```
TERM=adm3
```

to your `.profile` file.

It is also possible to use variables for abbreviation. If you find yourself frequently referring to some directory with a long name, it might be worthwhile adding a line like

d=/horribly/long/directory/name

to your profile, so that you can say things like

```
$ cd $d
```

Personal variables like d are conventionally spelled in lower case to distinguish them from those used by the shell itself, like PATH.

Finally, it's necessary to tell the shell that you intend to use the variables in other programs; this is done with the command export, to which we will return in Chapter 3:

```
export MAIL PATH TERM
```

To summarize, here is what a typical `.profile` file might look like:

```
$ cat .profile
stty erase '^h' -tabs
MAIL=/usr/spool/mail/you
PATH=:$HOME/bin:/bin:/usr/bin:/usr/games
TERM=adm3
b=$HOME/book
export MAIL PATH TERM b
date
who | wc -l
$
```

We have by no means exhausted the services that the shell provides. One of the most useful is that you can create your own commands by packaging existing commands into a file to be processed by the shell. It is remarkable how much can be achieved by this fundamentally simple mechanism. Our discussion of it begins in Chapter 3.

1.5 The rest of the UNIX system

There's much more to the UNIX system than we've addressed in this chapter, but then there's much more to this book. By now, you should feel comfortable with the system and, particularly, with the manual. When you have specific questions about when or how to use commands, the manual is the place to look.

It is also worth browsing in the manual occasionally, to refresh your knowledge of familiar commands and to discover new ones. The manual describes many programs we won't illustrate, including compilers for languages

like FORTRAN 77; calculator programs such as bc(1); cu(1) and uucp(1) for inter-machine communication; graphics packages; statistics programs; and esoterica such as `units`(1).

As we've said before, this book does not replace the manual, it supplements it. In the chapters that follow we will look at pieces and programs of the UNIX system, starting from the information in the manual but following the threads that connect the components. Although the program interrelationships are never made explicit in the manual, they form the fabric of the UNIX programming environment.

History and bibliographic notes

The original UNIX paper is by D. M. Ritchie and K. L. Thompson: "The UNIX Time-sharing System," *Communications of the ACM*, July, 1974, and reprinted in *CACM*, January, 1983. (Page 89 of the reprint is in the March 1983 issue.) This overview of the system for people interested in operating systems is worth reading by anyone who programs.

The Bell System Technical Journal (*BSTJ*) special issue on the UNIX system (July, 1978) contains many papers describing subsequent developments, and some retrospective material, including an update of the original *CACM* paper by Ritchie and Thompson. A second special issue of the BSTJ, containing new UNIX papers, is scheduled to be published in 1984.

"The UNIX Programming Environment," by B. W. Kernighan and J. R. Mashey (*IEEE Computer Magazine*, April, 1981), attempts to convey the essential features of the system for programmers.

The *UNIX Programmer's Manual*, in whatever version is appropriate for your system, lists commands, system routines and interfaces, file formats, and maintenance procedures. You can't live without this for long, although you will probably only need to read parts of Volume 1 until you start programming. Volume 1 of the 7th Edition manual is published by Holt, Rinehart and Winston.

Volume 2 of the *UNIX Programmer's Manual* is called "Documents for Use with the UNIX Time-sharing System" and contains tutorials and reference manuals for major commands. In particular, it describes document preparation programs and program development tools at some length. You will want to read most of this eventually.

A UNIX Primer, by Ann and Nico Lomuto (Prentice-Hall, 1983), is a good introduction for raw beginners, especially non-programmers.

```
$ pr ch* | lpr &
6951                Process-id of lpr
$
```

the processes in it are all started at once — the & applies to the whole pipeline. Only one process-id is printed, however, for the last process in the sequence.

The command

```
$ wait
```

waits until all processes initiated with & have finished. If it doesn't return immediately, you have commands still running. You can interrupt wait with DELETE.

You can use the process-id printed by the shell to stop a process initiated with &:

```
$ kill 6944
```

If you forget the process-id, you can use the command ps to tell you about everything you have running. If you are desperate, kill 0 will kill all your processes except your login shell. And if you're curious about what other users are doing, ps -ag will tell you about *all* processes that are currently running. Here is some sample output:

```
$ ps -ag
  PID TTY TIME CMD
   36 co  6:29 /etc/cron
 6423 5   0:02 -sh
 6704 1   0:04 -sh
 6722 1   0:12 vi paper
 4430 2   0:03 -sh
 6612 7   0:03 -sh
 6628 7   1:13 rogue
 6843 2   0:02 write dmr
 6949 4   0:01 login bimmler
 6952 5   0:08 pr ch1.1 ch1.2 ch1.3 ch1.4
 6951 5   0:03 lpr
 6959 5   0:02 ps -ag
 6844 1   0:02 write rob
$
```

PID is the process-id; TTY is the terminal associated with the process (as in who); TIME is the processor time used in minutes and seconds; and the rest is the command being run. ps is one of those commands that is different on different versions of the system, so your output may not be formatted like this. Even the arguments may be different — see the manual page ps(1).

Processes have the same sort of hierarchical structure that files do: each process has a parent, and may well have children. Your shell was created by a process associated with whatever terminal line connects you to the system. As

you run commands, those processes are the direct children of your shell. If you run a program from within one of those, for example with the ! command to escape from ed, that creates its own child process which is thus a grandchild of the shell.

Sometimes a process takes so long that you would like to start it running, then turn off the terminal and go home without waiting for it to finish. But if you turn off your terminal or break your connection, the process will normally be killed even if you used &. The command nohup ("no hangup") was created to deal with this situation: if you say

```
$ nohup command &
```

the command will continue to run if you log out. Any output from the command is saved in a file called nohup.out. There is no way to nohup a command retroactively.

If your process will take a lot of processor resources, it is kind to those who share your system to run your job with lower than normal priority; this is done by another program called nice:

```
$ nice expensive-command &
```

nohup automatically calls nice, because if you're going to log out you can afford to have the command take a little longer.

Finally, you can simply tell the system to start your process at some wee hour of the morning when normal people are asleep, not computing. The command is called at(1):

```
$ at time
whatever commands
you want ...
ctl-d
$
```

This is the typical usage, but of course the commands could come from a file:

```
$ at 3am <file
$
```

Times can be written in 24-hour style like 2130, or 12-hour style like 930pm.

Tailoring the environment

One of the virtues of the UNIX system is that there are several ways to bring it closer to your personal taste or the conventions of your local computing environment. For example, we mentioned earlier the problem of different standards for the erase and line kill characters, which by default are usually # and @. You can change these any time you want with

```
$ stty erase e kill k
```

where *e* is whatever character you want for erase and *k* is for line kill. But it's

Given the capability of redirecting output with >, it becomes possible to combine commands to achieve effects not possible otherwise. For example, to print an alphabetical list of users,

```
$ who >temp
$ sort <temp
```

Since who prints one line of output per logged-on user, and wc -l counts lines (suppressing the word and character counts), you can count users with

```
$ who >temp
$ wc -l <temp
```

You can count the files in the current directory with

```
$ ls >temp
$ wc -l <temp
```

though this includes the filename temp itself in the count. You can print the filenames in three columns with

```
$ ls >temp
$ pr -3 <temp
```

And you can see if a particular user is logged on by combining who and grep:

```
$ who >temp
$ grep mary <temp
```

In all of these examples, as with filename pattern characters like *, it's important to remember that the interpretation of > and < is being done by the shell, not by the individual programs. Centralizing the facility in the shell means that input and output redirection can be used with any program; the program itself isn't aware that something unusual has happened.

This brings up an important convention. The command

```
$ sort <temp
```

sorts the contents of the file temp, as does

```
$ sort temp
```

but there is a difference. Because the string <temp is interpreted by the shell, sort does not see the filename temp as an argument; it instead sorts its *standard input*, which the shell has redirected so it comes from the file. The latter example, however, passes the name temp as an argument to sort, which reads the file and sorts it. sort can be given a list of filenames, as in

```
$ sort temp1 temp2 temp3
```

but if no filenames are given, it sorts its standard input. This is an essential property of most commands: if no filenames are specified, the standard input is processed. This means that you can simply type at commands to see how they

work. For example,

```
$ sort
ghi
abc
def
ctl-d
abc
def
ghi
$
```

In the next section, we will see how this principle is exploited.

Exercise 1-5. Explain why

```
$ ls >ls.out
```

causes ls.out to be included in the list of names. □

Exercise 1-6. Explain the output from

```
$ wc temp >temp
```

If you misspell a command name, as in

```
$ woh >temp
```

what happens? □

Pipes

All of the examples at the end of the previous section rely on the same trick: putting the output of one program into the input of another via a temporary file. But the temporary file has no other purpose; indeed, it's clumsy to have to use such a file. This observation leads to one of the fundamental contributions of the UNIX system, the idea of a *pipe*. A pipe is a way to connect the output of one program to the input of another program without any temporary file; a *pipeline* is a connection of two or more programs through pipes.

Let us revise some of the earlier examples to use pipes instead of temporaries. The vertical bar character | tells the shell to set up a pipeline:

```
$ who | sort            Print sorted list of users
$ who | wc -l           Count users
$ ls | wc -l            Count files
$ ls | pr -3            3-column list of filenames
$ who | grep mary       Look for particular user
```

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can write to a pipe. This is where the convention of reading the standard input when no files are named pays off: any program that adheres to the convention can be used in pipelines. grep, pr, sort and wc are all used that way in the pipelines above.

You can have as many programs in a pipeline as you wish:

all by itself will take you back to your home directory, the directory where you log in.

Once your book is published, you can clean up the files. To remove the directory book, remove all the files in it (we'll show a fast way shortly), then cd to the parent directory of book and type

```
$ rmdir book
```

rmdir will only remove an empty directory.

1.4 The shell

When the system prints the prompt $ and you type commands that get executed, it's not the kernel that is talking to you, but a go-between called the command interpreter or *shell*. The shell is just an ordinary program like date or who, although it can do some remarkable things. The fact that the shell sits between you and the facilities of the kernel has real benefits, some of which we'll talk about here. There are three main ones:

- Filename shorthands: you can pick up a whole set of filenames as arguments to a program by specifying a pattern for the names — the shell will find the filenames that match your pattern.
- Input-output redirection: you can arrange for the output of any program to go into a file instead of onto the terminal, and for the input to come from a file instead of the terminal. Input and output can even be connected to other programs.
- Personalizing the environment: you can define your own commands and shorthands.

Filename shorthand

Let's begin with filename patterns. Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Physically it should be divided too, because it is cumbersome to edit large files. Thus you should type the document as a number of files. You might have separate files for each chapter, called ch1, ch2, etc. Or, if each chapter were broken into sections, you might create files called

```
ch1.1
ch1.2
ch1.3
...
ch2.1
ch2.2
...
```

which is the organization we used for this book. With a systematic naming convention, you can tell at a glance where a particular file fits into the whole.

What if you want to print the whole book? You could say


```
$ pr ch1.1 ch1.2 ch1.3 ...
```

but you would soon get bored typing filenames and start to make mistakes. This is where filename shorthand comes in. If you say

```
$ pr ch*
```

the shell takes the * to mean "any string of characters," so ch* is a pattern that matches all filenames in the current directory that begin with ch. The shell creates the list, in alphabetical† order, and passes the list to pr. The pr command never sees the *; the pattern match that the shell does in the current directory generates a list of strings that are passed to pr.

The crucial point is that filename shorthand is not a property of the pr command, but a service of the shell. Thus you can use it to generate a sequence of filenames for *any* command. For example, to count the words in the first chapter:

```
$ wc ch1.*
    113     562    3200 ch1.0
    935    4081   22435 ch1.1
    974    4191   22756 ch1.2
    378    1561    8481 ch1.3
   1293    5298   28841 ch1.4
     33     194    1190 ch1.5
     75     323    2030 ch1.6
   3801   16210   88933 total
$
```

There is a program called echo that is especially valuable for experimenting with the meaning of the shorthand characters. As you might guess, echo does nothing more than echo its arguments:

```
$ echo hello world
hello world
$
```

But the arguments can be generated by pattern-matching:

```
$ echo ch1.*
```

lists the names of all the files in Chapter 1,

```
$ echo *
```

lists *all* the filenames in the current directory in alphabetical order,

```
$ pr *
```

prints all your files (in alphabetical order), and

† Again, the order is not strictly alphabetical, in that upper case letters come before lower case letters. See ascii(7) for the ordering of the characters used in the sort.

Table 1.1: Common File System Commands

`ls`	list names of all files in current directory
`ls` *filenames*	list only the named files
`ls -t`	list in time order, most recent first.
`ls -l`	list long: more information: also `ls -lt`
`ls -u`	list by time last used; also `ls -lu`, `ls -lut`
`ls -r`	list in reverse order; also `-rt`, `-rlt`, etc.
`ed` *filename*	edit named file
`cp` *file1 file2*	copy *file1* to *file2*, overwrite old *file2* if it exists
`mv` *file1 file2*	move *file1* to *file2*, overwrite old *file2* if it exists
`rm` *filenames*	remove named files, irrevocably
`cat` *filenames*	print contents of named files
`pr` *filenames*	print contents with header, 66 lines per page
`pr -n` *filenames*	print in *n* columns
`pr -m` *filenames*	print named files side by side (multiple columns)
`wc` *filenames*	count lines, words and characters for each file
`wc -l` *filenames*	count lines for each file
`grep` *pattern filenames*	print lines matching *pattern*
`grep -v` *pattern files*	print lines not matching *pattern*
`sort` *filenames*	sort files alphabetically by line
`tail` *filename*	print last 10 lines of file
`tail -n` *filename*	print last *n* lines of file
`tail +n` *filename*	start printing file at line *n*
`cmp` *file1 file2*	print location of first difference
`diff` *file1 file2*	print all differences between files

```
$ pwd
/usr/you
$
```

This says that you are currently in the directory `you`, in the directory `usr`, which in turn is in the *root directory*, which is conventionally called just '/'. The / characters separate the components of the name; the limit of 14 characters mentioned above applies to each component of such a name. On many systems, `/usr` is a directory that contains the directories of all the normal users of the system. (Even if your home directory is not `/usr/you`, `pwd` will print something analogous, so you should be able to follow what happens below.)

If you now type


```
$ ls /usr/you
```

you should get exactly the same list of file names as you get from a plain `ls`. When no arguments are provided, `ls` lists the contents of the current directory; given the name of a directory, it lists the contents of that directory.

Next, try

```
$ ls /usr
```

This should print a long series of names, among which is your own login directory `you`.

The next step is to try listing the root itself. You should get a response similar to this:

```
$ ls /
bin
boot
dev
etc
lib
tmp
unix
usr
$
```

(Don't be confused by the two meanings of /: it's both the name of the root and a separator in filenames.) Most of these are directories, but `unix` is actually a file containing the executable form of the UNIX kernel. More on this in Chapter 2.

Now try

```
$ cat /usr/you/junk
```

(if `junk` is still in your directory). The name

```
/usr/you/junk
```

is called the *pathname* of the file. "Pathname" has an intuitive meaning: it represents the full name of the path from the root through the tree of directories to a particular file. It is a universal rule in the UNIX system that wherever you can use an ordinary filename, you can use a pathname.

The file system is structured like a genealogical tree; here is a picture that may make it clearer.

To make the discussion concrete, we'll use a file called poem that contains a familiar verse by Augustus De Morgan. Let's create it with ed:

```
$ ed
a
Great fleas have little fleas
  upon their backs to bite 'em,
And little fleas have lesser fleas,
  and so ad infinitum.
And the great fleas themselves, in turn,
  have greater fleas to go on;
While these again have greater still,
  and greater still, and so on.
.
w poem
263
q
$
```

The first command counts the lines, words and characters in one or more files; it is named wc after its word-counting function:

```
$ wc poem
      8     46    263 poem
$
```

That is, poem has 8 lines, 46 words, and 263 characters. The definition of a "word" is very simple: any string of characters that doesn't contain a blank, tab or newline.

wc will count more than one file for you (and print the totals), and it will also suppress any of the counts if requested. See wc(1).

The second command is called grep; it searches files for lines that match a pattern. (The name comes from the ed command g/*regular-expression*/p, which is explained in Appendix 1.) Suppose you want to look for the word "fleas" in poem:

```
$ grep fleas poem
Great fleas have little fleas
And little fleas have lesser fleas,
And the great fleas themselves, in turn,
  have greater fleas to go on;
$
```

grep will also look for lines that *don't* match the pattern, when the option -v is used. (It's named 'v' after the editor command; you can think of it as inverting the sense of the match.)


```
$ grep -v fleas poem
  upon their backs to bite 'em,
  and so ad infinitum.
While these again have greater still,
  and greater still, and so on.
$
```

grep can be used to search several files; in that case it will prefix the filename to each line that matches, so you can tell where the match took place. There are also options for counting, numbering, and so on. grep will also handle much more complicated patterns than just words like "fleas," but we will defer consideration of that until Chapter 4.

The third command is sort, which sorts its input into alphabetical order line by line. This isn't very interesting for the poem, but let's do it anyway, just to see what it looks like:

```
$ sort poem
  and greater still, and so on.
  and so ad infinitum.
  have greater fleas to go on;
  upon their backs to bite 'em,
And little fleas have lesser fleas,
And the great fleas themselves, in turn,
Great fleas have little fleas
While these again have greater still,
$
```

The sorting is line by line, but the default sorting order puts blanks first, then upper case letters, then lower case, so it's not strictly alphabetical.

sort has zillions of options to control the order of sorting — reverse order, numerical order, dictionary order, ignoring leading blanks, sorting on fields within the line, etc. — but usually one has to look up those options to be sure of them. Here are a handful of the most common:

sort -r	*Reverse normal order*
sort -n	*Sort in numeric order*
sort -nr	*Sort in reverse numeric order*
sort -f	*Fold upper and lower case together*
sort +*n*	*Sort starting at n+1-st field*

Chapter 4 has more information about sort.

Another file-examining command is tail, which prints the last 10 lines of a file. That's overkill for our eight-line poem, but it's good for larger files. Furthermore, tail has an option to specify the number of lines, so to print the last line of poem:

"`total 2`" tells how many blocks of disc space the files occupy; a block is usually either 512 or 1024 characters. The string `-rw-r--r--` tells who has permission to read and write the file; in this case, the owner (`you`) can read and write, but others can only read it. The "`1`" that follows is the number of links to the file; ignore it until Chapter 2. "`you`" is the owner of the file, that is, the person who created it. 19 and 22 are the number of characters in the corresponding files, which agree with the numbers you got from `ed`. The date and time tell when the file was last changed.

Options can be grouped: `ls -lt` gives the same data as `ls -l`, but sorted with most recent files first. The `-u` option gives information on when files were used: `ls -lut` gives a long (`-l`) listing in the order of most recent use. The option `-r` reverses the order of the output, so `ls -rt` lists in order of least recent use. You can also name the files you're interested in, and `ls` will list the information about them only:

```
$ ls -l junk
-rw-r--r-- 1 you        19 Sep 26 16:25 junk
$
```

The strings that follow the program name on the command line, such as `-l` and `junk` in the example above, are called the program's *arguments*. Arguments are usually options or names of files to be used by the command.

Specifying options by a minus sign and a single letter, such as `-t` or the combined `-lt`, is a common convention. In general, if a command accepts such optional arguments, they precede any filename arguments, but may otherwise appear in any order. But UNIX programs are capricious in their treatment of multiple options. For example, standard 7th Edition `ls` won't accept

```
$ ls -l -t                    Doesn't work in 7th Edition
```

as a synonym for `ls -lt`, while other programs *require* multiple options to be separated.

As you learn more, you will find that there is little regularity or system to optional arguments. Each command has its own idiosyncrasies, and its own choices of what letter means what (often different from the same function in other commands). This unpredictable behavior is disconcerting and is often cited as a major flaw of the system. Although the situation is improving — new versions often have more uniformity — all we can suggest is that you try to do better when you write your own programs, and in the meantime keep a copy of the manual handy.

Printing files — `cat` *and* `pr`

Now that you have some files, how do you look at their contents? There are many programs to do that, probably more than are needed. One possibility is to use the editor:


```
$ ed junk
19                         ed reports 19 characters in junk
1,$p                       Print lines 1 through last
To be or not to be         File has only one line
q                          All done
$
```

`ed` begins by reporting the number of characters in `junk`; the command `1,$p` tells it to print all the lines in the file. After you learn how to use the editor, you can be selective about the parts you print.

There are times when it's not feasible to use an editor for printing. For example, there is a limit — several thousand lines — on how big a file `ed` can handle. Furthermore, it will only print one file at a time, and sometimes you want to print several, one after another without pausing. So here are a couple of alternatives.

First is `cat`, the simplest of all the printing commands. `cat` prints the *contents* of all the files named by its arguments:

```
$ cat junk
To be or not to be
$ cat temp
That is the question.
$ cat junk temp
To be or not to be
That is the question.
$
```

The named file or files are catenated† (hence the name "`cat`") onto the terminal one after another with nothing between.

There's no problem with short files, but for long ones, if you have a high-speed connection to your computer, you have to be quick with *ctl-s* to stop output from `cat` before it flows off your screen. There is no "standard" command to print a file on a video terminal one screenful at a time, though almost every UNIX system has one. Your system might have one called `pg` or `more`. Ours is called `p`; we'll show you its implementation in Chapter 6.

Like `cat`, the command `pr` prints the contents of all the files named in a list, but in a form suitable for line printers: every page is 66 lines (11 inches) long, with the date and time that the file was changed, the page number, and the filename at the top of each page, and extra lines to skip over the fold in the paper. Thus, to print `junk` neatly, then skip to the top of a new page and print `temp` neatly:

† "Catenate" is a slightly obscure synonym for "concatenate."

Mary's terminal:	*Your terminal:*
`$ write you`	
	`$ Message from mary tty7...`
	`write mary`
`Message from you ttya...`	
`did you forget lunch? (o)`	
	`did you forget lunch? (o)`
	`five@`
	`ten minutes (o)`
`ten minutes (o)`	
`ok (oo)`	
	`ok (oo)`
	ctl-d
`EOF`	
ctl-d	
	`$ EOF`
`$`	

You can also exit from `write` by pressing DELETE. Notice that your typing errors do not appear on Mary's terminal.

If you try to write to someone who isn't logged in, or who doesn't want to be disturbed, you'll be told. If the target is logged in but doesn't answer after a decent interval, the person may be busy or away from the terminal; simply type *ctl*-d or DELETE. If *you* don't want to be disturbed, use `mesg(1)`.

News

Many UNIX systems provide a news service, to keep users abreast of interesting and not so interesting events. Try typing

    $ news

There is also a large network of UNIX systems that keep in touch through telephone calls; ask a local expert about `netnews` and USENET.

The manual

The *UNIX Programmer's Manual* describes most of what you need to know about the system. Section 1 deals with commands, including those we discuss in this chapter. Section 2 describes the system calls, the subject of Chapter 7, and Section 6 has information about games. The remaining sections talk about functions for use by C programmers, file formats, and system maintenance. (The numbering of these sections varies from system to system.) Don't forget the permuted index at the beginning; you can skim it quickly for commands that might be relevant to what you want to do. There is also an introduction to the system that gives an overview of how things work.

Often the manual is kept on-line so that you can read it on your terminal. If you get stuck on something, and can't find an expert to help, you can print any manual page on your terminal with the command `man` *command-name*.

Thus to read about the `who` command, type

    $ man who

and, of course,

    $ man man

tells about the `man` command.

Computer-aided instruction

Your system may have a command called `learn`, which provides computer-aided instruction on the file system and basic commands, the editor, document preparation, and even C programming. Try

    $ learn

If `learn` exists on your system, it will tell you what to do from there. If that fails, you might also try `teach`.

Games

It's not always admitted officially, but one of the best ways to get comfortable with a computer and a terminal is to play games. The UNIX system comes with a modest supply of games, often supplemented locally. Ask around, or see Section 6 of the manual.

1.2 Day-to-day use: files and common commands

Information in a UNIX system is stored in *files*, which are much like ordinary office files. Each file has a name, contents, a place to keep it, and some administrative information such as who owns it and how big it is. A file might contain a letter, or a list of names and addresses, or the source statements of a program, or data to be used by a program, or even programs in their executable form and other non-textual material.

The UNIX file system is organized so you can maintain your own personal files without interfering with files belonging to other people, and keep people from interfering with you too. There are myriad programs that manipulate files, but for now, we will look at only the more frequently used ones. Chapter 2 contains a systematic discussion of the file system, and introduces many of the other file-related commands.

Creating files — the editor

If you want to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with a *text editor*, which is a program for storing and manipulating information in the computer. Almost every UNIX system has a *screen editor*, an editor that takes advantage of modern terminals to display the effects of your editing changes in context as you make them. Two of the most popular are `vi` and `emacs`. We

Mistakes in typing

If you make a typing mistake, and see it before you have pressed RETURN, there are two ways to recover: *erase* characters one at a time or *kill* the whole line and re-type it.

If you type the *line kill* character, by default an at-sign @, it causes the whole line to be discarded, just as if you'd never typed it, and starts you over on a new line:

```
$ ddtae@                     Completely botched; start over
date                           on a new line
Mon Sep 26 12:23:39 EDT 1983
$
```

The sharp character # erases the last character typed; each # erases one more character, back to the beginning of the line (but not beyond). So if you type badly, you can correct as you go:

```
$ dd#atte##e                 Fix it as you go
Mon Sep 26 12:24:02 EDT 1983
$
```

The particular erase and line kill characters are *very* system dependent. On many systems (including the one we use), the erase character has been changed to backspace, which works nicely on video terminals. You can quickly check which is the case on your system:

```
$ datee←                      Try ←
datee←: not found             It's not ←
$ datee#                      Try #
Mon Sep 26 12:26:08 EDT 1983  It is #
$
```

(We printed the backspace as ← so you can see it.) Another common choice is *ctl-u* for line kill.

We will use the sharp as the erase character for the rest of this section because it's visible, but make the mental adjustment if your system is different. Later on, in "tailoring the environment," we will tell you how to set the erase and line kill characters to whatever you like, once and for all.

What if you must enter an erase or line kill character as part of the text? If you precede either # or @ by a backslash \, it loses its special meaning. So to enter a # or @, type \# or \@. The system may advance the terminal's cursor to the next line after your @, even if it was preceded by a backslash. Don't worry — the at-sign has been recorded.

The backslash, sometimes called the *escape character*, is used extensively to indicate that the following character is in some way special. To erase a backslash, you have to type two erase characters: \##. Do you see why?

The characters you type are examined and interpreted by a sequence of programs before they reach their destination, and exactly how they are interpreted depends not only on where they end up but how they got there.

Every character you type is immediately echoed to the terminal, unless echoing is turned off, which is rare. Until you press RETURN, the characters are held temporarily by the kernel, so typing mistakes can be corrected with the erase and line kill characters. When an erase or line kill character is preceded by a backslash, the kernel discards the backslash and holds the following character without interpretation.

When you press RETURN, the characters being held are sent to the program that is reading from the terminal. That program may in turn interpret the characters in special ways; for example, the shell turns off any special interpretation of a character if it is preceded by a backslash. We'll come back to this in Chapter 3. For now, you should remember that the kernel processes erase and line kill, and backslash only if it precedes erase or line kill; whatever characters are left after that may be interpreted by other programs as well.

Exercise 1-1. Explain what happens with

```
$ date\@
```

□

Exercise 1-2. Most shells (though not the 7th Edition shell) interpret # as introducing a comment, and ignore all text from the # to the end of the line. Given this, explain the following transcript, assuming your erase character is also #:

```
$ date
Mon Sep 26 12:39:56 EDT 1983
$ #date
Mon Sep 26 12:40:21 EDT 1983
$ \#date
$ \\#date
#date: not found
$
```

□

Type-ahead

The kernel reads what you type as you type it, even if it's busy with something else, so you can type as fast as you want, whenever you want, even when some command is printing at you. If you type while the system is printing, your input characters will appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. You can type commands one after another without waiting for them to finish or even to begin.

Stopping a program

You can stop most commands by typing the character DELETE. The BREAK key found on most terminals may also work, although this is system dependent. In a few programs, like text editors, DELETE stops whatever the program is doing but leaves you in that program. Turning off the terminal or

You will need a copy of the *UNIX Programmer's Manual*, even as you read this chapter; it's often easier for us to tell you to read about something in the manual than to repeat its contents here. This book is not supposed to replace it, but to show you how to make best use of the commands described in it. Furthermore, there may be differences between what we say here and what is true on your system. The manual has a permuted index at the beginning that's indispensable for finding the right programs to apply to a problem; learn to use it.

Finally, a word of advice: don't be afraid to experiment. If you are a beginner, there are very few accidental things you can do to hurt yourself or other users. So learn how things work by trying them. This is a long chapter, and the best way to read it is a few pages at a time, trying things out as you go.

1.1 Getting started

Some prerequisites about terminals and typing

To avoid explaining everything about using computers, we must assume you have some familiarity with computer terminals and how to use them. If any of the following statements are mystifying, you should ask a local expert for help.

The UNIX system is *full duplex*: the characters you type on the keyboard are sent to the system, which sends them back to the terminal to be printed on the screen. Normally, this *echo* process copies the characters directly to the screen, so you can see what you are typing, but sometimes, such as when you are typing a secret password, the echo is turned off so the characters do not appear on the screen.

Most of the keyboard characters are ordinary printing characters with no special significance, but a few tell the computer how to interpret your typing. By far the most important of these is the RETURN key. The RETURN key signifies the end of a line of input; the system echoes it by moving the terminal's cursor to the beginning of the next line on the screen. RETURN must be pressed before the system will interpret the characters you have typed.

RETURN is an example of a *control character* — an invisible character that controls some aspect of input and output on the terminal. On any reasonable terminal, RETURN has a key of its own, but most control characters do not. Instead, they must be typed by holding down the CONTROL key, sometimes called CTL or CNTL or CTRL, then pressing another key, usually a letter. For example, RETURN may be typed by pressing the RETURN key or, equivalently, holding down the CONTROL key and typing an 'm'. RETURN might therefore be called a control-m, which we will write as *ctl*-m. Other control characters include *ctl*-d, which tells a program that there is no more input; *ctl*-g, which rings the bell on the terminal; *ctl*-h, often called backspace, which can be used to correct typing mistakes; and *ctl*-i, often called tab, which

advances the cursor to the next tab stop, much as on a regular typewriter. Tab stops on UNIX systems are eight spaces apart. Both the backspace and tab characters have their own keys on most terminals.

Two other keys have special meaning: DELETE, sometimes called RUBOUT or some abbreviation, and BREAK, sometimes called INTERRUPT. On most UNIX systems, the DELETE key stops a program immediately, without waiting for it to finish. On some systems, *ctl*-c provides this service. And on some systems, depending on how the terminals are connected, BREAK is a synonym for DELETE or *ctl*-c.

A Session with UNIX

Let's begin with an annotated dialog between you and your UNIX system. Throughout the examples in this book, what you type is printed in *slanted letters*, computer responses are in `typewriter-style characters`, and explanations are in *italics*.

```
Establish a connection: dial a phone or turn on a switch as necessary.
Your system should say
login: you                   Type your name, then press RETURN
Password:                    Your password won't be echoed as you type it
You have mail.               There's mail to be read after you log in
$                            The system is now ready for your commands
$                            Press RETURN a couple of times
$ date                       What's the date and time?
Sun Sep 25 23:02:57 EDT 1983
$ who                        Who's using the machine?
jlb      tty0    Sep 25 13:59
you      tty2    Sep 25 23:01
mary     tty4    Sep 25 19:03
doug     tty5    Sep 25 19:22
egb      tty7    Sep 25 17:17
bob      tty8    Sep 25 20:48
$ mail                       Read your mail
From doug Sun Sep 25 20:53 EDT 1983
give me a call sometime monday

?                            RETURN moves on to the next message
From mary Sun Sep 25 19:07 EDT 1983     Next message
Lunch at noon tomorrow?

? d                          Delete this message
$                            No more mail
$ mail mary                  Send mail to mary
lunch at 12 is fine
ctl-d                        End of mail
$                            Hang up phone or turn off terminal
                             and that's the end
```

Sometimes that's all there is to a session, though occasionally people do