat old_filename | tr -cs A-Za-z '\012' > new_filename
```

The following lists all words in *filename* in alphabetical order.

```
cat filename | tr -cs A-Za-z '\012' | tr A-Z a-z | sort | uniq
```

You can find out when the file *.rhosts* was last modified by typing

```
echo .rhosts last modified on `/bin/ls -l .rhosts | cut -c33-44`
```

Typing `head -n` displays the first *n* lines of a file. And typing `last` lists the last logins.

[14] Send me mail at *taylor@ecn.purdue.edu* for more information on this service.

2.9 Things to try

Just for kicks, I have included a few commands for you to try typing in at the shell prompt. Make sure you type each line exactly as it appears here.

1. `If I had a ( for every $ Congress spent, what would I have?`
2. `[Where are all those MIAs?`
3. echo '[q]sa[ln0=aln256%Pln256/snlbx]sb3135071790101768542287578439snlbxq' —dc
4. `got a light?`
5. `man:  Why did you get a divorce?`
6. `make 'heads or tails of all this'`

Note: The auto pilot command found in section 2.5.2 doesn't exist.

3 Vi — Text Editing

3.1 Overview

Vi (officially pronounced 'vee-eye'/unofficially pronounced 'six' because of the feeling one gets when using *vi* that it may be the text editor of the antichrist) is a display oriented interactive text editor.

Vi[15] makes one major philosophical deviation from every other text editor I have come in contact with. The basic idea is that your hands don't have any business straying from the home row keys. This can be an advantage for the touch typist, but the guy who needs to see the letter on the key before pushing it down tends to be less enamored with this characteristic. Since your fingers can only reach about fifty (50) keys without moving your hands, and since *vi* has in excess of 100 commands, something drastic must be done in order to designate all the functions a decent text editor must have. Rather than relying on extra keys on your keyboard that seem a little too far away or special key combinations that involve keys that your keyboard may or may not have, *vi* simply assigns a couple functions to the keys in reach.

Vi operates in two modes[16] (*insert* and *command*) in order to determine which function should be performed when a key is pressed. This two mode novelty, in my opinion, is what causes some to confuse *vi* with the devil himself and causes others to place *vi* equal with God. Although I find it hard to justify worshiping a text editor, I can (after much effort) appreciate the utility of *vi*.

Why *vi*? *Vi* is the default editor for Unix. It is possible to use other editors, but if you learn *vi* you can be confident that it will be on any Unix machine you use. However, the same level of confidence with another editor may be shortlived. It would be to your advantage to learn at least the basics of *vi*. For those who use a text editor on a daily basis, particularly for programming, *vi* will become a joy to use after a few months of friendship building.

As with any friendship, an emphasis must be placed on quality time, not just quantity. Those who have little use for a text editor may be satisfied with a cold professional relationship with conversations limited to a few basic commands. The rest of us would certainly benefit from a little quality time with *vi*. By quality time, I don't mean merely having the same conversations over and over (repeating commands you already know). I don't mean just reading about what makes *vi* tick. Although these are important activities, I mean telling *vi* things you've never told it before and observing its response. Don't discuss important issues with *vi* until you're pretty sure you know how it will react. Make sure you make a backup copy of the file you experiment with.

---

[15] Short for *visual*, but in keeping with the Unix mentality that any command longer than five characters isn't worth using, the command has been truncated to the first two letters.

[16] Technically, there are three modes, but I have chosen to treat the *command* and *line editor* modes as one mode. The *line editor* commands are a carry-over from the *ex* line editor.

3.2 Starting vi

To start *vi* just type `vi` at the operating system prompt. You will see a screen with a column of tildes (~) down the left side of the screen. This signifies an empty workspace. To edit a file, just include the filename after it, e.g. `vi filename`. You will see the text of the file you included.[17] *Vi* is now in command mode. The most basic command to enter insert mode is `i` which lets you insert text to the left of the cursor.

3.3 Insert mode

I begin with a brief description of the insert mode because it is very straight forward. In insert mode the characters you type are inserted into your document. You can use the backspace key to delete any typing mistakes you have made on the current input line. The escape key (<*esc*>) takes you out of insert mode and back to the command mode. If you are ever in doubt about what mode you are in, just press <*esc*> a few times until *vi* starts complaining. You will then know that you are in the command mode.

3.4 Command mode

Command mode is where you do everything that isn't done in insert mode. In command mode the same keys that caused letters to appear on your screen in insert mode now represent totally different functions. Rather than go into a detailed discussion of the 100 or so commands, this section contains a list of the more popular commands. The next section contains a more comprehensive list of *vi* commands.

3.4.1 Moving around

```
h         move the cursor one character to the left
j         move the cursor one character down
k         move the cursor one character up
l         move the cursor one character to the right
0         move to the beginning of a line
$         move to the end of a line
G         move to the end of a file
1G        move to the first line of a file
<ctrl-F>  move down one screen
<ctrl-B>  move up one screen
```

If you try to move somewhere that *vi* doesn't want to go, e.g. pressing `h` when the cursor is in the left-most column, your terminal will complain by either beeping or flashing the screen.

[17] If the file does not exist, it will be created.

3.4.2 Deleting text

```
x     delete the character under the cursor
dd    delete a line
```

3.4.3 Saving and quitting

```
:w    write to disk
ZZ    write to disk and exit
:q!   exit without writing to disk
```

Actually, the command for quitting *vi* is `:q`. You can save and quit by typing `:wq` but `ZZ` does the same thing[18] and takes one less keystroke. If there are unsaved changes to the text and you try to quit using `:q`, *vi* will warn you that you have unsaved changes and will prevent you from quitting. In order to quit without saving the changes you must use the override switch, `!`.

3.4.4 Copy, delete, and moving text

Knowing how to copy, delete, and move text is a prerequisite to any serious text editing task. If you have a small amount of text to delete or copy you may find it convenient to use *dd* or *yy*. These commands delete or yank the line of text that the cursor is on. *dd* deletes the current line of text and places it in a buffer. *yy* copies the current line of text to a buffer while leaving the original text unaltered. Many commands can be preceded by a number. This number indicates the number of times the command is repeated. These commands are no exception, e.g. *3dd* deletes the line the cursor is on as well as the two lines below it and places them in a buffer. Text can be retrieved from the text buffer by typing `p`. The "pull" command inserts the text from the buffer into the text file beginning on the line below the cursor. This method of deleting and yanking works well for text blocks of known length or an easily countable number of lines, but is less satisfactory of large blocks of text.

As a result, *vi* has another method of text manipulation that involves marking text. *Vi* is capable of marking 26 different locations in a file. To mark a location in a text file move the cursor to the desired location and type `m` followed by the name you want to use. Each lowercase letter of the alphabet is a name.

Suppose we have a portion of text we wish to move from one location to another. We can do this by marking the beginning of the text block with the name *q*, i.e. `mq` will give the current cursor location the name *q*. Then we move to the end of the portion of text we wish to move and type `d'q`. This command deletes everything from the marked position to the cursor position and places it in a buffer. Text in the buffer is retrieved using the "pull" command already described. The "yank" command allows you to copy the text to the buffer

[18] For the pureist, `:wq` and `ZZ` are not exactly the same. `:wq` always saves, whereas `ZZ` saves only if changes have been made since the last save.

instead of deleting it. Typing `y'q` instead of `d'q` will place a copy of the text in the buffer and leave the original text unaltered.[19]

3.4.5 Search and replace

The search command is `/`. To search for *polite* type `/polite`. `n` repeats the search in the same direction, and `N` repeats the search in the opposite direction.

The search option accepts most of the standard Unix pattern matching language. (See section 2.7.1.) Suppose I had a file that contained the following text:

> *There was a young man of Milan*
> *Whose poems, they never would scan;*
> *When asked why it was,*
> *He said, 'It's because*
> *I always try to get as many words into the last line as I possibly can'.*
> *—anonymous*

Here are a few examples (using this text) that you will probably never use but may find inspiring:

```
/[a-z]as
```

will search for any lowercase letter followed by *as*. In this example, it would find *was* and *last* but not *as* or *asked*.

```
/[^c]an
```

will search for any *an* preceded by any character other than a *c*. In our text it would find *Milan* but not *scan* or *can*.

```
/^[A-Z].*\. *$
```

will search for any line that begins with a capital letter and ends with a period and any number of blanks. Our only match in the example text would be with the last line.

All of these search patterns can be used in the search and replace command that takes on the following structure:

```
:s/search_string/replacement_string/g
```

This command replaces every *search_string* on the current line with *replacement_string*. Omitting the `g` (global) flag at the end of the command will cause only the first occurrence of *search_string* to be altered. Often you may wish to confirm each replacement. This can be done with the confirm flag `c`. The confirm flag should be placed after or in place of the `g` flag. Suppose I had the following line:

[19] Using `'` in `d'q` begins deleting text at the beginning of the line that the `q` mark is on. Using `` ` `` instead, i.e. `` d`q `` begins deleting text at the exact location of the mark. This holds for the yank command as well.

> *Give a skeptic and inch... and he'll take a mile.*

and typed

```
:s/take a mile/measure it/
```

I would be left with

> *Give a skeptic and inch... and he'll measure it.*

Any command that begins with a "`:`" is called a line mode command and performs its duty on the line the cursor is currently on. However, you can override *vi*'s default of operating only on the current line by preceding them with a range of line numbers. For example, if I wanted to replace *guy* with *gal* on lines 32 through 56 I would type

```
:32,56s/guy/gal/g
```

Omitting the `g` would cause only the first occurrence of *guy* in each line to be replaced. The "`.`" and "`$`" play a special role in this sort of designation. "`.`" indicates the current line, and "`$`" indicates the last line of the file. Therefore, if I wanted to delete[20] from the current line to the end of the file I would enter:[21]

```
:.,$d
```

I could even do something like:

```
:.,/Edison/d
```

which would delete from the current line to the next line that contained *Edison*.

One other shortcut that might be worth mentioning is that `1,$` and `%` both indicate all the lines in the file. Therefore,

```
:1,$s/search_string/replacement_string/g
```

and

```
:%s/search_string/replacement_string/g
```

do exactly the same thing.

3.4.6 Undo

The undo command, `u`, is another feature that has saved me many times. Pressing `u` undoes the last command you told *vi* to perform. Another form of the undo command is `U` which undoes all the changes made to the current line since you moved there.

[20] This works because `:d` is a line mode command that deletes the current line.
[21] The same could be accomplished by typing `dG`.

3.4.7 Repeat

Often times you may desire to repeat the last command performed. This can be done with the "`.`" command. Place the cursor in the appropriate position and press "`.`" to repeat the most recent command. Suppose I had a C program in which I wished to switch a variable name from *no_way* to *yes_way* in two different places. One way I could accomplish this would be to place my cursor on the beginning of the first *no_way* and type `cw` (change word) and then type `yes_way` <*esc*>. This would accomplish my task for the first case. Now all I would need to do to change the second *no_way* would be to place my cursor at the beginning of it and type "`.`" to repeat the last command.

3.5 Vi reference

Here is a more comprehensive list of *vi* commands in command mode.[22] [23]

3.5.1 Notation for this reference

```
default values        : 1
<*>                   : `*' must not be taken literally
[*]                   : `*' is optional
^X                    : <ctrl-X>
<sp>                  : Space
<cr>                  : Carriage return
<lf>                  : Linefeed
<ht>                  : Horizontal tab
<esc>                 : Escape
<erase>               : Your erase character
<kill>                : Your kill character
<intr>                : Your interrupt character
<a-z>                 : An element in the range
N                     : Number (`*' = allowed, `-' = not appropriate)
CHAR                  : Char unequal to <ht>|<sp>
WORD                  : Word followed by <ht>|<sp>|<lf>
```

3.5.2 Move commands (See also Display commands)

```
 N | Command             | Meaning
---+---------------------+----------------------------------------------
 * | h | ^H | <erase>    | <*> chars to the left.
 * | j | <lf> | ^N       | <*> lines downward.
 * | l | <sp>            | <*> chars to the right.
 * | k | ^P              | <*> lines upward.
 * | $                   | To the end of line <*> from the cursor.
 - | ^                   | To the first CHAR of the line.
 * | _                   | To the first CHAR <*> - 1 lines lower.
 * | -                   | To the first CHAR <*> lines higher.
```

[22] Heavily borrowed from *Vi Reference* by Maarten Litmaath.
[23] Warning: some *vi* versions don't support the more esoteric features described in this document.


```
 * | + | <cr>            | To the first CHAR <*> lines lower.
 - | 0                   | To the first char of the line.
 * | |                   | To column <*> (<ht>: only to the endpoint).
 * | f<char>             | <*> <char>s to the right (find).
 * | t<char>             | Till before <*> <char>s to the right.
 * | F<char>             | <*> <char>s to the left.
 * | T<char>             | Till after <*> <char>s to the left.
 * | ;                   | Repeat latest `f'|`t'|`F'|`T' <*> times.
 * | ,                   | Idem in opposite direction.
 * | w                   | <*> words forward.
 * | W                   | <*> WORDS forward.
 * | b                   | <*> words backward.
 * | B                   | <*> WORDS backward.
 * | e                   | To the end of word <*> forward.
 * | E                   | To the end of WORD <*> forward.
 * | G                   | Go to line <*> (default EOF).
 * | H                   | To line <*> from top of the screen (home).
 * | L                   | To line <*> from bottom of the screen (last).
 - | M                   | To the middle line of the screen.
 * | )                   | <*> sentences forward.
 * | (                   | <*> sentences backward.
 * | }                   | <*> paragraphs forward.
 * | {                   | <*> paragraphs backward.
 - | ]]                  | To the next section (default EOF).
 - | [[                  | To the previous section (default begin of file).
 - | `<a-z>              | To the mark.
 - | '<a-z>              | To the first CHAR of the line with the mark.
 - | ``                  | To the cursor position before the latest absolute
                         |   jump (of which are examples `/' and `G').
 - | ''                  | To the first CHAR of the line on which the cursor
                         |   was placed before the latest absolute jump.
 - | /<string>           | To the next occurrence of <string>.
 - | /<string>/+n        | To the next nth occurrence of <string>.
 - | ?<string>           | To the previous occurrence of <string>.
 - | n                   | Repeat latest `/'|`?' (next).
 - | N                   | Idem in opposite direction.
 - | %                   | Find the next bracket and go to its match
                         |   (also with `{'|`}' and `['|`]').
```

Entries in this table that have an "`*`" are repeatable commands whereas entries with a "`-`" are not. For example, if I type `4w` my cursor will travel four words farther into my document.

3.5.3 Searching

```
:ta <name>              | Search in the tags file[s] where <name> is
                        |   defined (file, line), and go to it.
^]                      | Use the name under the cursor in a `:ta' command.
^T                      | Pop the previous tag off the tagstack and return
                        |   to its position.
:[x,y]g/<string>/<cmd>  | Search globally [from line x to y] for <string>
                        |   and execute the `ex' <cmd> on each occurrence.
:[x,y]v/<string>/<cmd>  | Execute <cmd> on the lines that don't match.
```

3.5.4 Undoing changes

```
u                       | Undo the latest change.
U                       | Undo all changes on a line, while not having
                        |   moved off it (unfortunately).
:q!                     | Quit vi without writing.
:e!                     | Re-edit a messed-up file.
```

3.5.5 Inserting text

End inserting text with $<esc>$

```
 * | a                  | <*> times after the cursor.
 * | A                  | <*> times at the end of line.
 * | i                  | <*> times before the cursor (insert).
 * | I                  | <*> times before the first CHAR of the line
 * | o                  | On a new line below the current (open).
                        |   The count is only useful on a slow terminal.
 * | O                  | On a new line above the current.
                        |   The count is only useful on a slow terminal.
 * | ><move>            | Shift the lines described by <*><move> one
                        |   shiftwidth to the right.
 * | >>                 | Shift <*> lines one shiftwidth to the right.
 * | ["<a-zA-Z1-9>]p    | Put the contents of the (default undo) buffer
                        |   <*> times after the cursor.
                        |   A buffer containing lines is put only once,
                        |   below the current line.
 * | ["<a-zA-Z1-9>]P    | Put the contents of the (default undo) buffer
                        |   <*> times before the cursor.
                        |   A buffer containing lines is put only once,
                        |   above the current line.
 * | .                  | Repeat previous command <*> times.  If the last
                        |   command before a `.' command references a
                        |   numbered buffer, the buffer number is
                        |   incremented first (and the count is ignored):
                        |
                        |   "1pu.u.u.u.u      - `walk through' buffers 1
                        |                       through 5
                        |   "1P....           - restore them
```

3.5.6 Deleting text

Everything deleted can be stored into a buffer. This is achieved by putting a `"` and a lowercase letter before the delete command. The deleted text will be in the buffer with the used letter. If an uppercase letter is used as buffer name, the the corresponding buffer will be augmented instead of overwritten with the text. The undo buffer always contains the latest change. Buffers <1-9> contain the latest 9 *line* deletions (`"1` is most recent).

```
 * | x                  | Delete <*> chars under and after the cursor.
 * | X                  | <*> chars before the cursor.
 * | d<move>            | From begin to endpoint of <*><move>.
 * | dd                 | <*> lines.
```



```
 - | D                  | The rest of the line.
 * | <<move>            | Shift the lines described by <*><move> one
                        |   shiftwidth to the left.
 * | <<                 | Shift <*> lines one shiftwidth to the left.
 * | .                  | Repeat latest command <*> times.
```

3.5.7 Changing text

End changing text with $<esc>$

```
 * | r<char>            | Replace <*> chars by <char> - no <esc>.
 * | R                  | Overwrite the rest of the line,
                        |   appending change <*> - 1 times.
 * | s                  | Substitute <*> chars.
 * | S                  | <*> lines.
 * | c<move>            | Change from begin to endpoint of <*><move>.
 * | cc                 | <*> lines.
 * | C                  | The rest of the line and <*> - 1 next lines.
 * | =<move>            | If the option `lisp' is set, this command
                        |   will realign the lines described by <*><move>
                        |   as though they had been typed with the option
                        |   `ai' set too.
 - | ~                  | Switch lower and upper cases
                        |   (should be an operator, like `c').
 * | J                  | Join <*> lines (default 2).
 * | .                  | Repeat latest command <*> times (`J' only once).
 - | &                  | Repeat latest `ex' substitute command, e.g.
                        |   `:s/wrong/good'.
 - | :[x,y]s/<p>/<r>/<f>| Substitute (on lines x through y) the pattern <p>
                        |   (default the last pattern) with <r>.  Useful
                        |   flags <f> are `g' for `global' (i.e. change
                        |   every non-overlapping occurrence of <p>) and
                        |   `c' for `confirm' (type `y' to confirm a
                        |   particular substitution, else <cr>).  Instead
                        |   of `/' any punctuation CHAR unequal to <lf>
                        |   can be used as delimiter.
```

3.5.8 Substitute replacement patterns

The basic meta-characters for the replacement pattern are "`&`" and "`~`"; these are given as `\&` and `\~` when nomagic is set. Each instance of "`&`" is replaced by the characters which the regular expression matched. The meta-character "`~`" stands, in the replacement pattern, for the defining text of the previous replacement pattern. Other meta-sequences possible in the replacement pattern are always introduced by the escaping character "`\`". The sequence `\n` (where n is an integer between 1 and 9) is replaced by the text matched by the n^{th} regular subexpression enclosed between `\(` and `\)`. The sequences `\u` and `\l` cause the immediately following character in the replacement to be converted to uppercase or lowercase respectively if this character is a letter. The sequences `\U` and `\L` turn such conversion on, either until `\E` or `\e` is encountered, or until the end of the replacement pattern.

3.5.9 Remembering text (yanking)

With yank commands you can put "<a--zA--Z> before the command, just as with delete commands. Otherwise you only copy to the undo buffer.

```
   * | y<move>          | Yank from begin to endpoint of <*><move>.
   * | yy               | <*> lines.
   * | Y                | Idem (should be equivalent to `y$' though).
   - | m<a-z>           | Mark the cursor position with a letter.
```

3.5.10 Commands while in insert or change mode

```
^@                      | If typed as the first character of the
                        |   insertion, it is replaced with the previous
                        |   text inserted (max. 128 chars), after which
                        |   the insertion is terminated.
^V                      | Deprive the next char of its special meaning
                        |   (e.g. <esc>).
^D                      | One shiftwidth to the left.
0^D                     | Remove all indentation on the current line
                        |   (there must be no other chars on the line).
^^D                     | Idem, but it is restored on the next line.
^T                      | One shiftwidth to the right
^H | <erase>            | One char back.
^W                      | One word back.
<kill>                  | Back to the begin of the change on the
                        |   current line.
<intr>                  | Like <esc> (but you get a beep as well).
```

3.5.11 Display commands (See also Move commands)

```
^G                      | Give file name, status, current line number
                        | and relative position.
^L                      | Refresh the screen (sometimes `^P' or `^R').
^R                      | Sometimes vi replaces a deleted line by a `@',
                        |   to be deleted by `^R' (see option `redraw').
[*]^E                   | Expose <*> more lines at bottom, cursor stays
                        | put (if possible).
[*]^Y                   | Expose <*> more lines at top, cursor stays put
                        | (if possible).
[*]^D                   | Scroll <*> lines downward
                        |   (default the number of the previous scroll;
                        |   initialization: half a page).
[*]^U                   | Scroll <*> lines upward
                        |   (default the number of the previous scroll;
                        |   initialization: half a page).
[*]^F                   | <*> pages forward.
[*]^B                   | <*> pages backward (in older versions `^B'
                        | only works without count).
z-                      | Move current line to bottom of the screen.
z.                      | Move current line to the center of the screen.
/string/z-              | Move line with string in it to the bottom of
                        | the screen.
```


If in the next commands the field <wi> is present, the windowsize will change to <wi>. The window will always be displayed at the bottom of the screen.

```
[*]z[wi]<cr>            | Put line <*> at the top of the window
                        |   (default the current line).
[*]z[wi]+               | Put line <*> at the top of the window
                        |   (default the first line of the next page).
[*]z[wi]-               | Put line <*> at the bottom of the window
                        |   (default the current line).
[*]z[wi]^               | Put line <*> at the bottom of the window
                        |   (default the last line of the previous page).
[*]z[wi].               | Put line <*> in the center of the window
                        |   (default the current line).
```

3.5.12 Writing, editing other files, and quitting vi

In ":" "ex" commands "%" denotes the current file, "#" is a synonym for the alternate file (which normally is the previous file). Marks can be used for line numbers too: '<a-z>. In the *:w*, *:f*, *:cd*, *:e*, and *:n* commands, shell meta-characters can be used.

```
:q                      | Quit vi, unless the buffer has been changed.
:q!                     | Quit vi without writing.
^Z                      | Suspend vi.
:w                      | Write the file.
:w <name>               | Write to the file <name>.
:w >> <name>            | Append the buffer to the file <name>.
:w! <name>              | Overwrite the file <name>.
:x,y w <name>           | Write lines x through y to the file <name>.
:wq                     | Write the file and quit vi; some versions quit
                        |   even if the write was unsuccessful!
                        |   Use `ZZ' instead.
ZZ                      | Write if the buffer has been changed, and
                        |   quit vi.  If you have invoked vi with the `-r'
                        |   option, you'd better write the file
                        |   explicitly (`w' or `w!'), or quit the
                        |   editor explicitly (`q!') if you don't want
                        |   to overwrite the file - some versions of vi
                        |   don't handle the `recover' option very well.
:x [<file>]             | Idem [but write to <file>].
:x! [<file>]            | `:w![<file>]' and `:q'.
:pre                    | Preserve the file - the buffer is saved as if
                        |   the system had just crashed; for emergencies,
                        |   when a `:w' command has failed and you don't
                        |   know how to save your work (see `vi -r').
:f <name>               | Set the current filename to <name>.
:cd [<dir>]             | Set the working directory to <dir>
                        |   (default home directory).
:cd! [<dir>]            | Idem, but don't save changes.
:e [+<cmd>] <file>      | Edit another file without quitting vi - the
                        |   buffers are not changed (except the undo
                        |   buffer), so text can be copied from one file to
```

```
                         |   another this way.  [Execute the 'ex' command
                         |   <cmd> (default '$') when the new file has been
                         |   read into the buffer.]  <cmd> must contain no
                         |   <sp> or <ht>.  See 'vi startup'.
:e! [+<cmd>] <file>      | Idem, without writing the current buffer.
^^                       | Edit the alternate (normally the previous) file.
:rew                     | Rewind the argument list, edit the first file.
:rew!                    | Idem, without writing the current buffer.
:n [+<cmd>] [<files>]    | Edit next file or specify a new argument list.
:n! [+<cmd>] [<files>]   | Idem, without writing the current buffer.
:args                    | Give the argument list, with the current file
                         |   between '[' and ']'.
```

3.5.13 Macros

When mapping take a look at the options *to* and *remap* (below).

```
:map <string> <seq>      | <string> is interpreted as <seq>, e.g.
                         |   ':map ^C :!cc %^V<cr>' to invoke 'cc' (the C
                         |   compiler) from within the editor
                         |   (vi replaces '%' with the current file name).
:map                     | Show all mappings.
:unmap <string>          | Deprive <string> of its mapping.  When vi
                         |   complains about non-mapped macros (whereas no
                         |   typos have been made), first do something like
                         |   ':map <string> Z', followed by
                         |   ':unmap <string>' ('Z' must not be a macro
                         |   itself), or switch to 'ex' mode first with 'Q'.

:map! <string> <seq>     | Mapping in append mode, e.g.
                         |   ':map! \be begin^V<cr>end;^V<esc>O<ht>'.
                         |   When in append mode <string> is preceded by
                         |   '^V', no mapping is done.
:map!                    | Show all append mode mappings.
:unmap! <string>         | Deprive <string> of its mapping (see ':unmap').
:ab <string> <seq>       | Whenever in append mode <string> is preceded and
                         |   followed by a breakpoint (e.g. <sp> or ','), it
                         |   is interpreted as <seq>, e.g.
                         |   ':ab ^P procedure'.  A '^V' immediately
                         |   following <string> inhibits expansion.
:ab                      | Show all abbreviations.
:unab <string>           | Do not consider <string> an abbreviation
                         |   anymore (see ':unmap').
@<a-z>                   | Consider the contents of the named register a
                         |   command, e.g.:
                         |       o0^D:s/wrong/good/<esc>"zdd
                         |   Explanation:
                         |       o               - open a new line
                         |       0^D             - remove indentation
                         |       :s/wrong/good/  - this input text is an
                         |                         'ex' substitute command
                         |       <esc>           - finish the input
                         |       "zdd            - delete the line just
                         |                         created into register 'z'
```



```
                         |   Now you can type '@z' to replace 'wrong'
                         |   with 'good' on the current line.
@@                       | Repeat last register command.
```

3.5.14 Switch and shell commands

```
Q | ^\ | <intr><intr>    | Switch from vi to 'ex'.
:                        | An 'ex' command can be given.
:vi                      | Switch from 'ex' to vi.
:sh                      | Execute a subshell, back to vi by '^D'.
:[x,y]!<cmd>             | Execute a shell <cmd> [on lines x through y;
                         |   these lines will serve as input for <cmd> and
                         |   will be replaced by its standard output].
:[x,y]!! [<args>]        | Repeat last shell command [and append <args>].
:[x,y]!<cmd> ! [<args>]  | Use the previous command (the second '!') in a
                         |   new command.
[*]!<move><cmd>          | The shell executes <cmd>, with as standard
                         |   input the lines described by <*><move>,
                         |   next the standard output replaces those lines
                         |   (think of 'cb', 'sort', 'nroff', etc.).
[*]!<move>!<args>        | Append <args> to the last <cmd> and execute it,
                         |   using the lines described by the current
                         |   <*><move>.
[*]!!<cmd>               | Give <*> lines as standard input to the
                         |   shell <cmd>, next let the standard output
                         |   replace those lines.
[*]!!! [<args>]          | Use the previous <cmd> [and append <args> to it].
:x,y w !<cmd>            | Let lines x to y be standard input for <cmd>
                         |   (notice the <sp> between the 'w' and the '!').
:r!<cmd>                 | Put the output of <cmd> onto a new line.
:r <name>                | Read the file <name> into the buffer.
```

3.5.15 Vi startup

As discussed earlier *vi* is started by simply typing `vi filename` where the filename is optional. It is possible to include a list of filenames instead of just the one. This tells *vi* to edit the first file. After you are finished with the file it edits the second and continues this process until all the files in the list have been edited.

The editor can be initialized by the shell variable *EXINIT*, which looks like:

```
EXINIT='<cmd>|<cmd>|...'
<cmd>: set options
       map ...
       ab ...
export EXINIT (in the Bourne shell)
```

However, a better way is to put the list of initializations into a file. If this file is located in your home directory, and is named *.exrc* and the variable *EXINIT* is not set, the list will be executed automatically at startup time. However, *vi* will always execute the contents of a *.exrc* in the current directory, if you own the file. Otherwise you have to type

```
:so file
```

to *source* the *file* yourself.

In a *.exrc* file a comment is introduced with a double quote character: the rest of the line is ignored.[24]

On-line initializations can be given with **vi +<cmd> file**, e.g.:

```
vi +x file              | The cursor will immediately jump to line x
                        |   (default last line).
vi +/<string> file      | Jump to the first occurrence of <string>.
```

You can start at a particular tag with:

```
vi -t <tag>             | Start in the right file in the right place.
```

Sometimes, e.g. if the system crashed while you were editing, it is possible to recover files lost in the editor by typing `vi -r file`. Typing `vi -r` shows the files you can recover. The *readonly* flag allows you to view a file with *vi* without the danger of accidentally saving changes. However, if you do make changes that you decide you want to save, typing `:w!` will override the *readonly* option.

3.5.16 The most important options

```
ai                      | autoindent - In append mode after a <cr> the
                        |   cursor will move directly below the first
                        |   CHAR on the previous line.  However, if the
                        |   option `lisp' is set, the cursor will align
                        |   at the first argument to the last open list.
aw                      | autowrite - Write at every shell escape
                        |   (useful when compiling from within vi).
dir=<string>            | directory - The directory for vi to make
                        |   temporary files (default `/tmp').
eb                      | errorbells - Beeps when you goof
                        |   (not on every terminal).
ic                      | ignorecase - No distinction between upper and
                        |   lower cases when searching.
lisp                    | Redefine the following commands:
                        |   `(', `)'   - move backward (forward) over
                        |                S-expressions
                        |   `{', `}'   - idem, but don't stop at atoms
                        |   `[[', `]]' - go to previous (next) line
                        |                beginning with a `('
                        |   See option `ai'.
list                    | <lf> is shown as `$', <ht> as `^I'.
magic                   | If this option is set (default), the chars `.',
                        |   `[' and `*' have special meanings within search
                        |   and `ex' substitute commands.  To deprive such
                        |   a char of its special function it must be
```

[24]An exception to this is if the last command on the line is a `map[!]` or `ab` command or a shell escape, a trailing comment is not recognized, but considered part of the command.


```
                        |   preceded by a `\'.  If the option is turned off
                        |   it's just the other way around.  Meta-chars:
                        |   ^<string>    - <string> must begin the line
                        |   <string>$    - <string> must end the line
                        |   .            - matches any char
                        |   [a-z]        - matches any char in the range
                        |   [^a-z]       - any char not in the range
                        |   [<string>]   - matches any char in <string>
                        |   [^<string>]  - any char not in <string>
                        |   <char>*      - 0 or more <char>s
                        |   \<<string>   - <string> must begin a word
                        |   <string>\>   - <string> must end a word
modeline                | When you read an existing file into the buffer,
                        |   and this option is set, the first and last 5
                        |   lines are checked for editing commands in the
                        |   following form:
                        |
                        |     <sp>vi:set options|map ...|ab ...|!...:
                        |
                        |   Instead of <sp> a <ht> can be used, instead of
                        |   `vi' there can be `ex'.  Warning: this option
                        |   could have nasty results if you edit a file
                        |   containing `strange' modelines.
nu                      | number - Numbers before the lines.
para=<string>           | paragraphs - Every pair of chars in <string> is
                        |   considered a paragraph delimiter nroff macro
                        |   (for `{' and `}').  A <sp> preceded by a `\'
                        |   indicates the previous char is a single letter
                        |   macro.  `:set para=P\ bp' introduces `.P' and
                        |   `.bp' as paragraph delimiters.  Empty lines and
                        |   section boundaries are paragraph boundaries
                        |   too.
redraw                  | The screen remains up to date.
remap                   | If on (default), macros are repeatedly
                        |   expanded until they are unchanged.
                        |   Example: if `o' is mapped to `A', and `A'
                        |   is mapped to `I', then `o' will map to `I'
                        |   if `remap' is set, else it will map to `A'.
report=<*>              | Vi reports whenever e.g. a delete
                        |   or yank command affects <*> or more lines.
ro                      | readonly - The file is not to be changed.
                        |   However, `:w!' will override this option.
sect=<string>           | sections - Gives the section delimiters (for `[['
                        |   and `]]'); see option `para'. A `{' beginning a
                        |   line also starts a section (as in C functions).
sh=<string>             | shell - The program to be used for shell escapes
                        |   (default `$SHELL' (default `/bin/sh')).
sw=<*>                  | shiftwidth - Gives the shiftwidth (default 8
                        |   positions).
sm                      | showmatch - Whenever you append a `)', vi shows
                        |   its match if it's on the same page; also with
                        |   `{' and `}'.  If there's no match at all, vi
                        |   will beep.
taglength=<*>           | The number of significant characters in tags
                        |   (0 = unlimited).
```

```
tags=<string>           | The space-separated list of tags files.
terse                   | Short error messages.
to                      | timeout - If this option is set, append mode
                        |   mappings will be interpreted only if they're
                        |   typed fast enough.
ts=<*>                  | tabstop - The length of a <ht>; warning: this is
                        |   only IN the editor, outside of it <ht>s have
                        |   their normal length (default 8 positions).
wa                      | writeany - No checks when writing (dangerous).
warn                    | Warn you when you try to quit without writing.
wi=<*>                  | window - The default number of lines vi shows.
wm=<*>                  | wrapmargin - In append mode vi automatically
                        |   puts a <lf> whenever there is a <sp> or <ht>
                        |   within <wm> columns from the right margin
                        |   (0 = don't put a <lf> in the file, yet put it
                        |   on the screen).
ws                      | wrapscan - When searching, the end is
                        |   considered 'stuck' to the begin of the file.
:set <option>           | Turn <option> on.
:set no<option>         | Turn <option> off.
:set <option>=<value>   | Set <option> to <value>.
:set                    | Show all non-default options and their values.
:set <option>?          | Show <option>'s value.
:set all                | Show all options and their values.
```

3.6 Miscellaneous tips

Although you should be able to come up with all of the following commands by studying the tables of *vi* commands, I have included a few examples that I have found useful.

3.6.1 Line deletions

```
:g/string/d
```

deletes every line that contains *string*, while

```
:v/string/d
```

deletes every line that does *not* contain *string*.

3.6.2 Switching cases

In the replacement part of a substitution command, i.e. between the second "/" and third "/",

```
\u means make the following character upper case
\l means make the following character lower case
\U means make the rest of the replacement upper case
\L means make the rest of the replacement lower case
```

How about a few examples?

1. Make the first letter of every word from line 18 to 43 uppercase.

   ```
   :18,43s/\<./\u&/g
   ```

2. Change "uPPeR" and "LoweR" in any mixture of cases to lowercase.

   ```
   :s/[UuLl][PpOo][PpWw][Ee][Rr]/\L&/
   ```

3. Make the whole file uppercase.

   ```
   :%s/.*/\U&/
   ```

4. Make the region from line *m* to line *n* all uppercase.

   ```
   :'m,'ns/.*/\U&/
   ```

5. Make a paragraph all lowercase.

   ```
   :?^$?,/^$/s/.*/\L&/
   ```

6. Make the first letter of every word in a paragraph uppercase.

   ```
   :?^$?,/^$/s/\([^ ][^ ]*\)/\u&/g
   ```

7. Make the second word of each line uppercase.

   ```
   :1,$s/^\([^ ]*\) \([^ ]*\) \(.*\)/\1 \U\2\e \3/
   ```

3.6.3 Spell checking in vi

To check the spelling of your document without exiting out of *vi*, type

```
:!spell % > %.sp
:e %.sp
:e#                     (To get back to your document)
```

3.6.4 Additional search and replace

To change Gravity isn't just a "good idea." It's the "law." to Gravity isn't just a ‘‘good idea.’’ It's the ‘‘law.’’ type

```
:g/$/s/"\([^"]*\)"/‘‘\1’’/g
```

The following will find words that begin and end with a vowel:

```
/\<[aeiouAEIOU][a-zA-Z']*[aeiouAEIOU]\>
```

3.6.5 Removing blank lines

Blank lines can be removed from a file with any of the following:

```
:v/./d
```

or

```
:g/^$/d
```

or

```
:%!/usr/ucb/cat -s
```

or

```
:%!sed /./,/^$/!d
```

Be aware that you may need to get rid of trailing whitespace first. This can be accomplished with

```
:%s/[ ^I]*$/!d
```

3.6.6 Writing from buffers

To save the contents of the *a* buffer to *filename*, type

```
:e filename<RETURN>"ap (to edit a new file and put 'a's contents in it)
:w                     (to save it)
```

To save a portion of a file to another file you could type

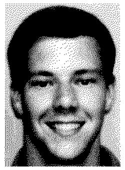
```
ma                    (mark text at the top of the region to be saved)
mb                    (mark text at the bottom of the region to be saved)
:'a,'b w filename
```

3.7 Further reading

A number of other *vi* documents can be found via anonymous ftp in: /pub/vi at cs.uwp.edu

4 About the Author

Chris Taylor's accomplishments speak for themselves but often have poor grammar. Here is a grammatically corrected version of what they say. Chris Taylor is a graduate student in electrical engineering at Purdue University where he is currently enjoying his twenty-fourth trip around the sun. He has consumed four tons of cow's milk and counted to 125,000 by ones all in the last decade. Chris was not homecoming king in high school or college, he did not graduate at the top of his high school class (although the same cannot be said of college). He was not involved with any political movements in the 1960's, and has never been in a rock band. Chris hasn't been convicted of grand theft auto in the last five years and has never been convicted of a violent crime. He has moved eighteen different times living in four countries and three states but contends that he is not running from anything or anyone. He has ridden more elephants than horses and hopes to one day swim across the Arctic Ocean.



Printed on recyclable paper. To reuse: Bleach in warm water, then let dry for three days.