With the release of OS X, the Macintosh has a command line for the first time since it was introduced in 1984. Admittedly, Apple has done a great job with OS X, and the vast majority of users can go a long time without having to ever touch the terminal. In fact, many of the longtime Mac faithful don't care to learn it at all, but there are growing numbers of people who would like to know how to use it. Mac OS X Command Line 101 is going to try and help those people do just that.

While Apple has done a tremendous job with OS X, no operating system is perfect. Sometimes a problem cannot be fixed using one of the standard GUI tools, or sometimes one of those GUI tools doesn’t do quite what you want. Sometimes, there are even times when using the command line is just easier.

Luckily, OS X is based on Unix. Unix was not market-driven; it was written by geeks for geeks in the real ivory tower of the past, Bell Labs. To understand what this means, consider what Larry Wall, creator of the Perl programming language, calls the three great virtues of programmers: laziness, impatience, and hubris. Laziness means you don't want to spend a lot of time and effort doing something simple, or doing it repeatedly. Impatience means you want to have the necessary tools for a particular task at your fingertips. Hubris means you want to know that the tool will just work without having to worry about it. Of course, getting to that point requires a lot of hard work, time, and testing. Fortunately, all of those attributes have gone into Unix already.

Granted, the Unix command line can occasionally look as if an inebriated cat walked over the keyboard; for first-timers, this can be a bit daunting. However, we longtime Mac loyalists are used to dealing with daunts; otherwise, we wouldn't have stuck with Apple through thick and thin, would we? While the Unix command line can have a steep learning curve, at first, part of that is because you can do so much with it.

The command line takes a toolbox approach. A tool does one job and does it well. Just as you don't use a saw to pound a nail into the wall, you should chose the right command line tool for the right job.

Unix uses shells as a command interpreter; you type the commands, the system runs them, and the shell acts as the go-between. The first shell was the Bourne Shell, created by Stephen Bourne in 1979. Later, the C shell (csh) was developed at Berkeley; it was so called because its syntax is similar to the C programming language. To encourage laziness, impatience, and hubris, an enhanced version, tcsh, was released. This is the shell that OS X uses when you start Terminal.app.

So what sets the Unix command-line apart?

• The basic philosophy is 'Simple things should be simple, complex things should be less simple, and practically anything should be possible.'

• The Unix toolbox. While any command-line will give you tools, the commands available in Unix are fantastic in quantity and quality. If you want to do something on the command line, someone else probably has wanted to do it as well and written a command for it. What's more, Unix allows you to string commands together so you can build complex tools out of simple ones. You'll particularly appreciate them if you've ever had to use DOS or worked on a mainframe.

• Documentation. The online help files, called "man pages," are very good; and many companies provide books to help as well.

• Flexible commands. Most commands have options that will allow you to modify their behavior if the default is not exactly what you need. And even if that isn't quite what you had in mind, you can combine them to suit your purpose.

• I/O Redirection. If you want to write the output to a file, or read parameters from a file, or string commands together so that you don't have to write to/read from files, Unix shells provide a very simple way to do this.

• Multiple processes. It may seem to be hard to believe now, but back when Unix was created, this was a novel concept. We may take it for granted, but one of the beauties of Unix is allowing you to have many processes running at once. What's more you can have many on the same command line, which puts a lot of power at your fingertips.

• Job control. This lets you send one task to run in the background while you do other work; you can also pull a background job to the foreground. You can start and stop jobs. And you can remove them from the system, nicely or with extreme prejudice.
• Protected memory. If one process goes rogue, it won't interfere with the memory space of another process. Anyone who has been forced to use ... other desktops ... will appreciate this.

• 'Quotes', "quotes", 'quotes'. Anyone who is given to quotation will tell you, from long experience, that you must quote in the correct context. Unix gives you three contexts, each one being appropriate in different situations.

• Regular expressions. While not unique to Unix, the OS has a rich set which give you great flexibility in searching and manipulating text. They are largely, though not entirely, consistent between commands.

• A file is a file is a file. Some systems, which shall go nameless, pack the bits differently if the file is binary or if it is ASCII. Unix makes no such distinction; even a directory is a file. The beauty of this is that you can, say, run a Perl script to make quick-and-dirty edits on a Word Document. (NOTE: That is not for the faint of heart, but the system won't stop you from trying.)

Admittedly, all of this is a lot to take in at first, but you have to walk before you run, and crawl before you walk. I will start with the basics and move slowly. If you have any questions, be sure to send them to me at richard@macobserver.com. If I like it well enough, I may use it in a future column.

And most of all, remember to have fun.

Learning How To Move Around
Part II of this series...
May 3rd, 2002

"I know God is an Irishman, because His book begins at the beginning."
- Dave Allen

We'll start with some basic commands in our first column. The three most important things you can know about anything are: where things are, where you are, and where you can find more information. That's as good a place to start as any.

>From the desktop, go into Applications -> Utilities, and open Terminal.app. This should give you a command line prompt:

Welcome to Darwin!
[localhost:~] dr_unix%

This provides you with some handy information. "localhost" is the computer name. ("localhost" means the computer in front of you.) After the colon is where you are, known as your current working directory. (The "~" is a shortcut that means 'your home directory'.) And the rest of the prompt before the "~" is your username.

So we know where we are, our home directory, but where, exactly, is that? Unix provides a simple way to do this, pwd. pwd will print the current working directory (i.e., where you are) to the command line. (Mnemonic: 'pwd' is an acronym for 'print working directory'.)

[localhost:~] dr_unix% pwd
/Users/dr_unix
[localhost:~] dr_unix%

When you first fire up Terminal.app, it will place you in what is called your home directory. This is your private area; it is yours to do with what you want (within reason, of course).

Now that you know where you are, you'll want to see what is around the place. The command ls will show you a list of all the things in your present working directory. (Mnemonic: 'ls' is short for 'list'.)

[localhost:~] dr_unix% ls
Desktop  Documents  Library  Movies  Music  Pictures
Public   Sites
[localhost:~] dr_unix%
As a way to compare, go to Aqua and open up your home folder. There will be a one-to-one correspondence between the files and folders that Aqua shows, and the files listed by `ls`. (At least there darn well should be.)

Now that we’ve taken a quick look at where we are, we’d like to be able to move around a bit. To change directories, use the command `cd`. (Mnemonic: ‘cd’ is an acronym for ‘change directory.’) The command takes the form `cd name_of_directory`. In this case, we are going to `cd` to the Library directory, which you’ll note above is in the directory where we are.

```
[localhost:~] dr_unix% cd Library
[localhost:~/Library] dr_unix% ls
Application Support Documentation       Keyboards
Screen Savers
Assistants            Favorites          Keychains
Sounds
Audio                 FontCollections     Logs
Voices
Caches
iTunes                Fonts              Preferences
ColorPickers          Internet Plug-Ins    Printers
[localhost:~/Library] dr_unix%
```

The `cd` command has a couple nice tricks. One is that, if you make a mistake, you can type `cd -` to go back to the previous directory.

```
[localhost:~] dr_unix% pwd
/Users/dr_unix
[localhost:~] dr_unix% cd Library
[localhost:~/Library] dr_unix% pwd
/Users/dr_unix/Library
[localhost:~/Library] dr_unix% cd -
[localhost:~] dr_unix% pwd
/Users/dr_unix
[localhost:~] dr_unix%
```

Also, if you do not provide `cd` with a directory, it will assume you want to go to your home directory.

```
[localhost:~/Library] dr_unix% pwd
/Users/dr_unix/Library
[localhost:~/Library] dr_unix% cd
[localhost:~] dr_unix% pwd
/Users/dr_unix
[localhost:~] dr_unix%
```

Thus, it is always easy to find your way home.

Also, notice that Unix is clever with directories. If you are going to a directory under your present working directory, you don’t need to type the entire directory. If you are in your home directory, `cd Library` and `cd /Users/dr_unix/Library` will do the same thing (assuming your username is “dr_unix”, of course).

Now that you can move around without losing your way home, let’s explore a little. Many, though not all, of Unix’ commands reside in a special area called “/usr/bin.” Let’s go there now and take a look around.

```
[localhost:~] dr_unix% cd /usr/bin
[localhost:/usr/bin] ls
[MANY MANY FILES LISTED]
[localhost:/usr/bin]
```

Now that’s a lot of stuff; this is where one of those daunts might appear. However, you know that each one of those things does something useful, otherwise it wouldn’t be there. (Remember, Unix was created by computer geeks, and good computer geeks hate creating more work for themselves.) Also, you know that each of those commands does one thing, because this is Unix. So how do you find out what does what? You could pester your local Unix geek, but we aren’t always the most approachable people. We don’t like being bothered; that’s why Unix has good

Command Line 101
online documentation. It's accessed using the `man` command. (Mnemonic: 'man' is short for 'manual'.) To find out about a command, just type: `man command_name` and you will get far more information than you ever wanted. (On a side note, `man` uses a command called `more` to display only one page at a time. Hit the space bar to advance to the next page, 'b' to go back a page, and 'q' to quit.)

```bash
[localhost:/usr/bin] dr_unix% man ls
man: Formatting manual page...
LS(1) System Reference Manual
LS(1)
NAME
ls - list directory contents
SYNOPSIS
ls [-ACFLRSTWacdfgiklnoqrstux] [file ...]
DESCRIPTION
For each operand that names a file of a type other than
directory, `ls` displays its name as well as any requested, associated
information. For
each operand that names a file of type directory, `ls` displays the
names
of files contained within that directory, as well as any
requested, associated information.

If no operands are given, the contents of the current directory
are dis-
played. If more than one operand is given, non-directory
operands are
displayed first; directory and non-directory operands are sorted
sepa-
rately and in lexicographical order.

The following options are available:

(I hit 'q' here because there were nine or ten more screens of information. I told you it was probably more than you wanted to know.)

`man` is good for finding out what a specific command does, but what if you want to do something but don't know what the command is? Yes, Unix gives you that, too, with `apropos`. Did you notice that the first line of information we got from `man ls` was a brief synopsis?

`ls - list directory contents`

`apropos` will look through these synopses in the man pages and provide you a list of commands (and their synopses) so you can look for the one that best suits your needs.

```bash
[localhost:/usr/bin] dr_unix% apropos list
ciphers(1) - SSL cipher display and cipher list
tool
column(1) - columnate lists
ls(1) - list directory contents
ls/of(8) - list open files
perl/todo(1) - Perl s-1TO-DOs0 List
sudoers(5) - list of which users may execute what
users(1) - list current users
```

Command Line 101
Page 4
xargs(1)                 - construct argument list(s) and execute utility
[localhost:/usr/bin] dr_unix%

And if that still isn't what you want, you can pester your Unix geek

Understanding Metacharacters

Part III of this series...
May 10th, 2002

"Well, I never metacharacter I didn't like."
- Ron Wurtz (a friend from grad school, you don't know him)

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in Hardcore X! forum.

[Editor's Note: This installment was originally supposed to have been published as Part IV of this series, due to an error on TMO's part. It should be read after Part IV, The Missing Argument Chapter, which has been published early for those working along with the series.]

In the previous column, we looked at arguments and options, using the cd and ls commands to show commands requiring and responding to different numbers and types of arguments. While the ability to manipulate the commands with different arguments is handy, it only hinted at the power at your fingertips.

The Unix shells have a set of special characters known as 'metacharacters'. We have run into these before. In our first discussion, involving the tcsh prompt and the cd command, it was noted that the tilde character, '~', is a shorthand for 'my home directory'. What is really means is 'the home directory of the user following me'; however, if there is no user, it defaults to your home directory.

[localhost:-] dr_unix% pwd
/Users/dr_unix
[localhost:-] dr_unix% cd ~
[localhost:-] dr_unix% pwd
/Users/dr_unix
[localhost:-] dr_unix% cd ~root
[localhost:-~root] dr_unix% pwd
/private/var/root
[localhost:-~root] dr_unix% cd ~fred
Unknown user: fred.
[localhost:-~root] dr_unix% cd ~
[localhost:-~root] dr_unix% pwd
/Users/dr_unix
[localhost:-~root] dr_unix%

Here, when we type cd ~, the '~' is expanded to our current home directory, /Users/dr_unix, before the command is passed to the system. The shell takes the cd ~root command, determines that the home directory for root is '/private/var/root', then passes the command cd /private/var/root to the system. Likewise, for the command cd ~fred, the shell cannot find a user 'fred' on the system, so it prints the error message (Unknown user: fred.) and does not change directory.

 '~' is part of a special class of characters called 'filename metacharacters'. The shell interprets them differently than a regular old character (like, say, a 'b' or an 'N'). As we've seen, in the case of '~', the shell will expand these characters behind the scenes, in a way that depends on which character is to be expanded. The filename metacharacters are:
*: Used to match any number of any characters
?: Used to match exactly one of any character
~: Your home directory
~name: Home directory of the user "name"
[abc]: Used to match any one of the characters in the square brackets (in this case, either an 'a', a 'b', or a 'c'). A dash (-) can be used to denote a range, like [A-M].
{abc,def,ghi}: Expand each comma-separated string inside the braces.
Let's use ls to show how this works. Because '*' can denote any number of any characters, the command ls * is the same as ls if there are no directories to be listed.
If you want to suppress the directory expansion with `ls` so that they are in fact the same, there is the `-d` flag:

```
[localhost:~] dr_unix% ls -d *
Adam.txt Documents Movies Pictures Sites temp.html who_list
Desktop Library Music Public personal test_1.txt
[localhost:~] dr_unix% ls -d D*
Desktop Documents
[localhost:~] dr_unix% ls *.txt
Adam.txt test_1.txt
[localhost:~] dr_unix% ls *st*
test_1.txt who_list
[localhost:~] dr_unix% ls -d D* *.txt *st*
Adam.txt Desktop Documents test_1.txt who_list
```

If you want a list of all the files starting with 'D', or ending with '.txt', or containing 'st' anywhere, just type:

```
[localhost:~] dr_unix% ls -d D* *
Desktop Documents
[localhost:~] dr_unix% ls *.txt
Adam.txt test_1.txt
[localhost:~] dr_unix% ls *st*
test_1.txt who_list
[localhost:~] dr_unix% ls -d D* *.txt *st*
Adam.txt Desktop Documents test_1.txt who_list
```

Note that metacharacters are expanded into a list and then shoved into the command. Notice that, in the example provided here, there are two directories that start with 'D'. If you pass the argument 'D*' to `cd`,

```
[localhost:~] dr_unix% cd D*
D*: Ambiguous.
```

you get an error message (D*: Ambiguous.). What happens is that, behind the scenes, the shell looks for all files that can be described with 'D*', expands that to 'Desktop Documents', and this is passed to the `cd` command, which becomes `cd Desktop Documents`. Since `cd` takes (at most) one argument, the `cd` command burps. Neat, huh?
"*" is, by far, the most commonly used filename metacharacter, but the others can be quite handy, when needed. (After all, that's why they exist; they are more common in shell scripts, but can be useful on the command line as well.) The '?' metacharacter is used less often, but when it is needed, it's indispensable. It will substitute for any one character, and '?'s can be strung in series to emulate 'any character, a specific number of times':

```
[localhost:~] dr_unix% ls t*
  temp.html test_1.txt
[localhost:~] dr_unix% ls te??.*
  temp.html
[localhost:~] dr_unix%
```

The '[]' metacharacters are mostly used in shell scripting, and rarely then, but again, they're available in Unix for a reason. The shell will match any character that is between the brackets.

```
[localhost:~] dr_unix% ls
Adam.txt Documents Movies Pictures Sites   temp.html who_list
Desktop  Library  Music  Public  personal  test_1.txt
[localhost:~] dr_unix% ls te*
  temp.html test_1.txt
[localhost:~] dr_unix% ls te[mono]*
  temp.html
[localhost:~] dr_unix%
```

If you want to look for a range of characters, you can use the '-' character:

```
[localhost:~] dr_unix% ls *.txt
Adam.txt  test_1.txt
[localhost:~] dr_unix% ls [a-z]*.txt
  test_1.txt
[localhost:~] dr_unix%
```

(Remember, Unix is case-sensitive, so the 'A' character is not the same as the 'a' character.)

In ten years of Unix, I have used the '()' metacharacters maybe twice, but again, when I needed them, they saved me a lot of work. The shell takes each comma-separated string in the '{}', expands them into a list, then does whatever else is required.

```
[localhost:~] dr_unix% ls
Adam.txt Documents Movies Pictures Sites   temp.html who_list
Desktop  Library  Music  Public  personal  test_1.txt
[localhost:~] dr_unix% ls {D,te}*
  temp.html test_1.txt
[localhost:~] dr_unix% ls -d {D,te}*
Desktop  Documents temp.html test_1.txt
[localhost:~] dr_unix%
```

The '{}' is rarely used; in fact, I'd forgotten about it until researching this column. But again, when it is needed, it will save you a lot of work.

Two other metacharacters (which aren't strictly filename metacharacters, but are close enough for government work) are '. and '..'. (I know that I've mentioned them before, but they are worth repeating.) These are present in every directory.

```
[localhost:~] dr_unix% ls -ad .*
  ..  .DS_Store           .Trash              .cshrc..
  .  .FBCIndex           .addressbook   .ssh
  .CFUserTextEncoding   .FBCLockFolder   .addressbook.lu  .tcsh_history
[localhost:~] dr_unix%
```

'.' is a shorthand for many things. In the context of directories, '..' is a shorthand for 'this directory'.

```
[localhost:~] dr_unix% ls
Adam.txt Library Pictures personal  test_1.txt
Desktop Movies Public  sigquotes what_list
Documents Music Sites temp.html who_list
[localhost:~] dr_unix% ls .
Adam.txt Library Pictures personal  test_1.txt
Desktop Movies Public  sigquotes what_list
Documents Music Sites temp.html who_list
[localhost:~] dr_unix%
```

Every directory has a '.' directory referring to itself. This may not seem too useful now, but it will be quite handy later on. Using '..', on the other hand, is so common that you will wonder how you ever got on without it. It means 'my parent directory'.

Command Line 101    Page 7
Every directory has a '..' referring to its parent directory. (The one exception is the root directory, '/', in which case it refers to itself.)

Now that we have looked at the filename metacharacters, options, and arguments, and have seen how a couple of the basic commands are used in action, we are ready for some meatier topics. We'll start with manipulating files and directories in the next column.

The Missing Argument Chapter

*Part IV of this series...*  
*May 13th,*

"This isn't an argument."
"Yes it is."
"No it isn't. It's just contradiction."
"No it isn't."
"Yes it is!"
- Michael Palin and John Cleese, Monty Python's Flying Circus, "The Argument Clinic"

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the [earlier columns](#), write back in the [comments](#) below, or join us in [Hardcore X! forum](#).

[Editor’s Note: This installment was originally planned to be run as Part III of this series, and should be read before the piece actually published as Part III, Understanding Metacharacters. As such, we are publishing it early this week for those working along with the series, numbered as Part IV.] Now that we've taken a look at some basic commands, let's examine two key, related features of Unix commands: arguments and options.

We saw command arguments in the previous column. Remember the `cd` command?

```bash
[localhost:~] dr_unix% pwd
/Users/dr_unix
[localhost:~] dr_unix% cd Sites
[localhost:~/Sites] dr_unix% pwd
/Users/dr_unix/Sites
[localhost:~/Sites] dr_unix% cd
[localhost:~] dr_unix% pwd
/Users/dr_unix
```

When we typed the command `cd Sites` to change our current working directory, we told the computer that 'Sites' is the target directory. 'Sites' is the argument to the `cd` command. Recall that, if `cd` is given no argument, it defaults to the user's home directory. Also, most, if not all, Unix commands will check to see if an argument is valid before running the intended operation. (And if one doesn't, it should.)

```bash
[localhost:~] dr_unix% cd Library
Library: No such file or directory.
[localhost:~] dr_unix%
```

Some commands are very persnickety about how many arguments they take, and will insist on the proper number. Others are more devil-may-care, and will take however many you wish. However, the behavior may vary by the number of arguments.

```
[localhost:~] dr_unix% cd Music Pictures
cd: Too many arguments.
[localhost:~] dr_unix%
```

Check the `man` page for a particular command for the details.
Take `ls` as an example of a command that can take any number of arguments. Recall that if you give it no arguments `ls` will print a list of all the files in your current working directory.

```bash
[localhost:~] dr_unix% pwd
/Users/dr_unix
[localhost:~] dr_unix% ls
Adam.txt  Documents  Movies  Pictures  Sites  test_1.txt
Desktop   Library   Music   Public   personal  who_list
```

If, however, you pass `ls` an argument, the behavior changes. If the argument is a file, you get the name of the file.

```bash
[localhost:~] dr_unix% ls who_list
who_list
[localhost:~] dr_unix%
```

That isn't too useful. However, if the argument is a directory, you get a list of files in that directory.

```bash
[localhost:~] dr_unix% ls Sites
images
index.html
[localhost:~] dr_unix% ls Sites/images
apache_pb.gif  macosxlogo.gif  web_share.gif
[localhost:~] dr_unix%
```

While that is useful, it is also a bit limited. Fortunately, `ls` is more powerful than we've seen so far. For one thing, it can take any number of arguments.

```bash
[localhost:~] dr_unix% ls test_1.txt who_list
test_1.txt who_list
[localhost:~] dr_unix%
```

Also, `ls` is clever enough that you can combine regular files with directories.

```bash
[localhost:~] dr_unix% ls test_1.txt Library Movies
test_1.txt
Movies:
Sites:
images
index.html
[localhost:~] dr_unix%
```

Notice how the directories are separated from the regular files, and how the directories are separated from each other. Also, notice that the Movies directory is listed, but as it contains no files, nothing is listed under it.

While lists of files are handy to know, you often want to know things like: how big are these files, when were they last changed, etc. `ls` provides this if you use different options.

Options are small flags that you pass to a command to slightly alter or refine its behavior. The options are denoted by starting with a `-` and come before arguments to the command start. The behavior of the options, and which ones are valid, varies by command. If you are unsure about what is available for a particular command, remember that you can always check the `man` page.

For `ls`, many options are available, but I'll use a few of the more common ones as examples. One of the most useful is the `-l` option, which gives a LOT of information.

```bash
[localhost:~] dr_unix% ls -l
total 24
-rw-r--r--  1 dr_unix staff 15 Mar 25 17:28 Adam.txt
drwx------  7 dr_unix staff 194 Mar 25 17:25 Desktop
drwx------  18 dr_unix staff 568 Mar 25 09:48 Documents
drwx------  26 dr_unix staff 840 Mar 17 12:42 Library
drwx------  2 dr_unix staff 264 Mar 21 20:30 Movies
drwx------  2 dr_unix staff 264 Mar 21 20:30 Music
drwx------  6 dr_unix staff 264 Mar 21 20:30 Pictures
drwxr-xr-x  4 dr_unix staff 264 Mar 22 18:06 Public
drwxr-xr-x  4 dr_unix staff 264 Aug 21 2001 Sites
drwxr-xr-x  7 dr_unix staff 264 Mar 23 11:21 personal
```
So, just what does all this mean? The first character tells us whether the file is a directory (d) or a regular file (-). (There are other possibilities, but don’t worry about that for now.) The next nine characters show the permissions on each file. For each file, there are three levels of permission: read (r), write (w), and execute (x). The first three show permission for the file’s owner, the second three for anyone else in the same user group, and the final three for the rest of the world. (If you aren’t familiar with user groups, don’t sweat it.) Next comes the number of links (again, don’t sweat it), followed by the file’s owner, the file’s user group, the size of the file, the date it was last altered, and finally the name of the file.

Normally, `ls` on the Mac sorts in ASCII order, which is alphabetical by capital letters followed by alphabetical by lowercase. If you want to list in reverse order, you can use the `-r` option.

```
[localhost:-] dr_unix% ls
Adam.txt Documents Movies Pictures Sites test_1.txt
Desktop Library Music Public personal who_list
[localhost:-] dr_unix% ls -r

who_list personal Public Music Library Desktop
test_1.txt Sites Pictures Movies Documents Adam.txt
[localhost:-] dr_unix%
```

Or if you want to sort from newest to oldest, use the `-t` option.

```
[localhost:-] dr_unix% ls -t
Adam.txt Documents Public Pictures who_list Library
Desktop personal Music Movies test_1.txt Sites
[localhost:-] dr_unix%
```

What if you want to sort from oldest to newest? You can just combine the `-t` and `-r` options. This can be entered as `ls -t -r`. For convenience (i.e. laziness), you can combine flags, like `ls -tr`.

```
[localhost:-] dr_unix% ls -tr
Sites test_1.txt Movies Music personal Desktop
Library who_list Pictures Public Documents Adam.txt
[localhost:-] dr_unix%
```

Or, going hog-wild,

```
[localhost:-] dr_unix% ls -lrt
total 24
drwxr-xr-x 4 dr_unix staff 264 Aug 21 2001 Sites
drwx------ 26 dr_unix staff 840 Mar 17 12:42 Library
drwxr--r-- 1 dr_unix staff 15 Mar 18 10:47 test_1.txt
drwxr--r-- 1 dr Unix staff 243 Mar 18 10:47 who_list
drwx------ 2 dr Unix staff 264 Mar 21 20:30 Movies
drwx------ 6 dr_unix staff 264 Mar 21 20:30 Pictures
drwx------ 2 dr_unix staff 264 Mar 21 20:30 Music
drwxr-xr-x 4 dr Unix staff 264 Mar 22 18:06 Public
drwxr-xr-x 7 dr Unix staff 264 Mar 23 11:21 personal
drwx------ 18 dr Unix staff 568 Mar 25 09:48 Documents
drwx------ 3 dr Unix staff 194 Mar 25 17:25 Desktop
drwx------ 1 dr Unix staff 15 Mar 25 17:28 Adam.txt
[localhost:-] dr_unix%
```

One other very handy option for `ls` is `-a`. This shows all files:

```
[localhost:-] dr Unix% ls -a
.
.. .Trash           Library          personal
.. .ssh            Movies           test_1.txt
.CFUserTextEncoding .tcsh_history Music
.DS_Store          Adam.txt        who_list
.FBCIndex          Desktop         Public
.FBClockFolder     Documents       Sites
[localhost:-] dr Unix%
```

The files that start with a `.` are known as ‘hidden files’. Normally they are files that hold your personal settings for particular applications (.mailrc for the Unix `mail` application, for example). Normally, you want to ignore them, so `ls` defaults to not printing them. And of course, `-a` can be combined with other options, and yes Virginia, hidden files can also be directories:
Two of these files that deserve our attention for now are the '.' and '..' directories. '.' means 'this directory'.

This may seem silly, but later we will see how handy this is. '..' represents 'the directory above this one':

So we can see, even from these limited examples, that arguments and options give Unix commands a lot of power and flexibility, yet they do it in a very compact, some would say elegant, way.

The real power of `ls`, and a key feature of the Unix command line, lies in the wildcard feature, which we'll see in the next column.

---

**Working With Files From The Command Line**

*Part V of this series...*

*May 24th, 2002*

"File under 'Wheat'."

- Comment printed on REM's *Green* album

*This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.*
Now that we have seen how to use commands with arguments and wildcards, let's see how we can use them to do some useful things. When you are on the command line, eventually you will have to know how to manipulate files and directories: how to move files, delete them, copy them, get information on them, etc.

Before we start, let's review the shell's approach to files and directories. When Terminal.app fires up a new shell, we can see where we are by `pwd`:

```
Welcome to Darwin!
[localhost:-] dr_unix% pwd
/Users/dr_unix
[localhost:-] dr_unix%
```

'/Users/dr_unix' is the full pathname of our current working directory. '/Users' is the parent directory of 'dr_unix'. A '/' separates directories and files in pathnames. Also, recall that a lone '.', in the context of a file or directory name, is a shortcut for 'my current directory'.

```
[localhost:-] dr_unix% ls .
Adam.txt Documents Movies Pictures Sites sigquotes test_1.txt
Desktop Library Music Public personal temp.html who_list
[localhost:-] dr_unix% cd ./Desktop
[localhost:-~/Desktop]
[localhost:-] dr_unix% pwd
/Users/dr_unix/Desktop
[localhost:-~/Desktop]
[localhost:-] dr_unix%
```

Also, recall that '..' is a shortcut for 'my parent directory'.

```
[localhost:-~/Desktop]
[localhost:-] dr_unix% pwd
/Users/dr_unix/Desktop
[localhost:-] dr_unix% cd ..
[localhost:-~]
[localhost:-] dr_unix% pwd
/Users/dr_unix
[localhost:-] dr_unix%
```

We've already seen and used three common commands for manipulating files and directories: `cd` for moving through the directory structure, `pwd` for finding out where we are, and `ls` for listing files in a directory and information on them.

Let's continue with the simplest command, `cp`. (Mnemonic: short for 'copy.') `cp` will copy a file or directory from one place to another. It simply takes the form `cp sourcefile destinationfile`:

```
[localhost:-] dr_unix% ls .
Adam.txt Documents Movies Pictures Sites sigquotes test_1.txt
Desktop Library Music Public personal temp.html who_list
[localhost:-] dr_unix% cp who_list what_list
[localhost:-] dr_unix% ls .
Adam.txt Library Pictures personal test_1.txt
Desktop Music Sites temp.html who_list
[localhost:-] dr_unix% ls -l wh*
-rw-r--r-- 1 dr_unix staff 243 Apr 3 09:51 what_list
-rw-r--r-- 1 dr_unix staff 243 Mar 18 10:47 who_list
[localhost:-] dr_unix%
```

We can see that the file was created at the proper time and has the same size. `cp` has some useful options, so check the man page (`man cp`) if your need is not quite what you had in mind--

Oh, you didn't want to copy that file? No problem. To get rid of a file, simply use the `rm` command. (Mnemonic: short for 'remove') Like most Unix shell commands, it is straightforward: `rm file1 file2 file3 ...`

```
[localhost:-] dr_unix% ls w*
what_list who_list
[localhost:-] dr_unix% rm what_list
[localhost:-] dr_unix% ls w*
who_list
[localhost:-] dr_unix%
```

If you want to delete a directory and all its contents, you can use the `-r` option. This is not recommended; you can do yourself a mischief, as you can well imagine. If you do want to remove an empty directory, but don't want to accidentally remove any files that may be there but forgotten, use the `rmdir` command instead. (Mnemonic: short for 'remove directory').
And if you want to make a directory, there is, of course, a command for this, _mkdir_. (Mnemonic: short for, you guessed it, 'make directory'.)

If you want to rename a file, you could, I suppose, do a _cp_ to the new name and then _rm_ the original file. However, that would be silly, wasteful, and Not The Unix Way. (That is, you shouldn’t have to type two commands when you can make do with one.) _mv_ is the command for renaming/moving a file. (Mnemonic: short for ‘mv’. See a pattern forming?)

"But how do I view the contents of a file?" I hear you ask. "I will tell you," I hear myself reply. There are two commands (well, three) that let you do this. The first is _cat_, which concatenates and prints the files to the screen, _cat file1 file2 ..._. This is fine when you are looking at a small file, or want to have a series of files viewed as one. If there are more lines in the file than lines on the screen, and there often are, _more_ is the proper choice. _more file1 file2 file3 ..._ prints one screenful of a file at a time.

<table>
<thead>
<tr>
<th>localhost:~</th>
<th>dr_unix% more test_1.txt</th>
</tr>
</thead>
<tbody>
<tr>
<td>This is a test.</td>
<td></td>
</tr>
<tr>
<td>localhost:~</td>
<td>dr_unix%</td>
</tr>
</tbody>
</table>

You can type _man more_ to see all the intricacies, but you can go a long way with ‘q’ to quit, ‘b’ to move back a screen, and ‘ ’ to advance one screen’s worth.

_less_ is similar to _more_. (All "less is more" jokes may be sent elsewhere. I’ve heard them before.) _less_ allows backward movement in the file in addition to forward movement. Also, _less_ does not have to read the entire file before it starts its output, so on really large files, you don’t have to wait as long. _less_‘s options are slightly different from _more_‘s, so consult your local _man_ page for more information.

If you are interested in only seeing the start or end of a file, _head_ and _tail_ will do the trick. The commands are in the form:

```
head -n file
tail -n file
```

where ‘n’ is the number of lines at the top/bottom that you wish to see. (If no number is given, both default to ten lines.)
tail does have one handy option that is worth mentioning. Let’s say you have a process running which is generating a log file, ‘logfile.txt’. (A web server comes to mind.) You want to see how the process is progressing and watch it live. tail -f logfile.txt will not stop when the end of the file is reached; rather, it will wait until additional data is appended to the file, then print it to the screen. And when you want it to stop, just hit ‘control-c’. Neat, huh? (There is also a -F option that is similar but more powerful; I’ve never needed its power, though.)

In this example, head and tail were not too useful. After all, the file was only five lines long. If you want to know this beforehand, wc (word count) does this. wc requires one of three flags, wc -l counts the number of lines in a file, wc -w the number of words, and wc -c the number of characters. If the command is given a list of files, it tells the size for each file and gives a grand total.

Now that we’ve seen how to manipulate files, we are nearly ready for some heavy-duty Unix wizardry. First, we must look at the shell’s different forms of quotes, which will be the next column.

Special Characters & The Command Line

Part VI of this series...

May 31st, 2002

"Quotes, unquotes, and quotes."
"That's three quotes?"
"Yes."
"Add another quote and make it a gallon."
-Zeppo and Groucho Marx, Animal Crackers

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

Depending on how you look at it, Unix has either three or four forms of quotes (as if you couldn’t guess from the header), each useful in its own way. Quoting allows you to suppress the special meaning of certain characters, or to enforce other special characters, or even embed commands within commands.

So far, we have seen that the Unix shell attaches certain meanings to particular characters, such as the tilde (~). This puts great power at your fingertips; however, it can also make things a bit sticky in certain circumstances. Suppose you do the following from your command line:

```
[localhost:-] dr_unix% ls -d Documents/A* Documents/Acrobat User Data Documents/AppleWorks User Data
[localhost:-] dr_unix% ls -d Documents/Acrobat User Data ls: Data: No such file or directory
```
ls: Documents/Acrobat: No such file or directory
ls: User: No such file or directory
[localhost:~] dr_unix%
What gives? Well, remember that the shell uses the spaces as the delimiter between elements in a list. Or, translating from Geek to English, when you give ls a list of files to list, it assumes that the spaces separate the file names. Thus, it looks for three files: Documents/Acrobat, User, and Data. The space is a special character to the command line. The best way to get around this is to enclose the filename in quotes:

```
[localhost:~] dr_unix% ls -d "Documents/Acrobat User Data"
Documents/Acrobat User Data
[localhost:~] dr_unix%
```

This tells the shell "ignore the special meanings of the characters inside the quotes". In this case, it is equivalent to saying "a space is not a filename separator, but part of a filename". The reason you don't need to do this when you do something like ls * is that, behind the scenes, Unix and tcsh are clever enough to know that things like spaces in filenames can cause issues, so they are dealt with quietly. Still, you can see why Unix people prefer to use the ' ' to separate words in a file or directory name; it avoids a lot of trouble. Apple has already been bitten by this. Remember the updated version of iTunes and how it could, under certain circumstances, delete a disk if the name of the disk had a space in it? Apple forgot the quotes.

In general, it is best to avoid special characters where they might cause confusion. However, we live in the world as read, and sometimes you just have to deal with other people's decisions. In addition, we will eventually want to create our own tools, and it only makes sense to be aware of them anyway. So just what are these special characters that can cause trouble to us command-line mavens?

<table>
<thead>
<tr>
<th>Characters</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>* ? [] ^ { } ~</td>
<td>Filename pattern matching and expansion</td>
</tr>
<tr>
<td>$</td>
<td>Used to denote variables</td>
</tr>
<tr>
<td>! ^</td>
<td>History reference and substitution</td>
</tr>
<tr>
<td>&amp;</td>
<td>Used to send commands to the background</td>
</tr>
<tr>
<td></td>
<td>Pipe</td>
</tr>
<tr>
<td>&gt; &lt;</td>
<td>Redirects output/input</td>
</tr>
<tr>
<td>;</td>
<td>Command separator</td>
</tr>
<tr>
<td>space</td>
<td>Argument separator</td>
</tr>
<tr>
<td>tab</td>
<td>Filename completion (in tcsh)</td>
</tr>
<tr>
<td>esc</td>
<td>Filename completion (in csh, may cause problems in tcsh)</td>
</tr>
<tr>
<td>()</td>
<td>Subshell execution</td>
</tr>
<tr>
<td>&quot;</td>
<td>Quote used for command substitution</td>
</tr>
<tr>
<td>\ &quot;</td>
<td>Quote characters</td>
</tr>
</tbody>
</table>

Some of these will be familiar, others will induce a condition known as MEGO (My Eyes Glaze Over). For now, don't worry about them except to note that they can cause problems if you aren't aware of them. We'll concentrate on the four characters used in quoting: \ " , and `.
The \ is an escape character; it turns off the special meaning of the character following it. For example,

earlier we used the double quote to turn off the special meaning of spaces in a filename.

```
[localhost:~] dr_unix% ls -d Documents/A*
Documents/Acrobat User Data Documents/AppleWorks User
Data [localhost:~] dr_unix% ls -d "Documents/Acrobat User Data"
Documents/Acrobat User Data
```

However, Unix being Unix, we could also have done this:

```
[localhost:~] dr_unix% ls -d "Documents/Acrobat User Data"
Documents/Acrobat User Data
```

What happens is that the \ are telling the shell "the character after me has a special meaning; ignore that
meaning and use it literally". Therefore, the shell interprets the command argument Documents/Acrobat User Data as one file, not three.

The two "normal" forms of quoting are the single quote (') and the double quote("). Why two forms? Well, for one thing, if what you have to say involves a single quote or double quote, one can turn of the special meaning of the other. Consider the echo command, which simply spits out whatever list you give it:

```
[localhost:~] dr_unix% echo This is a test.
This is a test. [localhost:~] dr_unix%
```

Now, what if you want echo to print something with an apostrophe in it?

```
[localhost:~] dr_unix% echo You can't say that
Unmatched '.
```

Without the double quotes, echo interprets the ' as a quote character and looks for another one to mark the end of the quote. However, there is not an ending quote, so the shell spits back an error. Enclosing the list in double quotes turns off the special meaning of the single quote.

And of course, you can do the reverse.

```
[localhost:~] dr_unix% echo 'You can"t say that'
You can"t say that [localhost:~] dr_unix%
```

In fact, one can even get silly with them, thus:

```
[localhost:~] dr_unix% echo ""'
''
```

What's more, if you want to use both the " and ', you can use the backslash to escape both of them:

```
[localhost:~] dr_unix% echo Gee isn't that "swell".
Gee isn't that "swell".
```

That is a lot simpler and less error-prone than:

```
[localhost:~] dr_unix% echo Gee isn't that "swell".
Gee isn't that "swell".
```

It should be noted that, in spite of the examples I've shown so far, single and double quotes have slightly different behavior. Obviously, they interpret each other as regular old characters, as we've seen above. In general, single quotes are more strict. They suppress the special meanings of all characters except for the ! to denote history substitution. (We'll get to history and history substitution in a later column; for now, don't worry about it.) The double quotes ignore the special meaning of everything between them except for the ! for history substitution, the $ to indicate a variable (again, more on that later), and the backticks ('command') for command substitution.

[Before moving to command substitution, I should note that some shells, notably the Bourne shell and Korn shell, handle single and double quotes a bit differently than csh and tcsh do. So if you are not using the default tcsh in OS X, you may not be able to duplicate this behavior with exactly the same commands.]

[You have been warned.]

The last form of quoting, using backticks ('), is probably the most powerful. What backticks do is take the command enclosed in them, run the command, and place the output of that command into the command line. An example will best illustrate this.
Remember our old friend, the `wc` command, which returns a count of the characters, words, or lines in a file, depending on the command line option? It is used thus:

```
[localhost:~] dr_unix% wc -l Adam.txt test_1.txt who_list
  1 Adam.txt
  1 test_1.txt
  5 who_list
  7 total
[localhost:~] dr_unix%
```

Now, what if you wanted the information for only certain files? Well, you could, of course, type out a whole list of files. However, that isn’t fun. What you’d like to do is have the system determine which files you want, and then run the `wc` command on those files, and do it all on the fly. Wouldn’t that be neat? Wouldn’t that appeal to your laziness? Naturally, in Unix, there is a simple way.

Let’s say there are some number of files ending with “*.doc” in your ~/Documents directory. At any time, the names and numbers of these files can change. And you want to know 1) what files are there, and 2) how many words are in each. You know you can use `wc -w` to get a count of the number of words for a list of files. You also know that you can use `ls ~/Documents/*:.doc` to get a list of those files. So if you can combine them, you don’t need to write your own tool from scratch.

```
You could change directories (`cd ~/Documents`) and try `wc -w *:.doc` but that’s requiring more steps than you want. Besides, you want to be able to run this command from anywhere, and you don’t want to change your directory. How to do it? Use the other form of quotes, backticks.

[localhost:~] dr_unix% wc -w `ls ~/Documents/*.doc`
  650 /Users/dr_unix/Documents/6197.doc
  1443 /Users/dr_unix/Documents/ISIHAC_LOG.doc
  2093 total
[localhost:~] dr_unix%
```

Here’s how command substitution works. The shell parses the command you send,

```
wc -w `ls ~/Documents/*.doc`
```

and sees the ‘`’ s. It takes the text in between the two ‘`’ s,

```
ls ~/Documents/*:.doc
```

and runs that as a command. The `ls` command returns two files:

```
/Users/dr_unix/Documents/6197.doc
/Users/dr_unix/Documents/ISIHAC_LOG.doc
```

This output is then substituted for the `ls` command in the command line:

```
```

This command is run, and the results are printed to the screen.

```
650 /Users/dr_unix/Documents/6197.doc
1443 /Users/dr_unix/Documents/ISIHAC_LOG.doc
2093 total
[localhost:~] dr_unix%
```

Admittedly, this example is pretty contrived. You could just as easily type in `wc -c ~/Documents/*:.doc` and get the same result. After all, the `ls` command is just building a list of files, and the “*” wildcard is doing the same thing. However, I chose this example because 1) I wanted to show how command substitution works, 2) this example is fairly straightforward, 3) it is a reminder that, in Unix, there is more than one way to skin a cat, 4) it is a powerful tool that, as we build up our command-line vocabulary, will become more and more important, and 5) it hints at some of the power behind the simple things we’ve done so far.

Backticks are just one example of how a Unix shell will allow you to combine commands. We will examine more of these after we discuss a related topic, redirecting input and output.

---

Input & Output From The Command Line

Part VII of this series...

June 7th, 2002
"Tiny differences in input could quickly become overwhelming differences in output."
- James Gleick, "Chaos"

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

So far, we have looked at examples where data is read from the keyboard and the results are printed to the screen. This is nice (in fact, this is vital), but it is not always appropriate. This column will examine how Unix I/O (input/output) can be redirected, a fancy way of saying "how to read and write data to and from files".

["And there was much rejoicing." Enough people have pestered me about this that I've moved this column up on the list. Actually, there is a lot more to I/O redirection than this, but, in the words of Johnny Cash, "One piece at a time."]

Many things happen behind the scenes when you open a window in Terminal.app. Among them, three file descriptors are automagically opened. A file descriptor is a low-level Unix I/O concept that is really only interesting to systems programmers. (What can I say, we're easily entertained.) The three are:

- 0, which represents Standard Input or stdin. This is the keyboard.
- 1, which represents Standard Output or stdout. This is the terminal window.
- 2, which represents Standard Error or stderr. This is also the terminal window.

Now, you may wonder why there are two different file descriptors that both go to the terminal window. The reason for the two descriptors is that there are times when you want to keep the Standard Output separated from the Standard Error. After all, if you are running an application, you want the errors to go into a report to the developers, but they should be separated from the results to go to your boss. (Bosses don't like error messages; reality frightens them.) So having the standard output and the standard error distinct makes a lot of sense. At the same time, the "natural" place for error messages is the same as the "natural" place for output, the terminal window. So the default puts them together, but they can be separated quite easily.

Redirecting Standard Output is quite simple and by far the most common; it uses the > symbol to send the output to a file, thus:

`command > outfile`

This is best shown by example. The `who` command prints information on all the users who are currently logged into the system. (Two handy options are `-H`, which prints header information, and `-u`, which includes the idle time for each user.) At the command line, from your home directory, type:

```
[localhost:~] dr_unix% who -Hu
USER     LINE     WHEN         IDLE     FROM
dr_unix  console  May 10 16:09 04:16
```

Now, you may wonder why there are two different file descriptors that both go to the terminal window. The reason for the two descriptors is that there are times when you want to keep the Standard Output separated from the Standard Error. After all, if you are running an application, you want the errors to go into a report to the developers, but they should be separated from the results to go to your boss. (Bosse's don't like error messages; reality frightens them.) So having the standard output and the standard error distinct makes a lot of sense. At the same time, the "natural" place for error messages is the same as the "natural" place for output, the terminal window. So the default puts them together, but they can be separated quite easily.

Redirecting Standard Output is quite simple and by far the most common; it uses the > symbol to send the output to a file, thus:

`command > outfile`

This is best shown by example. The `who` command prints information on all the users who are currently logged into the system. (Two handy options are `-H`, which prints header information, and `-u`, which includes the idle time for each user.) At the command line, from your home directory, type:

```
[localhost:~] dr_unix% who -Hu
USER     LINE     WHEN         IDLE     FROM
dr_unix  console  May 10 16:09 04:16
```

Yes, it really is that simple. The only pitfalls might be that 1) you could try to write the output to a nonexistent directory or one to which you are not permitted to write; or 2) you have the shell variable `noclobber` set so you don't overwrite a file that already exists. We'll cover permissions and shell variables like `noclobber` later. For now, we don't have to worry about them; `noclobber` can be turned off by typing `set noclobber=no`, and you can always write to your home directory; just use the tilde metacharacter (~): `command > ~/outfile`

But what if you want to append the data to a file, instead of overwriting it? Not a problem; use >> instead:

`command >> outfile`
As an aside, if you want to force output to a file even with noclobber set, use >! instead of >:

\texttt{command >! out\_file or command >>! out\_file}

But I digress.

By the same token, if you want to read input from a file, the < symbol can be used. This is less common and a little trickier; while you are learning the shell, you may be tempted to overuse input redirection. Partly that's because most commands will take a filename that you put on the command line. However, it does have its uses.

As an example, consider the \texttt{tr} command. \texttt{tr string1 string2} will copy Standard Input to Standard Output after translating the characters in \texttt{string1} to \texttt{string2}. (In other words, all occurrences of the first character in \texttt{string1} are changed to the first in \texttt{string2}, the second in \texttt{string1} to the second in \texttt{string2}, and so on. You can also use a '-' to denote a range of characters.) Let's say you want to make the file \texttt{who\_list} all upper case so some addle-pated MCSE, if you'll pardon the redundancy, won't have a hissy-fit. You just need to redirect \texttt{who\_list} to Standard Input, thus:

\begin{verbatim}
[localhost:~] dr_unix% cat who_list
USER     LINE     WHEN         IDLE     FROM
dr_unix  console  May 10 16:09 04:16
dr_unix  ttyt1   May 10 19:53  .
dr_unix  ttyt2   May 10 20:20  .
[localhost:~] dr_unix% tr a-z A-Z < who_list
USER     LINE     WHEN         IDLE     FROM
DR_UNIX  CONSOLE  MAY 10 16:09 04:16
DR_UNIX  TTYT1   MAY 10 19:53  .
DR_UNIX  TTYT2   MAY 10 20:20  .
[localhost:~] dr_unix%
\end{verbatim}

And if you need to save the results to a file, you can combine the two redirection symbols thus:

\begin{verbatim}
[localhost:~] dr_unix% tr a-z A-Z < who_list > Cap_who_list
[localhost:~] dr_unix% cat Cap_who_list
USER     LINE     WHEN         IDLE     FROM
DR_UNIX  CONSOLE  MAY 10 16:09 04:16
DR_UNIX  TTYT1   MAY 10 19:53  .
DR_UNIX  TTYT2   MAY 10 20:20  .
[localhost:~] dr_unix%
\end{verbatim}

In days of old, the < was commonly used with the \texttt{mail} command. You could write a message to a file, rather than type and edit it all at one sitting, and then send the message:

\texttt{mail user1 user2 user3 user4 user5 < message\_file}

These days, that is rare because of all the GUI mail utilities, but if you want to fire off a message or a log file from within a script, or you want to gain three geek points for having sent e-mail from the command line at least once, that is how you do it. And if I catch any of you using that to send SPAM, I will take you out back and shoot you. Slowly and painfully.

As you might guess, there is a corresponding << redirector. It is known as a "Here" document, and is used:

\texttt{command << endtext}

This means that \texttt{command} will read from Standard Input up to (but not including) a line identical to \texttt{endtext}:

\begin{verbatim}
[localhost:~] dr_unix% cat << END
Fred was here.
-> Now he is gone.
-> That would be okay.
-> Except I think he stole my corkscrew.
-> END
Fred was here.
Now he is gone.
That would be okay.
Except I think he stole my corkscrew.
[localhost:~] dr_unix%
\end{verbatim}
Generally, this is only used in shell scripting or in some übergeek commands like `cat`, `echo`, `ex`, or `sed`. Still, there are times when it is handy to know. And kiddies, here is a warning from a grumpy old programmer: watch out for those trailing spaces. If I had put an extra space after `cat << END` so that it was really `cat << END`, the `cat` command would have said “Aha, ‘END’ is not the same as ‘END’; so there must be more input coming’. That can be a tough bug to find.

`, `, and `<<` will take you a long way, but there are other forms of redirection, less commonly used but needed from time to time. We have seen how to redirect Standard Output to a file; any error message still comes to the screen.

```
[localhost:~] dr_unix% ls who_list xiphoid
ls: xiphoid: No such file or directory
who_list
[localhost:~] dr_unix% ls who_list xiphoid > temp
ls: xiphoid: No such file or directory
[localhost:~] dr_unix% cat temp
who_list
[localhost:~] dr_unix% cat
```

How do we make Standard Output and Standard Error go to the same file? With `&`, thus:

```
command >& out_file
```

[or `command >&! out_file` if you want to overwrite `out_file` even if `noclobber` is set.]

```
[localhost:~] dr_unix% ls who_list xiphoid
ls: xiphoid: No such file or directory
who_list
[localhost:~] dr_unix% ls who_list xiphoid >& temp
[localhost:~] dr_unix% cat temp
who_list
[localhost:~] dr_unix% cat
```

And if you want to append to a file, it's simply:

```
command >>>& out_file
```

[or, you guessed it, `command >>>&! out_file` if `noclobber` is set.]

And finally, if you want to send Standard Output to one file and Standard Error to another, this takes the form:

```
(command > out_file) >& error_file
```

```
[localhost:~] dr_unix% (ls who_list xiphoid > temp_out) >& temp_err
[localhost:~] dr_unix% cat temp_out
who_list
[localhost:~] dr_unix% cat temp_err
ls: xiphoid: No such file or directory
[localhost:~] dr_unix% cat
```

Finally, this seems as good a place as any to bring up a wonderful little Unix feature, `/dev/null`. `/dev/null` is a truly handy piece of software that acts as a black hole for whatever you send to it. So if you type:

```
(ls who_list xiphoid > temp_out) >& /dev/null
```

a wonderful thing will happen: the error messages will go away. Oh, the errors will still be there, but you don't see the error messages.

```
--
```

As an aside, I personally find that the Bourne and Korn shells redirectors to be a bit more flexible. For example, using the above command, `ls who_list xiphoid`, let's say that I want to redirect Standard Error to a file but keep Standard Output coming to the screen, note the steps I follow:

```
[localhost:~] dr_unix% w
10:01PM  up  5:53, 3 users, load averages: 0.16, 0.18, 0.00
```

Command Line 101  Page 20
If anyone can tell me a better way to do this in tcsh, without involving /dev/whatever files or other shells, let me know. I'll pass it on and be sure you get credit.

... 

If you look in your home directory now, you will find a lot of files that you need to clean up. Saving output to a file, only to use its contents as the input of another command, creates a mess; worse, it goes against laziness and impatience for you to have to remove these files. Even worse, you might remove one that you need. Fortunately, Unix allows you to redirect output from one command to another without saving to a temporary file; we'll look at that in the next column.

---

**Of Pipes & More**

*Part VIII of this series...*

*June 14th, 2002*

"Tiny differences in input could quickly become overwhelming differences in output."

- James Gleick, "Chaos"

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

In the last column, we looked at how input and output can be redirected to and from files. However, what if you want to run several commands on your data? Well, you could do something like this:

```bash
[localhost:~] dr_unix% command1 sourcefile > tempfile1
[localhost:~] dr_unix% command2 tempfile1 > tempfile2
[localhost:~] dr_unix% rm tempfile1
[localhost:~] dr_unix% command3 tempfile2 > tempfile3
[localhost:~] dr_unix% rm tempfile2
[localhost:~] dr_unix% command4 tempfile3 > out_file
[localhost:~] dr_unix% rm tempfile3
```
That is a lot of typing; it discourages laziness and impatience. Worse, you might erase the wrong file along the way. And even worse, it is slow. Not only do you have to spend a lot of time typing and waiting, but constant I/O to a drive is generally the worst drag on speed.

If you think about it, you really don't need those tempfiles. Oh, a crusty old mainframer might hit you with a slide rule if you say this out loud, but think about it: all you need is a mechanism that will take the output from command1 and make it the standard input for command2, and so on.

In Unix shells, this is accomplished with a pipe, "!". We could have performed the previous task all in one line.

```
[localhost:~] dr_unix% command1 sourcefile | command2 | command3 | command4 > out_file
```

Recall the example from the previous column where we typed:

```
[localhost:~] dr_unix% who -Hu > who_list
[localhost:~] dr_unix% cat who_list
 USER     LINE     WHEN         IDLE     ...  CONSOLE  MAY 10 16:09 04:16  
 dr_unix  console May 10 16:09 04:16
 dr_unix  tty1    May 10 19:53   .
 dr_unix  tty2    May 10 20:20   .
[localhost:~] dr_unix% tr a-z A-Z < who_list > Cap_who_list
[localhost:~] dr_unix% cat Cap_who_list
 USER     LINE     WHEN         ...  TTYP1    MAY 11 13:30   .
 DR_UNIX  CONSOLE MAY 10 16:09 04:16
 DR_UNIX TTY1    MAY 10 19:53   .
 DR_UNIX TTY2    MAY 10 20:20   .
[localhost:~] dr_unix%
```

That can now be replaced with:

```
[localhost:~] dr_unix% who -Hu | tr a-z A-Z > Cap_who_list
[localhost:~] cat Cap_who_list
 USER     LINE     WHEN         ...  TTYP1    MAY 11 13:30   .
 DR_UNIX  CONSOLE MAY 10 16:09 22:24
 DR_UNIX TTY1    MAY 11 13:30   .
 DR_UNIX TTY2    MAY 11 13:43   .
 DR_UNIX TTY3    MAY 10 22:34 15:36
[localhost:~] dr_unix%
```

What this does is that, when `tcsh` is parsing the command line behind the scenes, it sees the "!" character and says "What ho! A pipe. The Unix genius who wrote this wants me to take the output from the command on the left, `who -Hu`, and make it the input for the command on the right, `tr a-z A-Z`. " So it does.

The beauty of this is that you can take very simple tools, string them together in whatever order you wish, and create complex tools. This gives you great power. Given the many options that Unix commands can have, and the number of Unix commands, this also gives you great flexibility. And what's more, it makes things very simple for you. Anyone who has ever written code to do this in a language like C can tell you how great the shell
is at this. (Actually, once they start they'll never shut up; so don't ask, just take my word for it.)

One of the more common uses of the pipe on the command line is to take a large amount of output and present it only one screen at a time using the more command (or page or less). Recall from an earlier column that we took a quick look at the /usr/bin directory where a lot of commands reside. First, using a pipe to connect commands, we can determine how many files exist in the directory:

```
[localhost:/usr/bin] dr_unix% ls -l | wc -l
457
[localhost:/usr/bin] dr_unix%
```

That's a lot of files, far more than is standard for a Terminal.app console. So, what if we want to see all of the files, but not have them fly by at breakneck speed? Why, pipe the output through more (or page or less).

```
[localhost:/usr/bin] dr_unix% ls | more
CFInfoPlistConverter
a2p
addftinfo
addr
axml
afmtodit
appleping
appletviewer
apply
apropos
ar
arch
as
asa
at
at_cho_prn
atlookup
atos
atprint
atq
atrm
atstatus
autoconf
[localhost:/usr/bin] dr_unix%
```

(I hit 'q' at this point to quit out of more.)

The pipe is only one way to put more than one command on one command line, because there are different ways that you might want to combine them. For example, we've already seen the use of backticks, ``, to expand the results of a command into another command. To show the difference between the pipe and the backtick, let's look at how we can combine the ls and wc commands.

Consider my home directory:

```
[localhost:~] dr_unix% ls
Adam.txt  Documents  Movies  Pictures  Sites
```
temp.html  who_list
Desktop    Library    Music    Public    personal
test_1.txt
[localhost:~] dr_unix%

[Consult local listings for home directories in your area.] Using backticks will take the results of ls above and use them as the arguments for wc, thus:

[localhost:~] dr_unix% wc -l `ls`
  1 Adam.txt
wc: Desktop: Is a directory
    0 Desktop
wc: Documents: Is a directory
    0 Documents
wc: Library: Is a directory
    0 Library
wc: Movies: Is a directory
    0 Movies
wc: Music: Is a directory
    0 Music
wc: Pictures: Is a directory
    0 Pictures
wc: Public: Is a directory
    0 Public
wc: Sites: Is a directory
    0 Sites
wc: personal: Is a directory
    0 personal
    0 temp.html
    1 test_1.txt
    4 who_list
    6 total
[localhost:~] dr_unix%

The pipe, on the other hand, takes the output from ls and makes it as the standard input for the wc command.

[localhost:~] dr_unix% ls | wc -l
  13
[localhost:~] dr_unix%

Thus, the backticks give you the line count for each non-hidden item in your home directory, whereas the pipe builds a command that tells you how many non-hidden items there are in the home directory. If this seems slightly confusing at first, don’t worry. Try using pipes and backticks on your own to see the differences; after all, doing is the best way to learn.

A few other ways to combine commands exist. Generally, they are not used on the command line; they are more common in shell scripts. However, they do exist, and we’ll eventually want to write shell scripts anyway, so we may as well be aware of them. The first, and simplest, uses the ‘;’ character to separate commands:

"command1 ; command2"
The command line simply executes the commands sequentially. *command1* is run, and when it's finished, *command2* is run.

```
[localhost:~] dr_unix% who -Hu ; ls
USER     LINE     WHEN         IDLE     FROM
dr_unix  console  May 13 07:08  old
dr_unix  tttyp1 May 16 07:39  .
dr_unix  tttyp2 May 16 07:39  .
Adam.txt Documents Movies Pictures Sites
temp.html who_list
Desktop    Library    Music    Public     personal
```

test_1.txt
```
[localhost:~] dr_unix%
```

If you enclose commands in parentheses, they form a subshell. The commands are treated as a command group:

```
(command1 ; command2)
```

The idea behind this is to force commands to be grouped together so they are treated as one entity. We saw this when we looked at I/O redirection:

```
[localhost:~] dr_unix% (ls who_list xiphoid >
  temp_out) >& temp_err
[localhost:~] dr_unix% cat temp_out who_list
[localhost:~] dr_unix% cat temp_err
ls: xiphoid: No such file or directory
[localhost:~] dr_unix%
```

When we use a command group by itself on the command line, the shell behaves as if the parentheses weren't there

```
[localhost:~] dr_unix% (who -Hu ; ls)
USER     LINE     WHEN         IDLE     FROM
dr_unix  console  May 13 07:08  old
dr_unix  tttyp1 May 16 07:39  .
dr_unix  tttyp2 May 16 07:39  .
Adam.txt Documents Movies Pictures Sites
temp.html who_list
Desktop    Library    Music    Public     personal
```

test_1.txt
```
[localhost:~] dr_unix%
```

Two related forms also allow you to string together commands in a conditional sense. (Yes, Ye Editor, this is getting close to shell scripting, but this is where it fits.) The first is known as the AND form:

```
command1 && command2
```

The shell will run *command1*. If *command1* fails, the whole command ends and *command2* is not run. However, if *command1* succeeds, *command2* is then run. First consider an example where both succeed.
tcsh finds the ls command, runs it, then runs the who -Hu command. As a
counterexample, consider the following, which tries to run the non-existent command
fred:

```
[localhost:~] dr_unix% fred && who -Hu
fred: Command not found.
[localhost:~] dr_unix%
```

tcsh cannot find the command fred, so it doesn't run the who -Hu command.

The second form is known as the OR form:

```
command1 || command2
```

The shell will run EITHER command1 OR command2. If command1 succeeds, the whole
command ends and command2 is not run. However, if command1 fails, command2 is
then run. First consider an example where command1 succeeds.

```
[localhost:~] dr_unix% ls || who -Hu
Adam.txt Documents Movies Pictures Sites
temp.html who_list
Desktop Library Music Public personal
test_1.txt
[localhost:~] dr_unix%
```

tcsh sees the command line ls || who -HU and, behind the scenes, splits the command line
into the first command (ls) and the second command (who -Hu). It successfully runs the
first command, notes that the two are connected by an OR (||), and ends without running
the second command. And if the first command were to fail?

```
[localhost:~] dr_unix% fred || who -Hu
fred: Command not found.
[localhost:~] dr_unix%
```

Here, tcsh could not find the command fred, which results in a failure. So it looks at the
second command, who -Hu, and attempts to run it (in this case, successfully).
[A final note for people looking ahead: the shell’s definition of success and failure may not be quite the same as yours and mine might be. Consider:

```
[localhost:~] dr_unix% ls not_there
ls: not_there: No such file or directory
[localhost:~] dr_unix%
```

So we would think that is an error; after all, the file *not_there* is not found. So you and I might think that *ls not_there & & who -Hu* will not run the *who -Hu* command. However:

```
[localhost:~] dr_unix% ls not_there & & who -Hu
ls: not_there: No such file or directory
who_list
USER LINE WHEN IDLE FROM
dr_unix console May 13 07:08 old
dr_unix tty1 May 16 07:39 .
[localhost:~] dr_unix%
```

What happens is that *ls* runs, reports the relevant information about the list of files passed in the arguments, and exits successfully. So the shell says "What ho! The first command says it ran successfully; off to the second command, then." That may seem confusing. In fact, it is at first. You can probably get a good handle on how a command reacts from its *man* page or from considering how it would handle different situations. Examine the following:

```
[localhost:~] dr_unix% ls who_list fred & & who -Hu
ls: fred: No such file or directory
who_list
USER LINE WHEN IDLE FROM
dr_unix console May 17 09:28 21:34
dr_unix tty1 May 18 06:56 .
[localhost:~] dr_unix%
```

In this case, *ls* did find the file *who_list* but not *fred*, so *ls* did succeed in listing all of the existing files in its arguments. Clear as mud? Good.

[And as always in unix, there are ways around it. For this particular quirk, using *File Inquiry Operators* aka *File Test Operators* can do the trick. These are covered later.]

Again, just be aware of how things are supposed to work, and don’t be afraid to try little things before building those methods into bigger things. This will teach you a lot about the command line, and indeed about anything for which you take that approach.]

So far we’ve seen different ways to combine several commands into one, using many of the Unix shell's multitasking tools. But what if you want to run separate commands simultaneously? Tune in next week, same geek channel, same geek time.

---

**More Basics For The Shell**

*Part IX of this series...*

*June 21st, 2002*

"Que?"

"Right, let's start at the beginning. English is a non-inflected Indo-European language ..."

- Rowan Atkinson, *Blackadder II*

*This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum. "Que?"

"Right, let's start at the beginning. English is a non-inflected Indo-European language ..."

- Rowan Atkinson, *Blackadder II*
So far, we have built your syntax of tcsh to the point where you can do some fairly complex things. However, before we go too deep, let's step back and add to our breadth of shell knowledge.

One thing you may have noticed so far is that the Unix shell hides a lot of detail from you, or more accurately, hides them for you. Redirecting output to a file or another process may seem trivial, but if you've ever tried this in other command-line languages, you will truly appreciate the simplicity and elegance of the Unix shell. If you tried to do this using other operating systems' command lines, like VMS, ... on second thought, don't even think about it. However, now that we've seen the machine running, let's take a look under the hood.

Recall from way back when that the ls command lists files in a directory:

```
[localhost:~] dr_unix% ls
Adam.txt   Documents   Movies    Pictures   Sites    personal
test_1.txt Desktop    Library    Music     Public   login     temp.html
who_list
[localhost:~] dr_unix%
```

and that the -a option will show all files that start with a `.`:

```
[localhost:~] dr_unix% ls -a
.
..                        .qt                       Music
.saves-450-localhost~    .saves-456-localhost~   Public
.DS_Store                 .ssh                     Sites
.FBCIndex                 .tcsh_history           login
.FBClockFolder            Adam.txt                  personal
.Trash                    Desktop                   temp.html
.addressbook              Documents                 test_1.txt
.addressbook.lu           Library                   who_list
.tcshrc                   Movies
[localhost:~] dr_unix%
```

Files that start with a `.` are known as "hidden files." Recall that two of them, `.` and `..`, have already been discussed. (`.` is a shorthand for ‘this directory’ and `. ’ for my parent directory’, if you've forgotten.) Normally, you don't notice them, nor would you care to. Hidden files are used for things like storing settings for a particular application. Thus, Unix' `mail` uses the file `~/.mailrc`, `ssh` makes use of a directory called `.ssh`, the `vi` editor uses `~/.exrc` (not `~/.virc`, because it is based on the older `ex` editor), and so on. And on and on ...

One that we will focus on is the `.cshrc` file. As we've said, the default shell for OS X is `tcsh`, an extended, more user-friendly version of the venerable old C-shell (`csh`). When you open a window with Terminal, the startup files `.login` and `.cshrc` in your home directory are run. Actually, if you have a `.tcshrc` in your home directory and you use `tcsh`, that is run instead of `.cshrc`. So when you see me talking about the `.cshrc` file, take that to mean "the .cshrc file if it exists, otherwise the .tcshrc file." If none of these files exist, `tcsh` will not complain. (And if you are using a different shell, consult your local manpage.)

Now, why would you want to have these files which are automatically run whenever you start? Simple; they can make your life easier by automating settings that you would otherwise have to spend minutes typing whenever you logged in. These include, though are not limited to:

- Assigning values to variables, particularly environment variables. (We'll discuss these soon.)
- Setting aliases. Would you like to be able to type `ll` instead of `ls -alrsFt` from the command line? You can set an alias to permit this. (Again, coming soon.)
- If you connect to another system with a different default terminal, you can alter the terminal settings so that the keystrokes are familiar. This is most commonly used to set the backspace/delete key.
- Altering the behavior of the command line editor. `tcsh` comes with a simple command line editor, using the arrow keys to move through your command history. However, it is limited. If you are familiar with `vi` or `emacs`, you may wish to use their syntax to edit commands you've previously typed. (This is particularly nice if you are as inaccurate a typist as I am.)
- Altering the default behavior of completions. (Again, a later column.)

As an example, let's look at my `.cshrc` file. It is fairly sparse, but it does give a flavor of what these files do:

```
[localhost:~] dr_unix% more .cshrc
setenv PATH "${PATH}:/mysql/bin"
bindkey -v
biff y
alias ll 'ls -alF'
[localhost:~] dr_unix%
```

Don't worry yourself too much about the details, but here is what each line does. The first resets the PATH environment variable, appending the directory `mysql/bin` to the current value. The next line tells the command line editor to use `vi`'s interpretations of the various editing commands. The third says to inform me (beep) when I receive mail. The last line allows me to type `ll` instead of the longer, more error-prone
'ls -altF' from the command line and get the same information. Whenever I open a terminal window, ~/.cshrc is run, setting these values and saving me some work. You can imagine how a fussy, persnickety user might want to change all sorts of things and thus create a truly massive .cshrc file.

At this point, you may be wondering why there are two files, .login and .cshrc, which are run at startup instead of one. I've often wondered that myself, going back to when I first used csh in college. (And yes, there were computers back then.) I think the best explanation I got was that someone had the idea to place certain system-wide settings (as in biff y to turn on mail notification) into .login, but things specific to shell scripts would be placed in .cshrc instead. (In my current setup, running a script would mean that /mysql/bin would appear twice at the end of the PATH environment variable. This doesn't break anything, but it is sloppy. I should change that one day.) I don't know how valid this is, but it's the best explanation I ever got. Regardless, I generally put things into .cshrc instead of .login, but your mileage may vary.

As you might imagine, there is an equivalent .logout file that runs when you logout. It is not used too often; I have used it (if at all) for cleaning up directories where I like to dump messes.

For OS X, there are three system files that are also read when you log in or out. They are owned by root, the administrator account. They are /etc/csh.login, /etc/csh.cshrc, and /etc/csh.logout, and each does what you'd think.

In this column we touched on a few basics; in the following columns we will examine some of them more closely, starting the next time with variables.

Working With Variables In The Shell

Part X of this series...

June 20th, 2002

There is nothing in this world constant, but inconstancy.

- Jonathon Swift, "Critical Essays upon the Faculties of the Mind"

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

Variables are a very basic part of any computer language. They allow you to store and manipulate data, of course; but in doing so, they make possible many convenient things, like error checking or setting default behavior of a program (or command line environment). Variables in tcsh are a rather large topic. For now, we will just cover the basics.

There are two types of variables in tcsh: shell variables and environment variables. Shell variables are those known only to the particular shell you are running, be it a shell script or a command line window. While not as useful on the command line as in a shell script, shell variables are too useful to be ignored. (And besides, we want to be able to automate things with shell scripts some day, right?)

In tcsh and csh, a variable's value is set simply by typing

```
set variable_name = value
```

For example,

```
[localhost:~] dr_unix% set i = 9
```

This creates the variable 'i', if it does not exist yet, and assigns the value of '9' to that variable. You could also define the variable without giving it a value. This can be useful in a script in certain contrived circumstances, or in certain reserved or 'built-in' variables that tcsh uses:

```
[localhost:~] dr_unix% set i
```

Now, you might think that, to print out the value of the variable you could just tell the shell to print it:

```
[localhost:~] dr_unix% echo i
```

```
[localhost:~] dr_unix%
```

But you see the problem. The shell needs to differentiate between the character 'i' and the value of the variable 'i'. This is done with a '\', thus:

```
[localhost:~] dr_unix% echo \i
```

```
[localhost:~] dr_unix%
```
Now this bit of inconsistency may cause you to ask "So why the heck didn't they just let you say "$i=9" when you set the variable? Huh, Mr. Unix Geek?" And the answer is: I don't know. `csh` and `tcsh` do this the same way that the first shell, the Bourne Shell (`sh`), does it, so I think this is a case of 'Well, everyone/shell is doing it.' I'm hanged if I know why Stephen Bourne did it like this in the first place, though. However, you are not alone, because Larry Wall asked this, too, when he was creating Perl. Still, we're stuck with it, and to be honest, you'll get used to it very quickly. I just wanted to preempt a lot of e-mails.

By and large, variables are easy to embed within command lines. For example,

```
[localhost:~] dr_unix% set spy = Fred
[localhost:~] dr_unix% echo "I gave the papers to $spy in Munich."
I gave the papers to Fred in Munich.
[localhost:~] dr_unix%
```

And if you want to destroy a variable, just use `unset`:

```
[localhost:~] dr_unix% echo $spy
Fred
[localhost:~] dr_unix% unset spy
[localhost:~] dr_unix% echo $spy
spy: Undefined variable.
[localhost:~] dr_unix%
```

So far so good, but consider the following:

```
[localhost:~] dr_unix% echo $i
9
[localhost:~] dr_unix% echo "$i"
I am on the $i\text{th} page now."
 ith: Undefined variable.
[localhost:~] dr_unix%
```

You see the problem? `tcsh` interprets '$ith' to mean 'the value of the variable ith' and not 'the value of the variable i, followed by the letters t and h'. To get around this, you can use the '${}' construct. This tells the shell 'limit the variable name to what appears between the curly braces':

```
[localhost:~] dr_unix% echo ${i}9
I am on the 9\text{th} page now.
[localhost:~] dr_unix%
```

`csh` and `tcsh` also provide modifiers that allow you to extract parts of the variable's value without a lot of rigmarole using things like `sed`, `basename`, `dirname`, and so on. They can be used by either $variable_name:x or ${variable_name:x}, where 'x' is one of the following values:

<table>
<thead>
<tr>
<th>x</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>r</td>
<td>Root of value (everything but the extension following the dot)</td>
</tr>
<tr>
<td>e</td>
<td>Extension of value (the suffix following the dot)</td>
</tr>
<tr>
<td>h</td>
<td>Head of value (all but last pathname component)</td>
</tr>
<tr>
<td>t</td>
<td>Tail of value (last pathname component)</td>
</tr>
</tbody>
</table>

Some examples will probably best illustrate their use.

```
[localhost:~] dr_unix% set filename = /usr/bin/java-rmi.cgi
[localhost:~] dr_unix% echo ${filename:r}
/usr/bin/java-rmi.cgi
[localhost:~] dr_unix% echo ${filename:e}
cgi
```
These can be very useful, and again I think the only reason that they aren’t so common in shell scripts is that few people know that csh and tcsh offer them.

Before we move on, set has a couple of neat little features that can be handy. First, if you want to set a variable so that you don’t accidentally change it, you can use the ‘-r’ option to make it read-only:

```
[localhost:~] dr_unix% set -r j=30
[localhost:~] dr_unix% echo $j
30
[localhost:~] dr_unix% set j=31
set: $j is read-only.
[localhost:~] dr_unix%
```

Second, set will let you use variables as arrays if you surround the list of values with parentheses:

```
[localhost:~] dr_unix% set k=(Fred Barney Dino)
[localhost:~] dr_unix% echo $k
Fred Barney Dino
[localhost:~] dr_unix% echo $k[1]
Fred
```

Also, if you want a list of all defined variables, just type set with neither options nor arguments:

```
[localhost:~] dr_unix% set
_       setaddsuffixargv    ()autocorrectautoexpandautolistcdpath  /Users/dr_unixcorrect cmdcwd     /Users/dr_unixdefault_tcsh_initdir    /usr/share/init/tcsh
[... and on and on and on ...]
```

Most of the variables you see here are ones that are created by tcsh when the terminal window was opened. They are known as ‘reserved’ variables, because each one is ‘reserved’ by some aspect of the shell. There are many, many of these variables. You could probably go your whole life and never have to worry about most of them. A few, however, are worth noting.

**argv** is an array which contains a list of the arguments that were passed to the shell when it was created. For a terminal window, it will be blank. But when you want to create a shell script, you can say `scriptfile file1 file2 file3 ...` and behind the scenes, the shell will do the equivalent of `set argv = (file1 file2 file3 ...)`. This hides a lot of mess and is quite useful ... though not so much from the command line.

**cwd** is automagically set to your current working directory whenever this changes.

**histfile** tells the shell in which file to store the commands for the command history. (More on this later, I promise.)

**home** is the full pathname to your home directory.

**path** lists the directories that the shell searches through to find the command you type. It is also used by commands like `which`.

**prompt** is the prompt you see at the command line. If what you see with the `set` command is different than what you really see at the prompt, there is formatting going on.

**shell** can be used to remind yourself, either at the command line or in a script, whether you are running tcsh or csh. It is handy if you are writing a script and tcsh does something slightly different from csh.
uid is your user id. It is a number (in my case, 501) that is used to determine who owns a file, etc. Normally, this is not used, because it is easier for humans to remember that ...

user is your username.

Many of the ‘reserved’ shell variables are tied to particular environment variables. We'll cover that subject next time.

Son Of Mac OS X Variables
Part XI of this series...
July 12th, 2002

Everyone wants to save the environment, but no one wants to help Mom wash the dishes.
- P.J. O'Rourke

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

In our previous column, we looked at shell variables: what they are, how to set them and unset them, etc. We ended with a quick perusal of reserved shell variables that the shell uses to hold certain values that are used for different parts of the shell. Normally, these are stored in environment variables.

Environment variables are a special type of variable that are used to hold certain system-wide settings for a user. As an example, let's say that you are familiar and comfortable with the vi editor or the emacs editor (not to be mistaken for Apple's new eMac computer). Occasionally, a command-line program will want you to enter data and will use a built-in editor to provide a way to do this. Often a command-line program's default is to use either ed or ex, which shouldn't happen to a dog. It would be nice if this program would be able to say "Okay, operating system, fire off a new process using the user's preferred editor," at which point the operating system says "Right-ho!," looks up your preference, and fires up the appropriate editor. In Unix operating systems, this is done via environment variables. When the program wants to fire up an editor, and wants you to decide which to use, the system goes to a lookup table where it finds where you have set your editor of choice and starts it.

Unix shells allow users to access and alter the values in this lookup table via environment variables. By convention, environment variables are in all capital letters, while shell variables are all lower case. You don't have to follow this convention, of course, when defining your own environment variables. However, it is a de facto standard; all of the built-in environment variables are in all caps. Also, it does prevent confusion ... well, lessens it. So, like a good libertarian, I'm not saying that you have to do it, but I am saying that you really should.

To find out which environment variables are defined in your current shell, either by you or for you, simply use the env Unix command. You will see:

```
[localhost:~] dr_unix% env
HOME=/Users/dr_unix
SHELL=/bin/tcsh
USER=dr_unix
PATH=/bin:/usr/X11R6/bin:
:/usr/local/bin
:/usr/bin
:/usr/sbin:
:/bin
:/usr/sbin:/bin:
:/usr/local/bin
:.
__CF_USER_TEXT_ENCODING=0x1F5:0:0
TERM=vt100
TERMCAPI=d0|vt100|vt100-am|vt100am|dec vt100:
:do="^J"co^80:li^24:cl^E[;H\E[2
J:sf=2+^ED:  :le=^H:bs:am:cm=5^E[di%d;\dH:nd=2^E[C:up=2^E[A:
:ce=3^E[K:cd=5
0^E[7:so=2^E[7m:se=2^E[m:us=2^E[4m:ue=2^E[m:
:md=2^E[1m:mr=2^E[7m:mb=2^E[
5m:mm=2^E[m:
:rf=/usr/share/tabset/vt100:
:rs=^E[?31l^E[?41l^E[?75l^E
[?7h^E[?8h^E[r^E[0m^E[B^E]B^E[2J:
:ks=^E[?1h^E=:ke=^E[?1l^E:
:ku=^EOA:k
d=^EOB:kr=^EOC:k1=\EOD:kb=^H:
:ho=^E[H:k1=\EOP:k2=\EOQ:k3=\EOR:k4=\EOS:pt:sr=2*
\EM:vt#3:xn:  :sc=^E7:rc=^E8:cs=^E[1%d;%dr:
TERM_PROGRAM=Apple_Terminal
```
... or something like it. You can see that there are quite a few environment variables set when you open a terminal window. What some of them represent should be fairly obvious, such as OSTYPE=darwin, VENDOR=apple, USER=dr_unix, and so on. On the other hand, there is that big, ugly, massive value for TERM_PROGRAM. If that one is obvious to you, seek professional help. Now.

(In case you are curious, TERM_PROGRAM holds a multitude of settings about your terminal ... which type of terminal you are emulating; what functions do various keys perform, like linking the delete key to the delete action; and so on. Don't worry about it, and don't change it; otherwise, you might do yourself mischief.)

You can create a new environment variable, or change the value of a current environment variable, with the csh/tcsh command `setenv`:

```
setenv VAR value
```

For example,

```
[localhost:~-] dr_unix% setenv FRED Uncle
[localhost:~-] dr_unix% echo $FRED
Uncle
[localhost:~-] dr_unix%
```

Notice that you can use the value of an environment variable just like a shell variable, with the '$' or the '${}' construct to avoid confusion.

```
[localhost:~-] dr_unix% echo "$FRED" B.S., where were you November 3, 1963?
Uncle B.S., where were you November 3, 1963?
[localhost:~-] dr_unix%
```

[Author's note: Uncle B.S. is copyright Tim Wilson.]

Likewise, you can create an environment variable without assigning it a value.

```
[localhost:~-] dr_unix% echo $JOE
JOE: Undefined variable.
[localhost:~-] dr_unix% setenv JOE
[localhost:~-] dr_unix% echo $JOE
[localhost:~-] dr_unix%
```

This distinction can be important in scripts, because 'undefined' is not quite the same as 'defined but with no value'. Furthermore, if you provide `setenv` with no arguments, it lists all the environment variables and their values, just like `env`:

```
[localhost:~-] dr_unix% setenv
HOME=/Users/dr_unix
SHELL=/bin/tcsh
USER=dr_unix
PATH=/bin/powerpc-apple-darwin:/Users/dr_unix/bin:/usr/local/bin:/usr/bin:/bin
:usr/local/sbin:/usr/sbin:/sbin:/sbin:/usr/local/mysql/bin:
__CF_USER_TEXT_ENCODING=0x1F5:0:0
TERM=vt100
TERM_PROGRAM=d0|vt100|vt100-am|vt100am|dec vt100:
do="^J:co#80:li#24:cl:\E[;H\E[2J:sf=2\ED:   1e="^H:bs:am:cm=5%E%i%d;%dH:nd=2%E[C:up=2%E[A:
:ce=3%E[K:cd=5
0\E[J:so=2%E[7m:se=2%E[m:us=2%E[4m:ue=2%E[m:
:md=2%E[1m:mr=2%E[7m:mb=2%E[5
```
Note that $FRED and $JOE have now shown up. Now, given that you can set an environment variable, you might suspect that you can also unset a variable; and you would be right, you clever person. The `unsetenv` command does this:

```
[localhost:~] dr_unix% unsetenv JOE
[localhost:~] dr_unix% echo $JOE
JOE: Undefined variable.
```

As you might imagine, there are quite a few environment variables that are reserved to have particular meanings in `tcsh` and `csh`. The most useful ones in my experience, are:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>EDITOR</strong></td>
<td>The complete pathname to your preferred command-line editor. Usually this is <code>/usr/bin/vi</code> or <code>/usr/bin/emacs</code>.</td>
</tr>
<tr>
<td><strong>HOME</strong></td>
<td>Your home directory</td>
</tr>
<tr>
<td><strong>HOST</strong></td>
<td>The current machine name (tcsh only). Admittedly, this is not too useful if you aren’t on a network.</td>
</tr>
<tr>
<td><strong>PATH</strong></td>
<td>The directories where the shell looks, in order, for commands that you type.</td>
</tr>
<tr>
<td><strong>PRINTER</strong></td>
<td>Your default line printer.</td>
</tr>
<tr>
<td><strong>PWD</strong></td>
<td>Your current working directory. (This is what the <code>pwd</code> command uses.)</td>
</tr>
<tr>
<td><strong>SHELL</strong></td>
<td>Which Unix shell you are running. (This could be tcsh, csh, ksh, bash, zsh, sh, and so on. Darwin defaults to tcsh.)</td>
</tr>
<tr>
<td><strong>USER</strong></td>
<td>Your username.</td>
</tr>
<tr>
<td><strong>VISUAL</strong></td>
<td>Your preferred full-screen editor. Generally <code>/usr/bin/vi</code> or <code>/usr/bin/emacs</code>.</td>
</tr>
</tbody>
</table>

If you have found others to be useful, please let me know.

`$PATH` deserves special mention. When you type a command at the command line, or in a shell script, `tcsh` will use `PATH` to find the command. `PATH` contains a list of directories, separated by colons. `tcsh` will look in the first directory; if it finds the command, it runs it, otherwise it moves to the next directory. If it is not found at all, `tcsh` prints an error message. Consider an example using my `PATH`:

```
[localhost:~] dr_unix% which cp
/usr/local/bin/cp
```

If you type a command at the command line, or in a shell script, `tcsh` will use `PATH` to find the command. `PATH` contains a list of directories, separated by colons. `tcsh` will look in the first directory; if it finds the command, it runs it, otherwise it moves to the next directory. If it is not found at all, `tcsh` prints an error message. Consider an example using my `PATH`:
If, at the command line, I tell tcsh to run the command my_script, tcsh will look for the file /Users/dr_unix/bin/powerpc-apple-darwin/my_script. If there is no such file, it looks for /Users/dr_unix/bin/my_script, then /usr/local/bin/my_script, and so on, until it finds one. As mentioned above, the shell is looking for the same file that my_script (or any file with 'my_script' in its name) would be in if I were using this shell to run a shell script. If I have a file 'my_script' in any of these directories, the shell prints an error:

```
[localhost:~] dr_unix% my_script
my_script: Command not found.
[localhost:~] dr_unix%
```

Also, if you create a shell script that is the same name as another script in your $PATH, which one is run first depends on which is encountered first as the shell sequentially searches the directories listed in $PATH. So not only is it sensible not to create a shell script called ls because it is a standard tool, you can see now exactly how this can cause confusion. If this situation does arise, the which command is useful. It tells you which particular file will be run, if any, when you type in a command:

```
[localhost:~] dr_unix% which ls
/bin/ls
[localhost:~] dr_unix% which my_script
my_script: Command not found.
[localhost:~] dr_unix%
```

One of the primary reasons that csh was created was that the syntax is similar to the C programming language. The idea was that, by creating a shell with syntax similar to C, the learning curve would be much smaller and less steep. C and the Bourne shell (the first Unix shell) are obviously quite different languages. One of the features of C is that global variables (which are sort of analogous to a shell's environment variables) are not, generally, lower case; in C, those are generally used for macros and definitions. To reconcile this feature of C with this feature of Unix shells, csh has something called 'paired variables'.

```
[localhost:~] dr_unix% echo $term
vt100
[localhost:~] dr_unix% echo $TERM
vt100
[localhost:~] dr_unix% set term=vt200
[localhost:~] dr_unix% echo $TERM
vt200
[localhost:~] dr_unix% setenv TERM vt100
[localhost:~] dr_unix% echo $term
vt100
[localhost:~] dr_unix%
```

Note that, though the two values are tied together, the shell variable is still used and manipulated like a normal shell variable; ditto for the environment variable. Note, in the following example, how SPATH andSPATH are tied together, but each uses its own particular syntax. If we modify Spath it alters SPATH as well.

```
[localhost:~] dr_unix% echo $PATH
/Users/dr_unix/bin/powerpc-apple-darwin:/Users/dr_unix/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/mysql/bin:. 
[localhost:~] dr_unix% set path=($path ~/bin)
[localhost:~] dr_unix% echo $PATH
/Users/dr_unix/bin/powerpc-apple-darwin:/Users/dr_unix/bin:/usr/local/bin:/usr/bin:/bin:/usr/local/sbin:/usr/sbin:/sbin:/mysql/bin:~/bin
[localhost:~] dr_unix% setenv PATH "${PATH}:/Users/dr_unix/Desktop"
[localhost:~] dr_unix% echo $path
/~/bin
[localhost:~] dr_unix% echo $PATH
/Users/dr_unix/bin/powerpc-apple-darwin:/Users/dr_unix/bin:
```

Paired variables are a combination of a reserved shell variable and a reserved environment variable whose values are tied together. Both variables are set to point to the same spot in memory, so changing the value of one will also change the value of the other. For example, TERM and STERM both refer to the type of terminal one is emulating.
Personally, I don't depend on paired variables because other shells don't use them. Since I use $ksh$ at work and the default on OS X is $tcsh$, I prefer to use the environment variables for anything environment-related. Again, you don't have to do this, but you will find it easier to move between shells and Unixes ($Unic7$?) if you do so. And this is supposed to be about making life easy, isn't it?

By now you are probably ready to jump in and modify/create your $ .cshrc $ and $ .login $ files. However, you first must know how to edit files on the command line, which we will start in the next column.

Command Line Text Editing Basics

Part XII of this series...
July 26th, 2002

This method answers the purpose for which it was devised; it saves lazy editors from working and stupid editors from thinking.
- A.E. Housman, "The Editing of Manilus"

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

So far we have done quite a bit with manipulating files and examining information on them: their contents, properties, and so on. However, we have not yet discussed creating a file from scratch. True, we have shown how to create a file by redirecting Standard Output, and while this is powerful, it is not always the most convenient way to create all files. To do this, we need to know how to use an editor, so we will now learn the basics of $vi$.

Now before we start, the $emacs$ users must pipe down and quit lobbing rotten vegetables at the stage. For those of you who are new to Unix and the command line, the $vi$ vs. $emacs$ debate may have been the closest thing to a holy war that the Unix community has ever seen. Each side's adherents extol the virtue of their respective editor of choice and some feel the need to denigrate the other. This is complete eyewash, of course, as each editor has its virtues. I've chosen $vi$ for four reasons. First, it isn't as intimidating for a newcomer; $emacs$ has a plethora of commands, while $vi$ requires less to learn up front and doesn't present options that could be confusing to a newbie. Second, there are some Unix systems which still don't come with $emacs$ or is not installed by the sysadmin, while $vi$ is ubiquitous; this is hard to believe, but true. Third, Ye Editor probably wouldn't pay me the same huge commission for all the articles that would be required to cover $emacs$. And fourth, I am using $vi$ to write these columns as an example of what can be done from the command line. Besides, we all know that $vi$ people are superior humans, so phblt! to the $emacs$ lusers.

$vi$ is based on the venerable $ex$ editor, which is now more or less the Unix equivalent of the horse and buggy; it gets you there, but not what you want to use unless you must. Still, $ex$ did have its points, and $vi$ will let you run those useful $ex$ commands; more on that later. $vi$ has the advantage of being, get this, a full screen editor which lets you look at more than one line at a time. This seems like no big deal, but when $vi$ premiered, it was amazing. (Or so I'm told; I haven't been computing that long.) And in the grand tradition of Unix utilities, $vi$ makes simple things simple and complex things possible.

To create a file with $vi$, simply type:

```
vi filename
```

In this column, we will step through some of the basics so you can get started with $vi$. At the command line, type:

```
vi testfile
```

This should clear your terminal screen and give you something that looks like:

```
You may be wondering what's with all the tildes (~). Remember that vi is a full screen editor. However, when you give it the name of a file that doesn't exist, or a file with only a few lines, you can have more lines on the screen than lines in the file. To denote this, vi will indicate "this is beyond the end of the file, pal" with a tilde at the beginning of a line on the screen.

vi has two modes: a command mode and an input mode. When it starts, you are in command mode. To enter input mode (i.e. to insert text), just type the letter "i" and start typing. In your terminal, type and "i" and add the following. (Don't worry if you mistype something, I'll show you how to replace things in a bit.)

Now is the time for all good men (and women, of course) to come to the aid of the party.

We're out of ice cream.

This is a very long line, with more than the eighty characters that you will normally see on the terminal. Notice how it wrapped around?

Work is the curse of the drinking classes. - Oscar Wilde

When you have finished typing, hit the ESCAPE key (upper left of the keyboard) to go back to command mode; your cursor should be over the 'e' in Wilde. In fact, if you are in vi and you can't remember whether you are in command mode or input mode, hitting ESCAPE will always take you out of input mode; if you are already in command mode, you hear an error beep, but no harm is done.

You may well have made mistakes when typing that. After all, we're only human. Well, you are. One of the easiest way to reverse anything you wanted to do is with the undo command, "u". This will put things back to the way they were before your previous change. (Moving around does not count as a change.) This simple little command has saved my bacon more than once; keep it in mind.

If you are in command mode, you can move one space left, right, up, or down using the arrow keys. Yes, it's obvious and it should go without saying; that's why I said it. Back in the day, however, you couldn't count on having these keys on your keyboard or having them mapped the right way. In old Unix utilities, 'h', 'j', 'k', and 'l' do this: 'h' for left, 'j' for down, 'k' for up, and 'l' for right. (Anyone who plays nethack will find that familiar.) In fact, I half-recommend you use these letters; if you ever wind up on another Unix box, the keyboard may have the arrows set up a bit differently and you'll need to use hjkl. Take a few second to move around and get familiar with them.

Now, using hjkl can get the job done, but they can be tiresome, particularly if you have to traverse a whole line. Fortunately, there are some shortcuts to promote the virtue of laziness. First, vi will let you run almost any command repeatedly by prepending a number to it. (These commands are said to 'take a count' in the vernacular; I don't know if that's an official term.) Go all the way back to the beginning of the first
line. Say you want to move eight characters to the right. Instead of typing '8l' eight times, you can simply type "8l" to do the same thing. Try this; it should take your cursor to the 'h' in 'the'. "3j" will then take you to the 't' in 'out'.

Also, notice the line that starts "This is a very long ..." Assuming that you didn't hit RETURN after the 'nor' in 'normally', vi wrapped the line on the screen, but it still considers it one line. Trying to use "j" and "k" to move up and down through the line. You originally typed this as one line, so vi stores it internally as one line. That's why you may see the cursor moving through what seems to be more than one line on one "j" or "k".

In addition, there are other handy movement commands. A '0' (zero, not a capital 'o') will take you to the beginning of your current line; a '$' will take you to the end. A 'w' will advance you to the beginning of the next word; and yes, "3w" will advance you three words. Likewise, the 'b' will move you back one word; "2b" will take you back two. And if you want to advance to the end of a word, use the "e" command, which also can take a count.

So now we can fly around our test file with the greatest of ease. However, we still need to change things. If you forgot a letter, you can of course use the 'i' command to insert it. (Don't forget the ESCAPE.) The 'x' command will delete the character under the cursor. "x" also takes a count, like the movement commands. "5x" will delete the character under the cursor, plus the next four (for a total of five). "$" will replace the character under the cursor with the next character you type. Let's say you typed 'them' when you meant to type 'then'. You can just place your cursor over the 'm', then type "rn" and the word is fixed. Now, having said that, I must warn you. 'r' takes a count, but it may not do what you think. If you type 3rn the next three characters will all be replaced with 'n'. If you would like to replace, say, 'and' with 'but', use the "R" command. It will overwrite characters as long as you type; to escape from this form of input mode, you should, of course, hit ESCAPE.

Two other commands deserve attention: the delete and change commands. Delete, 'd', has two forms. You can use it to delete a word, "dw", or to delete a whole line, 'dd'. Both of these forms can take a count, as can both forms of the change command, 'c'. To change a word, use 'cw'; to change a whole line, use 'cc'. And as you can probably guess, once you are done typing in the change, you hit ESCAPE to get back to command mode.

With these commands, you can now get to anywhere in your file and correct any errors you made. Why don't you go ahead and do this? It will give you some nice practice, so even if you did type it properly, add a couple of mistakes, then change them back. Remember: ESCAPE takes you out of insert mode and into command mode. So toddle along; we'll wait.

Ah, back so soon? [HIDES ADULT BEVERAGE] Splendid. Capital. Now there is one thing we must do: save the file. vi offers many different ways to do this, depending on just what you want to do. If you want to save the file and leave vi, the "ZZ" command will do just that. However, if you want to continue editing after saving the file, just type ":w" while in command mode. If you wish to quit without saving your changes, use the ":q" command. (For those who are wondering, typing a colon while in command mode will allow you to enter an ex command. The reasoning here, I think, is that ex provided good commands to do some things already, whether the editor was a line editor like ex or a full-screen editor like vi, so vi just lets the commands it could inherit from ex work without vi's author(s) having to rewrite code.)

So, there you have the basics of vi. That isn't everything, of course. In the next column, we'll examine vi in more depth.

More Command Line Text Editing

Part XIII of this series...
August 16th, 2002

"Scene Two."
"I've seen three!"
- David Hatch and Bill Oddie, I'm Sorry I'll Read That Again, "Macbeth"

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

The previous column took a look at how to do simple, basic things in the vi editor. This was enough to be getting on with, as the saying goes, but there is much more available to you. As promised, we'll take a fuller look at the commands available to you in this installment. For now, type:

vi testfile

This will re-open the file we used last week (assuming you saved it) in vi so you can try out the various commands in today's column. I won't be telling you exactly what to do and where, because you are probably old enough to experiment (or mess around) on your own.

If you happen to find a list of the vi commands in alphabetical order, you will notice that many of the commands invoked by the uppercase and lowercase of the same letter seem to be associated. For example, we have seen that 'r' will replace one character of text, while 'R' will place
you in insert mode, replacing text until you hit ESCAPE. Where "x" deleted the character under the cursor, "X" deletes the character before the cursor.

Often, the uppercase command does something similar on/to/about an entire line. For example, an "i" will put you into insert mode, while an "I" will put you in insert mode at the beginning of the current line. An "a" will also put you in insert mode, appending text after the cursor, while an "A" will append text, not where the cursor is now, but at the end of the current line. And where a "u" undoes your previous command, a "U" restores the current line to its previous state.

The "C" and "D" commands are also similar to their lowercase counterparts. Recall that "c" is the change operator; you can use it to change a word, a line, etc. "C" will change text from the cursor to the end of the line. (As always, hit ESCAPE to stop changing.) Likewise, where "d" is the delete operator, "D" will delete all characters from the cursor to the end of the line.

"W", "B", and "E" are the same as their lowercase counterparts, but instead of moving a line at a time (recall that "h", "j", "k", and "l" do this), they ignore punctuation when moving forward/backward one word. For example, "w", "b", and "e" would see the word "We're" as three words: "We", "e", and "re". "W", "B", and "E" see it as one. It may seem like these three commands have limited usefulness, and I wouldn't disagree. However, they can be useful for mapping; more on that later.

Two other sets of commands are more useful in mapping than in day-to-day editing. "j" and "J" will search forward and backward, respectively, the current line for a character. Thus, if you type fj in command mode, either the cursor will move forward to the next 'x' on the current line, or the system will beep for an error if there is no 'x' after the cursor. "j" and "J" are nigh on the same thing, but not quite. They will search forward ("j") or backward ("J") in the current line for the character before the character you provide. Thus, typing fj in command mode will move the cursor forward to the character before the next 'x' on the current line, or the system will beep. Again, these are mostly used for mapping.

"o" and "O" are another pair. "o" will open up a new line for editing after the current line. "O" does the same, but before the current line.

However, not every pair of lowercase/uppercase letters correspond to similar commands. For one thing, commands are not defined for every letter. For example, "q" is undefined, while "Q" will place you into the ex editor. (If you want to get back to vi, just type "vi" at the colon.) Similarly, "g" is undefined, but "G" moves the cursor to the last line in the file. (This is the same as invoking the ex command :S :G will, oddly enough, take a count, but it doesn't move you to the last line repeatedly. (That would be silly.) Typing ng in command mode will take you to line ... well, whatever number you use for n. (This is the same as invoking the ex command :n where n is, of course, the line number.) If you don't know what your current line number is, just type a CONTROL-G (or, if you prefer, a :=.G to run the ex version of the command.)

The corresponding uppercase letters of the movement commands (hjkl) do not mimic their lowercase counterparts. "H" moves you to the top line of the screen, not the beginning of the line. (Recall that "0", zero, does that.) "S" will join the following line to the current line. This can be quite useful if you are doing a copy-and-paste of text that takes up more than one line. "K" is undefined. "L" takes a count and moves you to that number of lines before the last line on the screen. With no count, it moves the cursor to the last line. (A side note here is that "L" uses lines on the screen, not lines as determined by when you hit RETURN.)

Now, three basic functions of an editor are to cut, copy, and paste text. We have seen how to cut text with "d" and "D". We can, of course, copy text by highlighting it and selecting it from the dropdown menu at the top of the computer screen. Likewise, by typing "i" to go into insert mode, we can use the dropdown menu to paste text. While that is useful, it is not the Unix way. Also, vi was written before GUIs were around. (At least I think it was.) Therefore, you would expect vi to have the ability to copy and paste text, and you would be right, you clever person.

When you delete text, it is copied into a buffer so that it can be pasted later. If you just want to copy, not cut, vi lets you do that, too. The "y" operator will copy (yank) text into a buffer (without deleting it) so it can be pasted. yy will copy a word; yy will copy a line. Both take a count. Also, you can use different buffers. vi stores the text from the last nine deletes in separate buffers (numbered 1-9). Typing "yy" (a quote followed by a letter followed by two y's) will copy the text from the current line into a buffer labeled y. This allows you to copy to, and to paste from, several buffers at the same time, rather than the standard one buffer that the desktop uses.

The "p" and "P" commands let you paste/put text. When you have text in a buffer, "p" pastes the text after the cursor; "P" pastes it before the cursor. Now, given that you can yank text into a buffer that you label, it's no surprise that you can also paste from those buffers. "p" takes a count, sort of; typing "np" (double quote followed by a number up to nine followed by a 'p') will paste the text in delete buffer number 'n' after the cursor. Typing "ap" (double quote followed by the buffer's letter label followed by the letter 'p') will paste the text from that buffer before the cursor. Congratulations, you now have a cut-and-paste utility that is more powerful than GUI editors.

There is one more form of pasting. If you want to read text from another file and include it wholesale into the file you are editing, there is an ex command that lets you do this. By typing :r filename, the contents of the file filename get copied into the file after the cursor. Again, this can be a big time saver.

Okay, back to mapping. (And you thought I'd forgotten.) There are several letters that do not have commands associated with them: "y", "k", "v", "t", and several CONTROL- characters. These can be used to create aliases so that one keystroke can replace several. For example, I often type 'teh' for 'the' in spite of all my practice. It happens all teh --, er the time. To correct it, I always move the cursor to the 'e', type an x to
delete the 'e' (which places the cursor over the 'h'), then type a 'p' to paste the 'e' after the 'h'. Since I do this a lot, I'd like to be able to replace those two keystrokes with one. vi lets me do this every time I start it. And the best part is, I can use vi to set it up. Can I get a w00tness?

At the command line, cd to your home directory and look for the file .exrc:

```
[localhost:~] dr_unix% ls -a .exrc
ls: .exrc: No such file or directory
[localhost:~] dr_unix%
```

In my case, it isn't there, so I don't have to worry about copying it (cp) before changing it. (This is always a good idea if you don't know what you are doing. I rarely do.) So, after a cp .exrc .exrc_bak if that applies to you, type the following at the command line:

```
[localhost:~] dr_unix% vi .exrc
```

which gives the usual

```
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~
~

.exrc: new file: line 1
```

Now what I want to do is set the command "v" to switch the letter under the cursor with the one after it. This is done by issuing the command "x" followed by the command "p". So, in the .exrc file, I add the following line:

```
map v xp
```

After ESCAPE-ing to exit input mode, I type the "ZZ" command to save the file and quit vi. Now I can test if this works. At the command line I type:

```
[localhost:~] dr_unix% vi maptest
```

and I add the line:

```
This is teh end.
```

Hitting ESCAPE to go back to command mode, I move the cursor over the 'e' in 'teh'. I then enter the "v" command, wave a wand, and as if by Unix, the line becomes:

```
his is the end.
```

While changing a two-keystroke command for a one-keystroke command isn't that great an improvement, these maps can, of course, be as complex as you want. (The usefulness of overly complex maps is for you to decide.) For example, if you have a tendency to switch two words as you type, you can add the following to your .exrc file:

```
map V dwElp
```

This creates a new command, "V", which will delete a word ("dw"), move to the end of the word that is now under the cursor (using "E" instead of "e" in case the word contains an apostrophe), moves one character to the right ("l"), then pastes the word that was just deleted ("p"). Voila, you are now a vi power user, a Unix genius, and all-around geek-about town.

Of course, I saved the best until last. The "?" and "?" commands are another associated pair. Strictly speaking, "?" is not the uppercase version of the "?" character, but they are on the same key, and that's close enough for jazz. The "?" and "?" commands let you search forward and backward, respectively, from the cursor for the regular expression you provide after the command. They --

What's that? We haven't covered regular expressions yet? [Looks at watch.] Oh, dash it, that will have to wait until the next column.
This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

In the previous column (vi II: Electric Boogaloo) we left our hero vi (Ben Affleck) trying to create a new command (George C. Scott) that would allow the user (Ozzy Osbourne) to search for plain text (Robert Ludlum) in a way that was both slim and elegant (Callista Flockhart and Diana Rigg) instead of fat and bloated (Steve Balmer). How will he do it? Will Professor Tinkle discover the secret of the alien invasion? Will the cavalry (The Kiss Army) reach the beleaguered fort on time? Can Teddy the Wonder Ferret steer The HMS Wigglesworth away from the reef and save the princess from the clutches of The Dread Pirate Snodgrass (Eric Idle)? Tune in ... now!

Okay, now that I've taken my medicine, perhaps you should as well. Today we will cover a topic that may give you the cold sweats and curl your nose hair: regular expressions. On the other hand, once you see what you can do with regular expressions, you might get a geekgasm due to the unutterable coolness.

Having several friends who do not speak American English as their native language, I often find myself facing a befuddled look or a guffaw because I will use a phrase that is not taught in English classes. (I would offer an example, but this is a family Web site. What can I say, I grew up with preachers' kids.) In truth, English can be a dangerous language in this regard; it is chock full of expressions. If you hear someone say "That let the cat out of the bag", you know that there was no actual cat in an actual bag; it's just an expression that is not to be taken literally, but within the context of what is happening at that time. Regular expressions are Unix' equivalent to such phrases.

Many Unix tools use regular expressions: ed, ex, vi, emacs, sed, awk, grep, egrep, and so on. Regular expressions were first written for Unix by a man named Henry Spencer. If I remember correctly, with only minimal features. As each tool was written, some authors would enrich the regular expression syntax for that particular tool; unfortunately, that enrichment was not always seen in the other tools that existed. (Perl is probably the king of this; I sometimes see features in its regular expression syntax that I cannot imagine anyone ever needing ... but apparently someone did.) We will cover the basic syntax that is common across all utilities that use regular expressions, then move on from there.

Before we start, open our previous test file with the vi testfile command; once again, we will use it for practice. In command mode, type /the and hit RETURN. This should move the cursor to the next occurrence of the word 'the'. What happens is that vi saw the search command, "/" and noted that it was followed by 'the'. vi then began to search forward for the pattern 'the': a 't', followed by an 'h', followed by an 'e'. Finding this, vi then moved the cursor to the place where this pattern began. While a cool (and time-saving) feature, just matching a plain old string is a bit limited. For example, the pattern 'the' will not only match the word 'the', it will also match the beginning of 'then', the middle of 'apothecary', and the end of 'breathe'. Worse, it would miss the word at the beginning of a sentence, because 'The' is not the same as 'the'. The computer does not store a capital 'T' as the same value as a lowercase 't'. What we really want is something more clever and more flexible, a way to say 'look for the next occurrence of the word 'the' and only the word 'the', whether it is at the beginning of a sentence or not'. That's where regular expressions come in.
A regular expression uses symbols to represent text in an abstract way that is not to be taken literally. They are a bit like the **metacharacters** that the shell uses, but please don't confuse them. For one thing, the syntax is quite different. An asterisk, for example, means something different as a metacharacter than in a regular expression. Also, a metacharacter can be used pretty much anywhere on the command line, while a regular expression can be used only when and where a particular tool/utility is expecting one. (In the case of **vi**, this is when you hit the "/" command to search forward or "?" command to search backward.)

As you could probably guess from our example above, a letter will match itself, as will a number, and a space matches a space. Obvious, really, but we've all missed obvious things in the past.

What if you want your expression to be more flexible? Suppose you are looking for the next occurrence of any of 'the', 'tee', 'tie', or 'toe'. What you want is a wildcard to say "a 't' followed by any character followed by an 'e'". The period, ".", does this. It represents any character. In **vi**, you would type `/t.e` and the cursor would move to the next occurrence of 'the', 'tee', 'tie', or 'toe'.

Or would it? Remember that the "." can represent any character. What if the word "intrepid" were in the file? That contains a 't', followed by some character (an 'r'), followed by an 'e'. Suddenly that "." is looking a little too flexible. What you'd really like to say is "a 't'; followed by one of 'h', 'e', 'i', or 'o'; followed by an 'e'".

And yes, kiddies, this can be done. We can enclose our list of choices in square brackets thus: `/[theio]e` and **vi** will move the cursor to the next occurrence of 'the', 'tee', 'tie', or 'toe'.

Two characters have special meaning in a list. Suppose that we want to move to "intrepid", the next occurrence of 't.e' except for 'the', 'tee', 'tie', and 'toe'. It's quite simple; use the square brackets again, but precede the list of letters with a carat, `^` (The underscore, `_`, is considered a valid word character in Unix for reasons that are just far too boring to go into.)

Now, if you wanted to match 'any capital letter', or 'any digit', you could type `/[A-Z0-9]` or `/[A-Za-z0-9]`. But why? That is a whole lotta typing; there should be an easier way, and there is. A range can be denoted by a dash, `^-`. For those who like their geeky goodness, a dash denotes all characters from the first to last, inclusive, in ASCII order. If this means nothing to you, don't worry. Just remember that `[0-9]` will match any digit (and remember that computers -- well, Unix computers -- start counting at 0). `[A-Z]` will match any upper case letter, `[a-z]` any lower case letter, and any 'word character' can be matched by `[A-Za-z0-9-_]`. (The underscore, `_`, is considered a valid word character in Unix for reasons that are just far too boring to go into.)

And what if you want to match a list of characters that include the carat and/or the dash? Just make sure that the dash goes first and the carat isn't first in the list of characters between the `[]`. So, to find either a carat or a dash in **vi**, type `/[^-^]`.

As previously mentioned, the asterisk has a particular meaning in a regular expression. It indicates that one must match zero or more of the preceding character. So, in **vi**, the command `/te*` will move the cursor to the next occurrence of 't', or 'te', or 'tee', or 'tee' etc. That seems silly, but regular expression symbols can be combined to suit your purpose. Therefore, in **vi**, typing `//[Ti]he.*lawyer` will move the cursor to the next place where 'the' or 'The' is followed by a space, followed by any number of any characters (except a newline character), followed by a space, followed by 'lawyer'. This will match 'the slimy lawyer', 'the greedy lawyer', 'The evil, scum-sucking lawyer', and even 'The snowball was thrown at a deserving lawyer but hit another who probably deserved it, being a no-good, thieving, corrupt government lawyer'.

Notice one thing about that last phrase. The word 'lawyer' appears twice in the sentence. Regular expressions tend to be greedy, as the saying goes. They will match as large a chunk of text as they can. This doesn't mean anything when you are just searching for text. However, when we get to more advanced commands that can replace a regular expression with a string, this is worth knowing; you wouldn't want to take out too much text.
So far so good. But wait, there's more. Suppose you want to go to the next place that is matched by `/\[Tt\]he .* lawyer`, but only if it appears at the beginning of the line? This is done with the carat, `^`. Yes, yes, I know, we've already used that in a list of characters. I'm sorry, but I didn't make these rules. I only use them, and this is another case of playing the hand you're dealt. Anyway, to force the search to start at the beginning of the line (or 'anchor it'), use the command `/\[Tt\]he .* lawyer` instead. If you want to anchor the search to the end of the line, the dollar sign is used: `/\[Tt\]he .* lawyer$`.

At this point, you may be wondering (then again, you may not) about matching one of those special characters. After all, there will come a time when you want to match an actual asterisk or dollar sign, or even a `/`. The "\" acts as an escape character; it tells the regular expression interpreter "The following character may have a special meaning to you. However, ignore that, and use it literally." So, in `vi`, to advance the cursor to the next occurrence of 'St. Paul', just type `/St\. Paul`, and of course, it escapes itself, so to match a `\`, you enter `/\`

These symbols, `/`, `\`, `*`, `^`, `$`, `[]`, and `\[`, are the core of Unix regular expressions. The authors of some tools that use regular expressions added features that they found useful. However, they also tried to build on what was done before, so if a symbol has a special meaning in a regular expression for one Unix tool, it will have that special meaning in other tools, or no special meaning at all. For example, `ex` and `vi`, which you'll recall is based on `ex`, allow you to anchor searches at word boundaries. This is done with the `\<` and `\>` pairs of symbols, which anchor a match at the beginning and end of a word, respectively. It will be remembered (how could it be forgotten?) that we initially wanted to match the word 'the'. These word-boundary anchors allow us to do that. By typing `/<the`, we will match the word 'the' but not the word 'breathe'. `/the>` will match 'the' and 'breathe', but not 'theater'. To match the word 'the' and none other, and to include matches at the beginning of a sentence, we use `/<the>`. To include matches where 'the' is the first word in a sentence, this becomes `/<\[Tt\]he>`. As someone who has worked with regular expressions longer than you've had hot meals, I suggest this as the method to build them up, using simple steps to piece together a regular expression that would normally daunt you. In fact, it's a pretty good method to build anything, come to think of it.

(On a side note, it is inconsistent that the `\` normally prevents a character from having special meaning, while with anchoring a word, it provides a special meaning for `< and >. I don't know why it was done this way, either.)

Regular expressions put a lot of power in your hands. Some of the command line's most powerful tools use them. Beginning with the next column, we will examine one of the most popular ones, `grep`.

***

[*] Actually, it represents any character except a newline character, but how often is a word split across different lines like that? That is a nitpicky little detail that only us pathological types who program in Perl worry about. We're funny that way.

Understanding The "grep" Command In Mac OS X

Part XV of this series...

October 4th, 2002

I don't know why Dudley keeps trying to find himself. I found him years ago.
- Peter Cook

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.
In the previous column, we learned about regular expressions, and how to use them to search for text in vi. Having such a text-searching tool for the command line would be a valuable addition to Unix; naturally, such a tool exists. It is called grep, and it is the subject of today’s column.

grep allows you to search through your entire system, for either the name of a file, or for content within those files. This is similar to the way Sherlock used to work before Sherlock 3, and the way “Find” works today in Jaguar’s GUI. When you need to find a string of text on your system from the command line, grep is the way to do it. Now, on to how to use it.

The grep command will take a regular expression, as well as a list of files. It will then search through the files and, for each line that is matched by the regular expression, print the line. (Supposedly, the name grep comes from ed command g/RE/p, or ”global/regular expression/print”, which does the same thing within the editor. I can neither confirm nor deny this.) If there are no files indicated, grep will read from standard input. Therefore, you can do things like:

```
[localhost:~] dr_unix% ls
Adam.txt   Library   Pictures  login     test_1.txt
Desktop    Movies    Public     personal  testfile
Documents  Music     Sites      temp.html who_list
[localhost:~] dr_unix% ls | grep ".es.*"
Desktop
Movies
Pictures
Sites
test_1.txt
testfile
```

to give a more flexible search. Notice that the regular expression, ".es.*", was enclosed in double quotes. Otherwise, we get this:

```
[localhost:~] dr_unix% ls | grep .es.*
grep: No match.
```

**Note:** I think that this is because the asterisk and/or period will confuse the tcsh command line, which tries to use them as metacharacters, so you need the quotes. On the other hand, if you want to anchor the regular expression to the end of a line with a dollar sign, it interprets this as a variable $ and chokes. tcsh is quirky with regular expressions, and I haven't quite figured out everything with it. I know from experience that the Korn shell, ksh, does not suffer from this. On the other hand, ksh is not the default shell, so there y'are.

You also need quotes if you have spaces in the regular expression. The difference between `grep the file` and `grep "the " file` is that the former will match any occurrence of t-h-e, whereas the latter will match only for t-h-e-space. This means that the former will match "I was there" but the latter will not. Remember that the command line ignores extra spaces, collapsing many into one, unless the spaces are quoted.

As you might expect, grep takes the standard regular expression characters of .. *, ^, $, \, and [ ]. Thus, to count the number of blank lines in a file, do:

```
[localhost:~] dr_unix% grep ^$ testfile

[localhost:~] dr_unix% grep ^$ testfile | wc -l       3
[localhost:~] dr_unix%
```

Thus, we can see that `grep ^$ testfile` will print all three blank lines. We can use `wc` and the pipe, |, to build our own tool to count blank lines. Neat, huh?

In some Unixes (Unices?), there were two versions of grep, grep and egrep, whose primary difference was that each had slightly different additions to the basic regular expression syntax. In Darwin, and therefore in OS X, the syntaxes (syntaxes?) are combined, and using either command will get you the same as using the other. Thus, you can bounce back and forth between them like so many yo-yos (yo-yi?)

One set of regular expression characters available in grep is the \{ \} pair. This allows you to search for a range of occurrences. Suppose you want to look for "to", followed by three to nine characters, followed by an "a". This can be done by:
Again, the quotes are needed here. If you want to match exactly 3, the regular expression is `to.{3}a`. Normally, the `{ }` pair is only available in `grep`, but in Darwin and OS X, it is also available in `egrep`.

We have seen that the asterisk (*) is used to denote "any number of the thing preceding me." In OS X's `grep`, the plus sign, +, can be used to denote "at least one of the thing preceding me." So, while the regular expression `th*e` will match `te, the, thhe, ...`, the regular expression `h+` will match the, thhe, thhhe, ... So can see that `h+` is the same as `hh*`. The plus sign is often used in other utilities' regular expressions, but is not part of `grep` on most other systems. Make a note of it, there will be a quiz later.

Another bonus freebie that is thrown our way is the question mark, ?, unless you are British and over 35, in which case it is "a mark of interrogation." `grep` uses this in regular expressions to denote "zero or one occurrence of the thing before me", or "an optional [whatever is before me]." Therefore, the expression `lie?d` will match either `lied` or `lid`.

Finally, the vertical bar, |, can be used for either/or matching, just like in, you guessed it, `vi`.

grep can take several options; you can see them all via `man grep`, of course, but I've found that the most useful ones are (remember that this works in the `grep` option format):

- `-c`: "count the lines". Instead of printing all the matched lines, `-c` merely prints a count of matched lines for each file. Thus that `! wc -l` trick isn't needed for one file. (If you pass in a list of files, though, ...)

- `-e PATTERN`: "expression starts here." Using `-e` will tell `grep"What follows is the pattern with which to search." This is very useful when your pattern starts with a `-`. Otherwise, the command line might think that your expression is an option and get confused.

- `-f FILE`: "file holds the expression". `-f` allows you to store a pattern in a file and tell `grep"Yo, use this." I've mostly used this when writing scripts that will use the same pattern repeatedly. That way, if I have to change it later, I only have to change it in one place.

- `-i`: "ignore case". `-i` forces `grep` to ignore the distinction between uppercase and lowercase. Imagine you need to find matches in a file which may have come from Windows (include shudder here). Now imagine a long string of paired letters like `[Tt][Hh][Ee] [Cc][Aa][Tt]` and on and on. Just use `-i` instead and save yourself time and pain.

- `-l`: "list files". Instead of printing the matched lines, when you use the `-l option, `grep` will just print a list of the files which contain the expression. This is mostly used when you are doing something like `grep "expression" *` in a directory with a lot of files or when you just want to know which files need (processing, editing, etc).

- `-n`: "number". `-n` means that before each line of output, `grep` will print its line number within the file.

- `-v": "invert". `-v` instructs `grep` to print only those lines that don't match the expression.

As you can see, `grep` is a very powerful tool. It can be used to quickly search files and to filter output on the command line. It does have a couple limitations, though. First, it is no speed demon. Building those regular expressions and parsing a lot of text in a flexible way takes resources, and it takes time. (Admittedly, these days, that isn't much of an issue, but still, there it is.) Second, consider the following: you are working away, happy as a clam, and the boss says "Cyprian", if your name is Cyprian, "I just got a call from marketing, we need to change the search in all those voodoo scripts you wrote, and we need it in ten minutes."

Now, you know and I know that you can look for the expression `the\.?.*ca\[r\]?t` and search for it using `\ after `\ after `\ But my lord, and your duke for that matter, who the heck would want to? Do you realize that you would look for `the\\\\n\\ca\[r\]?t` (or something along those lines) and heaven forbid you should make the slightest mistake. If you're like me, and I know I am, you'd think "Now dash it, there must be an easier way. Surely, in all the history of Unix, someone has had to face just such an emergency and written a grep-like tool to deal with this. Like that Cyprian chap, maybe." Well, Cyprian has come through. It's called `fgrep` (for "fast grep"), and it works a lot like `grep` except it doesn't take a regular expression.

```
fgrep [options] string [files]
```
Where you would normally place a regular expression, just put in a literal string. Originally it was used to be a fast alternative to `grep` by trading the power and flexibility of regular expressions for speed. As quick as computers are these days, that isn't an issue, but if you want to find something that contains a literal period or a literal asterisk, it's the bee's knees.

[*] This joke was borrowed at great embarrassment from Shelley Berman. All young whippersnappers are advised to ask their parents or grandparents.

---

Pico: An Easy To Use Command Line Editor

Part XVI of this series...  
November 1st, 2002

For every problem, there is an explanation that is simple, neat, and wrong.  
- H.L. Mencken, Prejudices

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

If you read this column regularly, you have my sympathy. But seriously, folks, you may remember the mild hoo-ha that resulted when I actually dared to "gasp" write about the `vi` editor. Like most holy wars, that involving your Unix editor choice often comes down to personal tastes and prejudices. Therefore, I tried to chose a middle path between something pretty complex (`emacs`) and something pretty simple (`pico`). Naturally it satisfied no one. Let this be a lesson, kids: by staying in the middle of the road, you're more likely to get hit by traffic.

However, as sometimes happens with shouts from the peanut gallery, someone yells out a good point. In this case, it may be that `vi` can be daunting for the inexperienced. And as I've said before, no one needs more daunts. Therefore, today's column is about a much simpler (though less powerful) editor, `pico`.

`pico` comes with Jaguar, so you don't need to download and install it. Just to be sure, check:

```
[localhost:~] dr_unix% which pico
/usr/bin/pico
```

First, a little background. `pico` was originally developed to be the editor of choice for `pine`, a simple, easy-to-use command-line mail program written at the University of Washington. (In Unix' grand tradition of nested and self-referential acronyms, `pine` allegedly stands for "pine is not elm", `elm` being another command-line mail tool.)

`pine` was to be used by people who aren't comfortable with a Unix command line: secretaries, data entry clerks, students, and moving down the evolutionary scale, tenured professors and university administrators. As such, `pine` had to be plain, simple, and straightforward. Likewise, the editor had to be just as plain, simple, and straightforward as `pine`. This is no small task, but the people at U-dub are to be commended for the results. `pine` is about the best command-line mail utility I've stumbled across, and the editor, `pico`, can be grasped easily by secretaries and data entry clerks. Even Perl programmers like yours truly can use it. (Alas, bureaucrats have trouble with it, and the students are rather hit and miss. Well, let's face it; some things just aren't possible.)

Oh, and don't bother looking for `pine` with the `which` command; it doesn't come with Jaguar.

Invoking `pico` is straightforward. Just type:

```
pico filename
```

So, if you've been following this column for a while, and still have the old file `testfile` in your home directory, type:

```
pico testfile
```

at the command line prompt. This should give you something along the lines of:

```
UW PICO(tm) 2.5                   File: testfile
Now is the time for all good men (and women, of course) to come to the aid of the party.
```

---

[90x53]Command Line 101  Page 46
We're out of ice cream.

This is a very long line, with more than the eighty characters that you will no$

Work is the curse of the drinking classes. - Oscar Wilde

So, what have we here? Remember, pico is designed for ease of use. If you want to add text, just start typing. There is no 'command mode' and 'insert mode' as in vi. Likewise, you can move around using the four arrow keys.

All of the pico editing commands are control-characters, and some of the most commonly used are listed along the bottom of the terminal window. For those new to these sorts of things, this just means that you hold down the control key (on the lower left or lower right of your keyboard) while you hit the key. Since there is no printable "control" character as such, it is represented by a caret, '^', followed by the letter. See the "^G Get Help" in the lower left of the screen? The "^G" denotes a control-G. Hold down the control key and hit a "G" (lower or upper case, it doesn't matter). This should take you to the help screen, thus:

UW PICO(tm) 2.5 File: testfile

Pico Help Text
with a layout very similar to the pine mailer. The status line at the top of the display shows pico’s version, the current file being edited and whether or not there are outstanding modifications that have not been saved. The third line from the bottom is used to report informational messages and for additional input.

The bottom two lines list the available editing commands.

Each character typed is automatically inserted into the buffer at the current cursor position. Editing commands and movement (besides arrow keys) are given to pico by typing special control-key sequences. A caret, '^', is used to denote the control key, sometimes marked "CTRL", so the CTRL-q
key combination is written as `Q.

The following functions are available in pico (where applicable, corresponding function key commands are in parentheses).

|^X Exit| Help | ^V Next| Pg
If you do a ^V to move down a page, you should see the start of a complete list of pico commands. One of the first things that will strike you is that pico has a lot fewer commands than vi. This makes pico easier to learn, though less powerful. (Everything comes at a price, you know.)

The commands I've found most useful are:

• The arrow keys, obviously.
• ^G: Get Help. Because I don't use pico all the time, I don't have all the commands at the tip of my brain (so-called).
• ^E: End of line. This moves the cursor to the end of the line. It saves a lot of arrowing around.
• ^A: Star of line. This moves the cursor to the start of the line. Ditto on the arrowing.
• ^V: Advance the cursor one page.
• ^Y: Move the cursor back one page.
• ^C: Refresh the screen. This is useful if the system throws warning messages to your terminal screen.
• ^W: "Where is". Search forward for text.
• ^K: Cut text. This removes the current line. If you used the ^^ (control-^) command to select text, then the selected text is removed.
• ^J: Justify text.
• ^U: Uncut (aka paste) the cut text. However, immediately after a Justify, it will unjustify instead. (More on this below.)
• ^R: Insert the contents of an external file at the current cursor position.
• ^O: Output file. Save the file without exiting pico.
• ^X: Exit. (You are asked if you want to save or not at this point.)

Most of these are straightforward. It should go without saying that the ^W command searches for plain, case-insensitive text, not for a regular expression; that's why I'm saying it. Again, it's simpler but not as powerful. Using ^J to justify text might take a bit of explanation. Recall the line:

This is a very long line, with more than the eighty characters that you will no$

In vi, the whole line was printed to the terminal but 'wrapped' around the end of the window. pico shows one line of text, however long, per line in the terminal window. The dollar sign ($) at the end is pico's way to tell you that the line is longer than the width of your window. If you want to see the whole thing you can just use the arrow keys or the ^W command to move along it. If you want to split it into a series of lines, and why wouldn't you, just tell pico to justify the text. Use the arrow keys to move the cursor to the line in question, then hit ^J and ... tada!

This is a very long line, with more than the eighty characters that you will normally see on the terminal. Notice how it wrapped around?

^J can also be used to piece together fragments of consecutive lines into justified text, which is nice for reformatting a paragraph which you've either added to or taken from.

Also, notice that a message has appeared just above the list of commands on the bottom two lines.

[ Can now UnJustify! ]

^G Get Help  ^O WriteOut  ^R Read File  ^Y Prev Pg  ^K Cut Text
^C Cur Pos  ^X Exit  ^J Justify  ^W Where is  ^V Next Pg  ^U UnJustify
^T To Spell

pico will generally print any such messages on this line. Notice that the ^U command is now "UnJustify", not "UnCut Text". This is one place where pico is inconsistent in commands. Upon issuing this Unjustify command, we get:

This is a very long line, with more than the eighty characters that you will no$

If you are really bored, you can do this all day. You have to be quick about it, though. After justifying the text, if you move your cursor off of the paragraph, ^U returns to "UnCut Text".

So there we have it, a simple, straightforward, easy-to-use command-line editor. In fact, almost the experienced geeks among you may have noticed that many things are in place for writing our own shell scripts. We're not there yet, though; in our next column, we'll look at editing your commands on the command line, using some of the tools we've seen so far.
Command Line History & Editing Your Commands

Part XVII of this series...
November 22nd, 2002

Shome mistake, sharely?
- catchphrase of Private Eye magazine

This series is designed to help you learn more about the Mac OS X command line. If you have any questions about what you read here, check out the earlier columns, write back in the comments below, or join us in the Hardcore X! forum.

If you’ve been following this column, not only are you a person of impeccable taste, you’re also likely to have typed a mistake on the command line. That’s nothing to be ashamed of, we’re only human. (Well, you are.) You’ve probably miss-typed a command and been faced with retying the whole command again. You might be able to do a copy-and-paste, if the previous command is still on the screen. However, that is not always possible. And if you’ll recall, Unix is more than a decade older than the first available GUI (the one on the Apple Lisa); with no mouse, doing a copy-and-paste is not so intuitive. What’s needed is a way to move back and forth through a list of previous commands.

You will not be surprised to read that this is possible. And if you know Unix, you will not be surprised to read that there is more than one way to do this.

tcsh provides a very simple mechanism to do this using the arrow keys. It’s so simple, in fact, that even a rocket scientist can grasp it. (And I oughta know ...)

Memories...

Open a terminal window, and type in a few commands: ls, pwd, who, whatever floats your boat. You can now move up and down the list of commands you just typed using the up and down arrow keys. You can also, shock of shocks, move the cursor back and forth across a command with the left and right arrows. It’s very similar to moving around pico, the simple editor I discussed in the previous column. It is not pico, though; the pico commands (the CONTROL- characters) don’t do the same thing. Sometimes they do nothing, sometimes they cause some goofy behavior, so until you are ready for more than the arrow keys, it’s best to avoid them. For the morbidly curious, you can find out what they are by typing bindkey. Make sure you pipe it through the more command, it takes several screens to list them all. You might also want to keep a cool, damp cloth for your forehead within easy reach.

Auto completion is a life saver

One other feature is command completion, which is related to command line editing. Sorta. In your terminal window, go to your home directory:

```bash
[localhost:~] dr_unix% cd
[localhost:~] dr_unix%
```

Now let's look at the directory's list of files, using the -F flag of ls so that directories are denoted by a trailing '/':

```bash
[localhost:~] dr_unix% ls -F
Adam.txt  Library/  Public/  personal/  testfile
Desktop/  Library/  Music/  Sites/  temp.html  who_list
Documents/  Pictures/  login  test_1.txt
[localhost:~] dr_unix%
```

Better, use our new tool fgrep to filter out those files which are not directories:

```bash
[localhost:~] dr_unix% ls -F | fgrep '/'
Desktop/
Documents/
Library/
Music/
Pictures/
Public/
Sites/
personal/
[localhost:~] dr_unix%
```

Say you want to change your default directory to Library. If you are lazy and/or a bad typist, and who isn't, you don't have to type the whole command:

```bash
[localhost:~] dr_unix% cd Library
[localhost:~/] dr_unix% pwd
/Users/dr_unix/Library
[localhost:~/] dr_unix%
```
Instead, you can simply type (without hitting RETURN):

```
[localhost:~] dr_unix% cd Lib
```

If you **now** hit TAB, you get:

```
[localhost:~] dr_unix% cd Library/
```

Since there is only one file in the directory that starts "Lib", `tcsh` will 'complete' the command when you hit the TAB.

```
[localhost:~] dr_unix% cd Library/
[localhost:~/Library] dr_unix% pwd
/Users/dr_unix/Library
[localhost:~/Library] dr_unix%
```

Neat, huh?

Get it on, bang a gong...

Now, most of us aren't complete geeks (and some who are won't admit it), so this will take a lot of us a long way. However, there are more powerful forms of command line editing available. Go to your terminal window, and type the `ls` command:

```
[localhost:~] dr_unix% ls
Adam.txt   Documents  Music      Public     login      temp.html
testfile   Desktop    Library    Pictures   Sites      personal   test_1.txt
who_list
[localhost:~] dr_unix%
```

Now, type `!!` and hit return:

```
[localhost:~] dr_unix% ls
Adam.txt   Documents  Music      Public     login      temp.html
testfile   Desktop    Library    Pictures   Sites      personal   test_1.txt
who_list
[localhost:~] dr_unix% !!
ls
Adam.txt   Documents  Music      Public     login      temp.html
testfile   Desktop    Library    Pictures   Sites      personal   test_1.txt
who_list
[localhost:~] dr_unix%
```

The `!!` command will run the previous command. Notice how, after you hit RETURN, it showed what the command was before it executed it? If you keep typing `!!` over and over again, you can run the `ls` command from here to Goshen. In fact, `tcsh` is clever enough that you can add to the previous command:

```
[localhost:~] dr_unix% ls
Adam.txt   Documents  Music      Public     login      temp.html
testfile   Desktop    Library    Pictures   Sites      personal   test_1.txt
who_list
[localhost:~] dr_unix% !! | grep t
ls | grep t
Adam.txt
Desktop
Documents
Pictures
Sites
temp.html
test_1.txt
testfile
who_list
[localhost:~] dr_unix%
```

**History**

What's behind all this is `tcsh`'s built-in history commands. Each terminal keeps track of a list of the previously run commands, storing up to the number denoted by the "history" variable:
Of course, history being just a variable, you can set it to whatever you want, viz:

```
[localhost:~] dr_unix% set history=20
[localhost:~] dr_unix% echo $history
20
```

If you want to see the list of commands in your history, just type `history`:

```
[localhost:~] dr_unix% history
  8 19:39   cd Library
  9 19:39   pwd
 10 19:49   cd ..
 11 19:59   cd Library/
 12 19:59   pwd
 13 20:01   cd
 14 20:01   echo $history
 15 20:01   ls -a
 16 20:01   more login
 17 20:01   w
 18 20:02   ps
 19 20:03   clear
 20 20:03   ls
 21 20:04   ls
 22 20:12   ls
 23 20:12   ls -a
 24 20:19   echo $history
 25 20:20   set history=20
 26 20:20   echo $history
 27 20:23   history
```

Notice that this gives you a list of commands, each with an event number and a timestamp. You can also print just the last $n$ commands by giving `history` an argument:

```
[localhost:~] dr_unix% history 5
  24 20:19   echo $history
  25 20:20   set history=20
  26 20:20   echo $history
  27 20:23   history
  28 20:26   history 5
```

`!!` is a special/default case. If you want to rerun a specific command, you simply type `!n`, where $n$ is the event number (the sequential number that is at the beginning of each line):

```
[localhost:~] dr_unix% !25
  24 20:19   echo $history
set history=20
```

If you want to repeat the previous command which started with a string, just type `!` followed by the string:

```
[localhost:~] dr_unix% !history 5
```

(And yes, I did sneak another command in there on you.) This is useful, but not always what you want. Often, you will be using several commands in succession on a particular file, and the name of which will not be the first part of the command. Let's say your history file looks something like:

```
54 12:45   cd Documents/novels/
55 12:45   ls
56 12:46   vi big_fish_little_blonde.txt
57 12:49   nethack
58 14:32   ls -l big_fish_little_blonde.txt
59 14:32   wc -l big_fish_little_blonde.txt
60 14:32   vi big_fish_little_blonde.txt
61 14:33   nethack
62 16:09   pwd
```

(If that looks silly, that's because it's based on two inside jokes, and darned obscure ones, known to some people in our forums.) If you want to repeat the last command on the file, put 's around the string:

```bash
[localhost:~] dr_unix% !?big_fish?
```
That will take you back into vi, editing the file so named

### Multiple histories

If you open several terminals at once and fool around with each one's command history, you will notice that each terminal's command history is kept separate from the others. However, if you want to open another terminal later and have it know the history you now have, this can be done. (Didn't you just know it?)

If you set the `savehist` shell variable (by simply typing `set savehist`), the shell will save its history list to the file `~/.history` when you log out. The next time you log in, this file is read and the list is placed into the new shell's history list. If `savehist` is not given an argument, the entire list is saved to `~/.history`; if you give it a value (which should be less than the value of `$history`), it will save that many of the most recent commands.

### More useless command stuff

There are more things in manipulating command history, but I have found them useful in rare cases, if ever. Given all that there is and how rarely you use it, the remaining commands are ignorable. But if you just *half* to know them all, a good book on csh and/or tcsh should list those commands.

### Editing Your Commands

As I've mentioned before, Unix has a couple of text editors that are powerful. What's more, they inspire great loyalty or derision, depending on your prejudices. Now, given the nature of Unix, or rather Unix developers, you would expect that someone could choose to use vi or emacs commands to edit your commands, search through the history lists, etc. What else would you expect from such charming rogues (vi) and rakish chaps (emacs), plus the occasional handsome brute?

Well, you can, and it is done with the `bindkey` command. `bindkey` will let you set your editing commands to either emacs (bindkey -e) or to vi (bindkey -v). For now, we will look at the vi version, for two reasons: vi has already been covered in this series, and the emacs version doesn't require any explanation, once you know emacs.

As I'm sure you'll recall, vi has a command mode and an input mode. To enter the input mode, you need to enter one of several commands (i, a, R, etc.). To return to command mode requires hitting the ESCAPE key. Now, if you think about it, normally you just want to start typing commands and not have to worry about editing the commands. It would be pretty silly to have to start each command by typing an 'i' to enter input mode. Therefore, when you set your key binding to vi, you are placed into input mode automatically after each command. To enter command mode, you must hit the ESCAPE key.

Let's try this, shall we? Open a new terminal session, and set your key bindings to vi mode.

```bash
[localhost:~] dr_unix% bindkey -v
[localhost:~] dr_unix%
```
(Those of you who know emacs are welcome to set your key bindings with a -e.)

Now type a few familiar commands:

```bash
[localhost:~] dr_unix% pwd
/Users/dr_unix
[localhost:~] dr_unix% w
3:50PM  up  1:37, 3 users, load averages: 0.60, 0.43, 0.26
USER   TTY   FROM     LOGIN@  IDLE  WHAT
dr_unix  co   2:15PM  1:37  -
dr_unix  p1   2:20PM  0    -
dr_unix  p2   3:39PM  0    -
[localhost:~] dr_unix% date
Sat Oct 26 15:50:09 EST 2002
[localhost:~] dr_unix%
```
Now hit ESCAPE to enter command mode. You can move up and down your history list using vi's 'j' and 'k' commands. You can also move within a command using the movement commands like 'w', 'E', and so on. This includes the '/' command to search for a command which
matches a regular expression. And you can use any vi command to edit that command. As a plus, tcsh lets you use the arrow keys at the same time.

Congratulations, you are now well on your way to being a Unix shell power user. Next, we'll look at handling multiple processes.