8

The Bourne Again Shell

In This Chapter

This chapter picks up where Chapter 5 left off by focusing on the Bourne Again Shell (bash). It notes where the TC Shell (tcsh) implementation of a feature differs from that of bash; if appropriate, you are directed to the page where the alternative implementation is discussed. Chapter 13 expands on this chapter, exploring control flow commands and more advanced aspects of programming the Bourne Again Shell. The bash home page is www.gnu.org/software/bash. The bash info page is a complete Bourne Again Shell reference.

The Bourne Again Shell and TC Shell are command interpreters and high-level programming languages. As command interpreters, they process commands you enter on the command line in response to a prompt. When you use the shell as a programming language, it processes commands stored in files called *shell scripts*. Like other languages, shells have variables and control flow commands (for example, **for** loops and **if** statements).

When you use a shell as a command interpreter, you can customize the environment you work in. You can make your prompt display the name of the working directory, create a

function or alias for cp that keeps it from overwriting certain kinds of files, take advantage of keyword variables to change aspects of how the shell works, and so on. You can also write shell scripts that do your bidding, from a one-line script that stores a long, complex command to a longer script that runs a set of reports, prints them, and mails you a reminder when the job is done. More complex shell scripts are themselves programs; they do not just run other programs. Chapter 13 has some examples of these types of scripts.

Most system shell scripts are written to run under the Bourne Again Shell. If you will ever work in single-user mode—as when you boot your system or do system maintenance, administration, or repair work, for example—it is a good idea to become familiar with this shell.

This chapter expands on the interactive features of the shell described in Chapter 5, explains how to create and run simple shell scripts, discusses job control, introduces the basic aspects of shell programming, talks about history and aliases, and describes command line expansion. Chapter 9 covers interactive use of the TC Shell and TC Shell programming, and Chapter 13 presents some more challenging shell programming problems.

BACKGROUND

The Bourne Again Shell is based on the Bourne Shell (the early UNIX shell; this book refers to it as the *original Bourne Shell* to avoid confusion), which was written by Steve Bourne of AT&T's Bell Laboratories. Over the years the original Bourne Shell has been expanded but it remains the basic shell provided with many commercial versions of UNIX.

sh Shell

Because of its long and successful history, the original Bourne Shell has been used to write many of the shell scripts that help manage UNIX systems. Some of these scripts appear in Mac OS X as Bourne Again Shell scripts. Although the Bourne Again Shell includes many extensions and features not found in the original Bourne Shell, bash maintains compatibility with the original Bourne Shell so you can run Bourne Shell scripts under bash. On UNIX systems the original Bourne Shell is named sh. On Mac OS X sh is a copy of bash ensuring that scripts that require the presence of the Bourne Shell still run. When called as sh, bash does its best to emulate the original Bourne Shell.

Korn Shell

System V UNIX introduced the Korn Shell (ksh), written by David Korn. This shell extended many features of the original Bourne Shell and added many new features. Some features of the Bourne Again Shell, such as command aliases and command line editing, are based on similar features from the Korn Shell. Recent releases of Mac OS X include the Korn Shell.

POSIX standards

The POSIX (the Portable Operating System Interface) family of related standards is being developed by PASC (IEEE's Portable Application Standards Committee, www.pasc.org). A comprehensive FAQ on POSIX, including many links, appears at www.opengroup.org/austin/papers/posix_faq.html.

POSIX standard 1003.2 describes shell functionality. The Bourne Again Shell provides the features that match the requirements of this POSIX standard. Efforts are under way to make the Bourne Again Shell fully comply with the POSIX standard. In the meantime, if you invoke bash with the **––posix** option, the behavior of the Bourne Again Shell will more closely match the POSIX requirements.

SHELL BASICS

This section covers writing and using startup files, redirecting standard error, writing and executing simple shell scripts, separating and grouping commands, implementing job control, and manipulating the directory stack.

STARTUP FILES

When a shell starts, it runs startup files to initialize itself. Which files the shell runs depends on whether it is a login shell, an interactive shell that is not a login shell (such as you get by giving the command **bash**), or a noninteractive shell (one used to execute a shell script). You must have read access to a startup file to execute the commands in it. Mac OS X puts appropriate commands in some of these files. This section covers bash startup files. See page 340 for information on tcsh startup files.

LOGIN SHELLS

The files covered in this section are executed by login shells and shells that you start with the **––login** option. Login shells are, by their nature, interactive.

/etc/profile The shell first executes the commands in **/etc/profile**. By default when called from bash, a command in this file calls **/etc/bashrc**. Superuser can set up this file to establish systemwide default characteristics for bash users.

.bash_profile
.bash_login
.profile Next the shell looks for **~/.bash_profile**, **~/.bash_login**, and **~/.profile** (**~/** is shorthand for your home directory), in that order, executing the commands in the first of these files it finds. You can put commands in one of these files to override the defaults set in **/etc/profile**.

.bash_logout When you log out, bash executes commands in the **~/.bash_logout** file. Frequently commands that clean up after a session, such as those that remove temporary files, go in this file.

INTERACTIVE NONLOGIN SHELLS

The commands in the preceding startup files are not executed by interactive, nonlogin shells. However, these shells inherit from the login shell variables that are set by these startup files.

/etc/bashrc Although not called by bash directly, many **~/.bashrc** files call **/etc/bashrc**. This setup allows Superuser to establish systemwide default characteristics for nonlogin bash shells.

.bashrc An interactive nonlogin shell executes commands in the **~/.bashrc** file. Typically a startup file for a login shell, such as **.bash_profile**, runs this file, so that both login and nonlogin shells benefit from the commands in **.bashrc**.

NONINTERACTIVE SHELLS

The commands in the previously described startup files are not executed by noninteractive shells, such as those that runs shell scripts. However, these shells inherit from the login shell variables that are set by these startup files.

BASH_ENV Noninteractive shells look for the environment variable **BASH_ENV** (or **ENV**, if the shell is called as sh) and execute commands in the file named by this variable.

SETTING UP STARTUP FILES

Although many startup files and types of shells exist, usually all you need are the **.bash_profile** and **.bashrc** files in your home directory. Commands similar to the following in **.bash_profile** run commands from **.bashrc** for login shells (when **.bashrc** exists). With this setup, the commands in **.bashrc** are executed by login and nonlogin shells.

```
if [ -f ~/.bashrc ]; then source ~/.bashrc; fi
```

The [**–f ~/.bashrc**] tests whether the file named **.bashrc** in your home directory exists. See page 871 for more information on test and its synonym [ ].

Use .bash_profile to set PATH

tip Because commands in **.bashrc** may be executed many times, and because subshells inherit exported variables, it is a good idea to put commands that add to existing variables in the **.bash_profile** file. For example, the following command adds the **bin** subdirectory of the **home** directory to **PATH** (page 285) and should go in **.bash_profile**:

```
PATH=$PATH:$HOME/bin
```

When you put this command in **.bash_profile** and not in **.bashrc**, the string is added to the **PATH** variable only once, when you log in.

Modifying a variable in **.bash_profile** allows changes you make in an interactive session to propagate to subshells. In contrast, modifying a variable in **.bashrc** overrides changes inherited from a parent shell.

Sample **.bash_profile** and **.bashrc** files follow. Some of the commands used in these files are not covered until later in this chapter. In any startup file, you must export variables and functions that you want to be available to child processes. For more information refer to "Locality of Variables" on page 560.

```
$ cat ~/.bash_profile
if [ -f ~/.bashrc ]; then
    source ~/.bashrc          # read local startup file if it exists
fi
PATH=$PATH:.                  # add the working directory to PATH
export PS1='[\h \W \!]\$ '    # set prompt
```

The first command in the preceding **.bash_profile** file executes the commands in the user's **.bashrc** file if it exists. The next command adds to the **PATH** variable (page 285). Typically **PATH** is set and exported in **/etc/profile** so it does not need to be exported in a user's startup file. The final command sets and exports **PS1** (page 286), which controls the user's prompt.

Next is a sample **.bashrc** file. The first command executes the commands in the **/etc/bashrc** file if it exists. Next the **VIMINIT** (page 180) variable is set and exported and several aliases (page 311) are established. The final command defines a function (page 314) that swaps the names of two files.

```
$ cat ~/.bashrc
if [ -f /etc/bashrc ]; then
    source /etc/bashrc        # read global startup file if it exists
fi

set -o noclobber              # prevent overwriting files
unset MAILCHECK               # turn off "you have new mail" notice
export VIMINIT='set ai aw'    # set vim options
alias df='df -h'              # set up aliases
alias rm='rm -i'              # always do interactive rm's
alias lt='ls -ltrh | tail'
alias h='history | tail'
alias ch='chmod 755 '

function switch()             # a function to exchange the names
{                             # of two files
    local tmp=$$switch
    mv "$1" $tmp
    mv "$2" "$1"
    mv $tmp "$2"
}
```

. (DOT) OR source: RUNS A STARTUP FILE IN THE CURRENT SHELL

After you edit a startup file such as **.bashrc**, you do not have to log out and log in again to put the changes into effect. You can run the startup file using the **.** (dot) or source builtin (they are the same command under bash; only source is available under tcsh [page 376], and only the dot builtin is available under ksh). As with all other commands, the **.** must be followed by a SPACE on the command line. Using the **.** or source builtin is similar to running a shell script, except that these commands run the script as part of the current process. Consequently, when you use **.** or source to run a script, changes you make to variables from within the script affect the shell that you run the script from. You can use the **.** or source command to run any shell script—not just a startup file—but undesirable side effects (such as changes in the values of shell variables you rely on) may occur. If you ran a startup file as a regular shell script and did not use the **.** or source builtin, the variables created in the startup file would remain in effect only in the subshell running the script—not in the shell you ran the script from. For more information refer to "Locality of Variables" on page 560.

In the following example, **.bashrc** sets several variables and sets **PS1**, the prompt, to the name of the host. The **.** builtin puts the new values into effect.

```
$ cat ~/.bashrc
export TERM=vt100             # set the terminal type
export PS1="$(hostname -f): " # set the prompt string
export CDPATH=:$HOME          # add HOME to CDPATH string
stty kill '^u'                # set kill line to control-u
$ . ~/.bashrc
bravo.example.com:
```

COMMANDS THAT ARE SYMBOLS

The Bourne Again Shell uses the symbols (,), [,], and **$** in a variety of ways. To minimize confusion, Table 8-1 lists the most common use of each of these symbols, even though some of them are not introduced until later.

Table 8-1 Builtin commands that are symbols

Symbol	Command
( )	Subshell (page 271)
$( )	Command substitution (page 327)
(( ))	Arithmetic evaluation; a synonym for let (use when the enclosed value contains an equal sign) (page 585)
$(( ))	Arithmetic expansion (not for use with an enclosed equal sign) (page 325)
[ ]	The test command. (pages 525, 527, 540, and 871)
[[ ]]	Conditional expression; similar to **[ ]** but adds string comparisons (page 586)

REDIRECTING STANDARD ERROR

Chapter 5 covered the concept of standard output and explained how to redirect standard output of a command. In addition to standard output, commands can send output to *standard error*. A command can send error messages to standard error to keep them from getting mixed up with the information it sends to standard output.

Just as it does with standard output, by default the shell sends a command's standard error to the screen. Unless you redirect one or the other, you may not know the difference between the output a command sends to standard output and the output it sends to standard error. This section covers the syntax used by the Bourne Again Shell. See page 346 if you are using the TC Shell.

File descriptors

A *file descriptor* is the place a program sends its output to and gets its input from. When you execute a program, the process running the program opens three file descriptors: 0 (standard input), 1 (standard output), and 2 (standard error). The redirect output symbol (**>** [page 122]) is shorthand for **1>**, which tells the shell to redirect standard output. Similarly **<** (page 124) is short for **0<**, which redirects standard input. The symbols **2>** redirect standard error. For more information refer to "File Descriptors" on page 555.

The following examples demonstrate how to redirect standard output and standard error to different files and to the same file. When you run the cat utility with the name of a file that does not exist and the name of a file that does exist, cat sends an error message to standard error and copies the file that does exist to standard output. Unless you redirect them, both messages appear on the screen.

```
$ cat y
This is y.
$ cat x
cat: x: No such file or directory

$ cat x y
cat: x: No such file or directory
This is y.
```

When you redirect standard output of a command, output sent to standard error is not affected and still appears on the screen.

```
$ cat x y > hold
cat: x: No such file or directory
$ cat hold
This is y.
```

Similarly, when you send standard output through a pipe, standard error is not affected. The following example sends standard output of cat through a pipe to tr (page 879), which in this example converts lowercase characters to uppercase. The text that cat sends to standard error is not translated because it goes directly to the screen rather than through the pipe.

```
$ cat x y | tr "[a-z]" "[A-Z]"
cat: x: No such file or directory
THIS IS Y.
```

The following example redirects standard output and standard error to different files. The notation **2>** tells the shell where to redirect standard error (file descriptor 2). The **1>** tells the shell where to redirect standard output (file descriptor 1). You can use **>** in place of **1>**.

```
$ cat x y 1> hold1 2> hold2
$ cat hold1
This is y.
$ cat hold2
cat: x: No such file or directory
```

Duplicating a file descriptor

In the next example, **1>** redirects standard output to **hold**. Then **2>&1** declares file descriptor 2 to be a duplicate of file descriptor 1. As a result both standard output and standard error are redirected to **hold**.

```
$ cat x y 1> hold 2>&1
$ cat hold
cat: x: No such file or directory
This is y.
```

In the preceding example, **1> hold** precedes **2>&1**. If they had been listed in the opposite order, standard error would have been made a duplicate of standard output

before standard output was redirected to **hold**. In that case only standard output would have been redirected to **hold**.

The next example declares file descriptor 2 to be a duplicate of file descriptor 1 and sends the output for file descriptor 1 through a pipe to the tr command.

```
$ cat x y 2>&1 | tr "[a-z]" "[A-Z]"
CAT: X: NO SUCH FILE OR DIRECTORY
THIS IS Y.
```

Sending errors to standard error

You can also use **1>&2** to redirect standard output of a command to standard error. This technique is often used in shell scripts to send the output of echo to standard error. In the following script, standard output of the first echo is redirected to standard error:

```
$ cat message_demo
echo This is an error message. 1>&2
echo This is not an error message.
```

If you redirect standard output of **message_demo**, error messages such as the one produced by the first echo will still go to the screen because you have not redirected standard error. Because standard output of a shell script is frequently redirected to another file, you can use this technique to display on the screen error messages generated by the script. The **lnks** script (page 532) uses this technique. You can also use the exec builtin to create additional file descriptors and to redirect standard input, standard output, and standard error of a shell script from within the script (page 574).

The Bourne Again Shell supports the redirection operators shown in Table 8-2.

Table 8-2 Redirection operators

Operator	Meaning
< ***filename***	Redirects standard input from ***filename***.
> ***filename***	Redirects standard output to ***filename*** unless ***filename*** exists and **noclobber** (page 125) is set. If **noclobber** is not set, this redirection creates ***filename*** if it does not exist.
>| ***filename***	Redirects standard output to ***filename***, even if the file exists and **noclobber** (page 125) is set.
>> ***filename***	Redirects and appends standard output to ***filename*** unless ***filename*** exists and **noclobber** (page 125) is set. If **noclobber** is not set, this redirection creates ***filename*** if it does not exist.
<&***m***	Duplicates standard input from file descriptor ***m*** (page 555).
[n]>&***m***	Duplicates standard output or file descriptor ***n*** if specified from file descriptor ***m*** (page 555).
[n]<&–	Closes standard input or file descriptor ***n*** if specified (page 555).
[n]>&–	Closes standard output or file descriptor ***n*** if specified.

WRITING A SIMPLE SHELL SCRIPT

A *shell script* is a file that contains commands that the shell can execute. The commands in a shell script can be any commands you can enter in response to a shell prompt. For example, a command in a shell script might run a Mac OS X utility, a compiled program, or another shell script. Like the commands you give on the command line, a command in a shell script can use ambiguous file references and can have its input or output redirected from or to a file or sent through a pipe (page 128). You can also use pipes and redirection with the input and output of the script itself.

In addition to the commands you would ordinarily use on the command line, *control flow* commands (also called *control structures*) find most of their use in shell scripts. This group of commands enables you to alter the order of execution of commands in a script just as you would alter the order of execution of statements using a structured programming language. Refer to "Control Structures" on page 524 (bash) and page 364 (tcsh) for specifics.

The shell interprets and executes the commands in a shell script, one after another. Thus a shell script enables you to simply and quickly initiate a complex series of tasks or a repetitive procedure.

chmod: MAKES A FILE EXECUTABLE

To execute a shell script by giving its name as a command, you must have permission to read and execute the file that contains the script (refer to "Access Permissions" on page 87). Read permission enables you to read the file that holds the script. Execute permission tells the shell and the system that the owner, group, and/or public has permission to execute the file; it implies that the content of the file is executable.

When you create a shell script using an editor, the file does not typically have its execute permission set. The following example shows a file named **whoson** that contains a shell script:

```
$ cat whoson
date
echo "Users Currently Logged In"
who

$ whoson
bash: ./whoson: Permission denied
```

You cannot execute **whoson** by giving its name as a command because you do not have execute permission for the file. The shell does not recognize **whoson** as an executable file and issues an error message when you try to execute it. When you give the filename as an argument to bash (**bash whoson**), bash takes the argument to be a shell script and executes it. In this case **bash** is executable and **whoson** is an argument that bash executes so you do not need to have permission to execute **whoson**. You can do the same with tcsh script files.

Command not found?

tip If you get the message

```
$ whoson
bash: whoson: command not found
```

the shell is not set up to search for executable files in the working directory. Give this command instead:

```
$ ./whoson
```

The **./** tells the shell explicitly to look for an executable file in the working directory. To change the environment so that the shell searches the working directory automatically, see page 285.

The chmod utility changes the access privileges associated with a file. Figure 8-1 shows ls with the **–l** option displaying the access privileges of **whoson** before and after chmod gives execute permission to the file's owner.

The first ls displays a hyphen (**–**) as the fourth character, indicating that the owner does not have permission to execute the file. Next chmod gives the owner execute permission: The **u+x** causes chmod to add (**+**) execute permission (**x**) for the owner (**u**). (The **u** stands for *user*, although it means the owner of the file who may be the user of the file at any given time.) The second argument is the name of the file. The second ls shows an **x** in the fourth position, indicating that the owner now has execute permission.

If other users will execute the file, you must also change group and/or public access permissions for the file. Any user must have execute access to use the file's name as a command. If the file is a shell script, the user trying to execute the file must also have read access to the file. You do not need read access to execute a binary executable (compiled program).

```
$ ls -l whoson
-rw-rw-r--   1 alex alex 40 May 24 11:30 whoson

$ chmod u+x whoson
$ ls -l whoson
-rwxrw-r--   1 alex alex 40 May 24 11:30 whoson

$ whoson
Tue May 24 11:40:49 PDT 2005
Users Currently Logged In
zach     console  May  2 12:39
sam      ttyp1    May  4 17:23 (localhost)
zach     ttyp2    May  3 03:32
max      ttyp3    May  4 17:24 (localhost)
```

Figure 8-1 Using chmod to make a shell script executable

The final command in Figure 8-1 shows the shell executing the file when its name is given as a command. For more information refer to "Access Permissions" on page 87 and to ls and chmod in Part V.

#! SPECIFIES A SHELL

You can put a special sequence of characters on the first line of a file to tell the operating system which shell should execute the file. Because the operating system checks the initial characters of a program before attempting to **exec** it, these characters save the system from making an unsuccessful attempt. If **#!** are the first two characters of a script, the system interprets the characters that follow as the absolute pathname of the utility that should execute the script. This can be the pathname of any program, not just a shell. The following example specifies that bash should run the script:

```
$ cat bash_script
#!/bin/bash
echo "This is a Bourne Again Shell script."
```

The **#!** characters are useful if you have a script that you want to run with a shell other than the shell you are running the script from. The following example shows a script that should be executed by tcsh:

```
$ cat tcsh_script
#!/bin/tcsh
echo "This is a tcsh script."
set person = jenny
echo "person is $person"
```

Because of the **#!** line, the operating system ensures that tcsh executes the script no matter which shell you run it from.

You can use **ps –l** within a shell script to display the name of the shell that is executing the script. The three lines that ps displays in the following example show the process running the parent bash shell, the process running the tcsh script, and the process running the ps command:

```
$ cat tcsh_script2
#!/bin/tcsh
ps -l

$ tcsh_script2
  UID   PID  PPID CPU PRI NI   VSZ  RSS WCHAN STAT  TT    TIME COMMAND
  501   915   914   0  31  0 27792  828 -     S+    p2 0:00.07 -bash
  501  2160  2156   0  31  0 27792  808 -     Ss    p4 0:00.02 -bash
  501  2165  2160   0  31  0 31816 1148 -     S+    p4 0:00.01 /bin/tc
```

If you do not follow **#!** with the name of an executable program, the shell reports that it cannot find the command that you asked it to run. You can optionally follow **#!** with SPACES. If you omit the **#!** line and try to run, for example, a tcsh script from bash, the shell may generate error messages or the script may not run properly.

See page 653 for an example of a stand-alone sed script that uses **#!**.

BEGINS A COMMENT

Comments make shell scripts and all code easier to read and maintain by you and others. The comment syntax is common to both the Bourne Again and the TC Shells.

If a pound sign (**#**) in the first character position of the first line of a script is not immediately followed by an exclamation point (**!**) or if a pound sign occurs in any other location in a script, the shell interprets it as the beginning of a comment. The shell then ignores everything between the pound sign and the end of the line (the next NEWLINE character).

RUNNING A SHELL SCRIPT

fork and **exec** system calls

A command on the command line causes the shell to **fork** a new process, creating a duplicate of the shell process (a subshell). The new process attempts to **exec** (execute) the command. Like **fork**, the **exec** routine is executed by the operating system (a system call). If the command is a binary executable program, such as a compiled C program, **exec** succeeds and the system overlays the newly created subshell with the executable program. If the command is a shell script, **exec** fails. When **exec** fails, the command is assumed to be a shell script, and the subshell runs the commands in the script. Unlike a login shell, which expects input from the command line, the subshell takes its input from a file: the shell script.

As discussed earlier, if you have a shell script in a file that you do not have execute permission for, you can run the commands in the script by using a bash command to **exec** a shell to run the script directly. In the following example, bash creates a new shell that takes its input from the file named **whoson**:

```
$ bash whoson
```

Because the bash command expects to read a file containing commands, you do not need execute permission for **whoson**. (You do need read permission.) Even though bash reads and executes the commands in **whoson**, standard input, standard output, and standard error remain connected to the terminal.

Although you can use bash to execute a shell script, this technique causes the script to run more slowly than giving yourself execute permission and directly invoking the script. Users typically prefer to make the file executable and run the script by typing its name on the command line. It is also easier to type the name, and this practice is consistent with the way other kinds of programs are invoked (so you do not need to know whether you are running a shell script or another kind of program). However, if bash is not your interactive shell or if you want to see how the script runs with different shells, you may want to run a script as an argument to **bash** or **tcsh**.

SEPARATING AND GROUPING COMMANDS

Whether you give the shell commands interactively or write a shell script, you must separate commands from one another. This section, which applies to the Bourne

sh does not call the original Bourne Shell

caution The original Bourne Shell was invoked with the command **sh**. Although you can call bash with an **sh** command, it is not the original Bourne Shell. The **sh** command (**/bin/sh**) is a copy of **/bin/bash**, so it is simply another name for the **bash** command. When you call bash using the command **sh**, bash tries to mimic the behavior of the original Bourne Shell as closely as possible. It does not always succeed.

Again and the TC Shells, reviews the ways to separate commands that were covered in Chapter 5 and introduces a few new ones.

; AND NEWLINE SEPARATE COMMANDS

The NEWLINE character is a unique command separator because it initiates execution of the command preceding it. You have seen this throughout this book each time you press the RETURN key at the end of a command line.

The semicolon (**;**) is a command separator that *does not* initiate execution of a command and *does not* change any aspect of how the command functions. You can execute a series of commands sequentially by entering them on a single command line and separating each from the next with a semicolon (**;**). You initiate execution of the sequence of commands by pressing RETURN:

```
$ x ; y ; z
```

If **x**, **y**, and **z** are commands, the preceding command line yields the same results as the next three commands. The difference is that in the next example the shell issues a prompt after each of the commands (**x**, **y**, and **z**) finishes executing, whereas the preceding command line causes the shell to issue a prompt only after **z** is complete:

```
$ x
$ y
$ z
```

Whitespace Although the whitespace around the semicolons in the earlier example makes the command line easier to read, it is not necessary. None of the command separators needs to be surrounded by SPACEs or TABs.

\ CONTINUES A COMMAND

When you enter a long command line and the cursor reaches the right side of the screen, you can use a backslash (\) character to continue the command on the next line. The backslash quotes, or escapes, the NEWLINE character that follows it so that the shell does not treat the NEWLINE as a command terminator. Enclosing a backslash within single quotation marks turns off the power of a backslash to quote special characters such as NEWLINE. Enclosing a backslash within double quotation marks has no effect on the power of the backslash.

Although you can break a line in the middle of a word (token), it is typically easier to break a line just before or after whitespace.

optional You can enter a RETURN in the middle of a quoted string on a command line without using a backslash. The NEWLINE (RETURN) that you enter will then be part of the string:

```
$ echo "Please enter the three values
> required to complete the transaction."
Please enter the three values
required to complete the transaction.
```

In the three examples in this section, the shell does not interpret RETURN as a command terminator because it occurs within a quoted string. The **>** is a secondary prompt indicating that the shell is waiting for you to continue the unfinished command. In the next example, the first RETURN is quoted (escaped) so the shell treats it as a separator and does not interpret it literally.

```
$ echo "Please enter the three values \
> required to complete the transaction."
Please enter the three values required to complete the transaction.
```

Single quotation marks cause the shell to interpret a backslash literally:

```
$ echo 'Please enter the three values \
> required to complete the transaction.'
Please enter the three values \
required to complete the transaction.
```

| AND & SEPARATE COMMANDS AND DO SOMETHING ELSE

The pipe symbol (|) and the background task symbol (&) are also command separators. They *do not* start execution of a command but *do* change some aspect of how the command functions. The pipe symbol alters the source of standard input or the destination of standard output. The background task symbol causes the shell to execute the task in the background so you get a prompt immediately and can continue working on other tasks.

Each of the following command lines initiates a single job comprising three tasks:

```
$ x | y | z
$ ls -l | grep tmp | less
```

In the first job, the shell redirects standard output of task **x** to standard input of task **y** and redirects **y**'s standard output to **z**'s standard input. Because it runs the entire job in the foreground, the shell does not display a prompt until task **z** runs to completion: Task **z** does not finish until task **y** finishes, and task **y** does not finish until task **x** finishes. In the second job, task **x** is an **ls –l** command, task **y** is **grep tmp**, and task **z** is the pager **less**. The shell displays a long (wide) listing of the files in the working directory that contain the string **tmp**, piped through less.

The next command line executes tasks **d** and **e** in the background and task **f** in the foreground:

```
$ d & e & f
[1] 14271
[2] 14272
```

The shell displays the job number between brackets and the PID (process identification) number for each process running in the background. You get a prompt as soon as **f** finishes, which may be before **d** or **e** finishes.

Before displaying a prompt for a new command, the shell checks whether any background jobs have completed. For each job that has completed, the shell displays its job number, the word **Done**, and the command line that invoked the job; then the shell displays a prompt. When the job numbers are listed, the number of the last job started is followed by a **+** character and the job number of the previous job is followed by a **–** character. Any other jobs listed show a SPACE character. After running the last command, the shell displays the following before issuing a prompt:

```
[1]-  Done                    d
[2]+  Done                    e
```

The next command line executes all three tasks as background jobs. You get a shell prompt immediately:

```
$ d & e & f &
[1] 14290
[2] 14291
[3] 14292
```

You can use pipes to send the output from one task to the next task and an ampersand (&) to run the entire job as a background task. Again the prompt comes back immediately. The shell regards the commands joined by a pipe as being a single job. That is, it treats all pipes as single jobs, no matter how many tasks are connected with the pipe (|) symbol or how complex they are. The Bourne Again Shell shows only one process placed in the background:

```
$ d | e | f &
[1] 14295
```

The TC Shell shows three processes (all belonging to job 1) placed in the background:

```
tcsh $ d | e | f &
[1] 14302 14304 14306
```

optional () GROUPS COMMANDS

You can use parentheses to group commands. The shell creates a copy of itself, called a *subshell,* for each group. It treats each group of commands as a job and creates a new process to execute each command (refer to "Process Structure" on page 293 for more information on creating subshells). Each subshell (job) has its own environment, meaning that it has its own set of variables with values that can differ from those of other subshells.

The following command line executes commands **a** and **b** sequentially in the background while executing **c** in the background. The shell prompt returns immediately.

```
$ (a ; b) & c &
[1] 15520
[2] 15521
```

The preceding example differs from the earlier example **d & e & f &** in that tasks **a** and **b** are initiated sequentially, not concurrently.

Similarly the following command line executes **a** and **b** sequentially in the background and, at the same time, executes **c** and **d** sequentially in the background. The subshell running **a** and **b** and the subshell running **c** and **d** run concurrently. The prompt returns immediately.

```
$ (a ; b) & (c ; d) &
[1] 15528
[2] 15529
```

The next script copies one directory to another. The second pair of parentheses creates a subshell to run the commands following the pipe. Because of these parentheses, the output of the first tar command is available for the second tar command despite the intervening cd command. Without the parentheses, the output of the first tar command would be sent to cd and lost because cd does not process input from standard input. The shell variables **$1** and **$2** represent the first and second command line arguments (page 565), respectively. The first pair of parentheses, which creates a subshell to run the first two commands, allows users to call **cpdir** with relative pathnames. Without them the first cd command would change the working directory of the script (and consequently the working directory of the second cd command). With them only the working directory of the subshell is changed.

```
$ cat cpdir
(cd $1 ; tar -cf - . ) | (cd $2 ; tar -xvf - )
$ cpdir /Users/alex/sources /Users/alex/memo/biblio
```

The **cpdir** command line copies the files and directories in the **/Users/alex/sources** directory to the directory named **/Users/alex/memo/biblio**. This shell script is almost the same as using cp with the **–r** option. Refer to Part V for more information on cp (page 690) and tar (page 862).

JOB CONTROL

A job is a command pipeline. You run a simple job whenever you give the shell a command. For example, type **date** on the command line and press RETURN: You have run a job. You can also create several jobs with multiple commands on a single command line:

```
$ find . -print | sort | lpr & grep -l alex /tmp/* > alexfiles &
[1] 18839
[2] 18876
```

The portion of the command line up to the first & is one job consisting of three processes connected by pipes: find (page 728), sort (page 49), and lpr (page 45). The second job is a single process running grep. Both jobs have been put into the background by the trailing & characters, so bash does not wait for them to complete before displaying a prompt.

Using job control you can move commands from the foreground to the background (and vice versa), stop commands temporarily, and list all the commands that are running in the background or stopped.

jobs: LISTS JOBS

The jobs builtin lists all background jobs. The following sequence demonstrates what happens when you give a **jobs** command. Here the sleep command runs in the background and creates a background job that jobs reports on:

```
$ sleep 60 &
[1] 7809
$ jobs
[1] + Running                 sleep 60 &
```

fg: BRINGS A JOB TO THE FOREGROUND

The shell assigns job numbers to commands you run in the background (page 270). Several jobs are started in the background in the next example. For each job the shell lists the job number and PID number immediately, just before it issues a prompt.

```
$ vim memo &
[1] 1246
$ date &
[2] 1247
$ Sun Dec 4 11:44:40 PST 2005
[2]+ Done          date
$ find /usr -name ace -print > findout &
[2] 1269
$ jobs
[1]- Running         vim memo &
[2]+ Running         find /usr -name ace -print > findout &
```

Job numbers, which are discarded when a job is finished, can be reused. When you start or put a job in the background, the shell assigns a job number that is one more than the highest job number in use.

In the preceding example, the jobs command lists the first job, **vim memo**, as job 1. The date command does not appear in the jobs list because it finished before jobs was run. Because the date command was completed before find was run, the find command became job 2.

To move a background job into the foreground, use the fg builtin followed by the job number. Alternatively, you can give a percent sign (%) followed immediately by

the job number as a command. Either of the following commands moves job 2 into the foreground:

```
$ fg 2
```

or

```
$ %2
```

You can also refer to a job by following the percent sign with a string that uniquely identifies the beginning of the command line used to start the job. Instead of the preceding command, you could have used either **fg %find** or **fg %f** because both uniquely identify job 2. If you follow the percent sign with a question mark and a string, the string can match any part of the command line. In the preceding example, **fg %?ace** also brings job 2 into the foreground.

Often the job you wish to bring into the foreground is the only job running in the background or is the job that jobs lists with a plus (+). In these cases you can use fg without an argument.

bg: SENDS A JOB TO THE BACKGROUND

To move the foreground job to the background, you must first suspend (temporarily stop) the job by pressing the suspend key (usually CONTROL-Z). Pressing the suspend key immediately suspends the job in the foreground. You can then use the bg builtin to resume execution of the job in the background.

```
$ bg
```

If a background job attempts to read from the terminal, the shell stops it and notifies you that the job has been stopped and is waiting for input. You must then move the job into the foreground so that it can read from the terminal. The shell displays the command line when it moves the job into the foreground.

```
$ (sleep 5; cat > mytext) &
[1] 1343
$ date
Sun Dec  4 11:58:20 PST 2005
[1]+ Stopped                 ( sleep 5; cat >mytext )
$ fg
( sleep 5; cat >mytext )
Remember to let the cat out!
CONTROL-D
$
```

In the preceding example, the shell displays the job number and PID number of the background job as soon as it starts, followed by a prompt. Demonstrating that you can give a command at this point, the user gives the command date and its output appears on the screen. The shell waits until just before it issues a prompt (after date has finished) to notify you that job 1 is stopped. When you give an **fg** command, the shell puts the job in the foreground and you can enter the input that the command is

waiting for. In this case the input needs to be terminated with a CONTROL-D to signify EOF (end of file). The shell then displays another prompt.

The shell keeps you informed about changes in the status of a job, notifying you when a background job starts, completes, or is stopped, perhaps waiting for input from the terminal. The shell also lets you know when a foreground job is suspended. Because notices about a job being run in the background can disrupt your work, the shell delays displaying these notices until just before it displays a prompt. You can set **notify** (page 320) to make the shell display these notices without delay.

If you try to exit from a shell while jobs are stopped, the shell issues a warning and does not allow you to exit. If you then use jobs to review the list of jobs or you immediately try to leave the shell again, the shell allows you to leave and terminates the stopped jobs. Jobs that are running (not stopped) in the background continue to run. In the following example, find (job 1) continues to run after the second exit terminates the shell, but cat (job 2) is terminated:

```
$ find / -size +100k > $HOME/bigfiles 2>&1 &
[1] 1426
$ cat > mytest &
[2] 1428
$ exit
exit
There are stopped jobs.

[2]+  Stopped                 cat >mytest
$ exit
exit

login:
```

MANIPULATING THE DIRECTORY STACK

Both the Bourne Again and the TC Shells allow you to store a list of directories you are working with, enabling you to move easily among them. This list is referred to as a *stack*. It is analogous to a stack of dinner plates: You typically add plates to and remove plates from the top of the stack, creating a last-in first-out, (*LIFO*) stack.

dirs: DISPLAYS THE STACK

The dirs builtin displays the contents of the directory stack. If you call dirs when the directory stack is empty, it displays the name of the working directory:

```
$ dirs
~/literature
```

The dirs builtin uses a tilde (~) to represent the name of the home directory. The examples in the next several sections assume that you are referring to the directory structure shown in Figure 8-2.

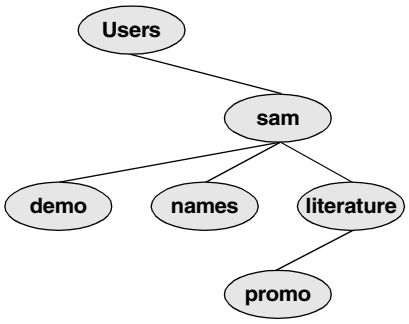

Figure 8-2 The directory structure in the examples

pushd: PUSHES A DIRECTORY ON THE STACK

To change directories and at the same time add a new directory to the top of the stack, use the pushd (push directory) builtin. In addition to changing directories, the pushd builtin displays the contents of the stack. The following example is illustrated in Figure 8-3:

```
$ pushd ../demo
~/demo ~/literature
$ pwd
/Users/sam/demo
$ pushd ../names
~/names ~/demo ~/literature
$ pwd
/Users/sam/names
```

When you use pushd without an argument, it swaps the top two directories on the stack and makes the new top directory (which was the second directory) become the new working directory (Figure 8-4):

```
$ pushd
~/demo ~/names ~/literature
$ pwd
/Users/sam/demo
```

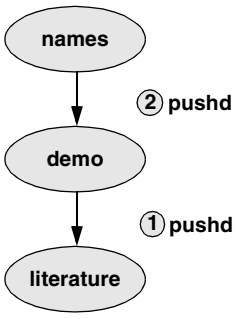

Figure 8-3 Creating a directory stack

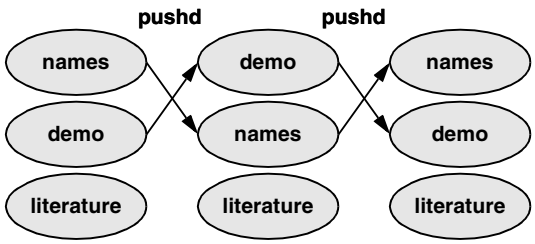

Figure 8-4 Using pushd to change working directories

Using pushd in this way, you can easily move back and forth between two directories. You can also use **cd –** to change to the previous directory, whether or not you have explicitly created a directory stack. To access another directory in the stack, call pushd with a numeric argument preceded by a plus sign. The directories in the stack are numbered starting with the top directory, which is number 0. The following pushd command continues with the previous example, changing the working directory to **literature** and moving **literature** to the top of the stack:

```
$ pushd +2
~/literature ~/demo ~/names
$ pwd
/Users/sam/literature
```

popd: POPS A DIRECTORY OFF THE STACK

To remove a directory from the stack, use the popd (pop directory) builtin. As the following example and Figure 8-5 show, popd used without an argument removes the top directory from the stack and changes the working directory to the new top directory:

```
$ dirs
~/literature ~/demo ~/names
$ popd
~/demo ~/names
$ pwd
/Users/sam/demo
```

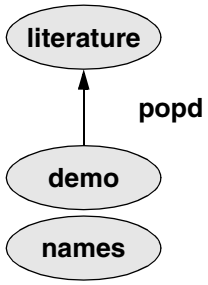

Figure 8-5 Using popd to remove a directory from the stack

To remove a directory other than the top one from the stack, use popd with a numeric argument preceded by a plus sign. The following example removes directory number 1, **demo**:

```
$ dirs
~/literature ~/demo ~/names
$ popd +1
~/literature ~/names
```

Removing a directory other than directory number 0 does not change the working directory.

PARAMETERS AND VARIABLES

Variables

Within a shell, a *shell parameter* is associated with a value that is accessible to the user. There are several kinds of shell parameters. Parameters whose names consist of letters, digits, and underscores are often referred to as *shell variables,* or simply *variables*. A variable name must start with a letter or underscore, not with a number. Thus **A76**, **MY_CAT**, and **___X___** are valid variable names, whereas **69TH_STREET** (starts with a digit) and **MY-NAME** (contains a hyphen) are not.

User-created variables

Shell variables that you name and assign values to are *user-created variables*. You can change the values of user-created variables at any time, or you can make them *readonly* so that their values cannot be changed. You can also make user-created variables *global*. A global variable (also called an *environment variable*) is available to all shells and other programs you fork from the original shell. One naming convention is to use only uppercase letters for global variables and to use mixed-case or lowercase letters for other variables. Refer to "Locality of Variables" on page 560 for more information on global variables.

To assign a value to a variable in the Bourne Again Shell, use the following syntax:

VARIABLE=value

There can be no whitespace on either side of the equal sign (**=**). An example assignment follows:

```
$ myvar=abc
```

Under the TC Shell the assignment must be preceded by the word **set** and the SPACEs on either side of the equal sign are optional:

```
$ set myvar = abc
```

The Bourne Again Shell permits you to put variable assignments on a command line. These assignments are local to the command shell—that is, they apply to the command only. The **my_script** shell script displays the value of **TEMPDIR**. The following command runs **my_script** with **TEMPDIR** set to **/Users/sam/temp**. The echo builtin shows that the interactive shell has no value for **TEMPDIR** after running **my_script**. If **TEMPDIR** had been set in the interactive shell, running **my_script** in this manner would have had no effect on its value.

```
$ cat my_script
echo $TEMPDIR
$ TEMPDIR=/Users/sam/temp my_script
/Users/sam/temp
$ echo $TEMPDIR

$
```

Keyword variables

Keyword shell variables (or simply *keyword variables*) have special meaning to the shell and usually have short, mnemonic names. When you start a shell (by logging in, for example), the shell inherits several keyword variables from the environment. Among these variables are **HOME**, which identifies your home directory, and **PATH**, which determines which directories the shell searches and in what order to locate commands that you give the shell. The shell creates and initializes (with default values) other keyword variables when you start it. Still other variables do not exist until you set them.

You can change the values of most of the keyword shell variables at any time but it is usually not necessary to change the values of keyword variables initialized in the **/etc/profile** or **/etc/csh.cshrc** systemwide startup files. If you need to change the value of a bash keyword variable, do so in one of your startup files (for bash see page 259; for tcsh see page 340). Just as you can make user-created variables global, so you can make keyword variables global; this is usually done automatically in the startup files. You can also make a keyword variable readonly.

Positional parameters

Special parameters

The names of one group of parameters do not resemble variable names. Most of these parameters have one-character names (for example, **1**, **?**, and **#**) and are referenced (as are all variables) by preceding the name with a dollar sign (**$1**, **$?**, and **$#**). The values of these parameters reflect different aspects of your ongoing interaction with the shell.

Whenever you give a command, each argument on the command line becomes the value of a *positional parameter*. Positional parameters (page 564) enable you to access command line arguments, a capability that you will often require when you write shell scripts. The set builtin (page 568) enables you to assign values to positional parameters.

Other frequently needed shell script values, such as the name of the last command executed, the number of command line arguments, and the status of the most recently executed command, are available as *special parameters*. You cannot assign values to special parameters.

USER-CREATED VARIABLES

The first line in the following example declares the variable named **person** and initializes it with the value **alex** (use **set person = alex** in tcsh):

```
$ person=alex
$ echo person
person
$ echo $person
alex
```

Because the echo builtin copies its arguments to standard output, you can use it to display the values of variables. The second line of the preceding example shows that **person** does not represent **alex**. Instead, the string **person** is echoed as **person**. The shell substitutes the value of a variable only when you precede the name of the variable with a dollar sign (**$**). The command **echo $person** displays the value of the variable **person**; it does not display **$person** because the shell does not pass **$person** to echo as an argument. Because of the leading **$**, the shell recognizes that **$person** is the name of a variable, *substitutes* the value of the variable, and passes that value to echo. The echo builtin displays the value of the variable—not its name—never knowing that you called it with a variable.

Quoting the $

You can prevent the shell from substituting the value of a variable by quoting the leading **$**. Double quotation marks do not prevent the substitution; single quotation marks or a backslash (\) do.

```
$ echo $person
alex
$ echo "$person"
alex
$ echo '$person'
$person
$ echo \$person
$person
```

SPACEs

Because they do not prevent variable substitution but do turn off the special meanings of most other characters, double quotation marks are useful when you assign values to variables and when you use those values. To assign a value that contains SPACEs or TABs to a variable, use double quotation marks around the value. Although double quotation marks are not required in all cases, using them is a good habit.

```
$ person="alex and jenny"
$ echo $person
alex and jenny
$ person=alex and jenny
bash: and: command not found
```

When you reference a variable that contains TABs or multiple adjacent SPACEs, you need to use quotation marks to preserve the spacing. If you do not quote the variable, the shell collapses each string of blank characters into a single SPACE before passing the variable to the utility:

```
$ person="alex   and   jenny"
$ echo $person
alex and jenny
$ echo "$person"
alex   and   jenny
```

When you execute a command with a variable as an argument, the shell replaces the name of the variable with the value of the variable and passes that value to the program being executed. If the value of the variable contains a special character, such as * or ?, the shell *may* expand that variable.

Pathname expansion in assignments

The first line in the following sequence of commands assigns the string **alex*** to the variable **memo**. The Bourne Again Shell does *not expand the string* because bash does not perform pathname expansion (page 133) when assigning a value to a

variable. All shells process a command line in a specific order. Within this order bash (but not tcsh) expands variables before it interprets commands. In the following echo command line, the double quotation marks quote the asterisk (*) in the expanded value of **$memo** and prevent bash from performing pathname expansion on the expanded **memo** variable before passing its value to the echo command:

```
$ memo=alex*
$ echo "$memo"
alex*
```

All shells interpret special characters as special when you reference a variable that contains an unquoted special character. In the following example, the shell expands the value of the **memo** variable because it is not quoted:

```
$ ls
alex.report
alex.summary
$ echo $memo
alex.report alex.summary
```

Here the shell expands **$memo** to **alex***, expands **alex*** to **alex.report** and **alex.summary**, and passes these two values to echo.

optional

Braces

The *$VARIABLE* syntax is a special case of the more general syntax *${VARIABLE}*, in which the variable name is enclosed by **${}**. The braces insulate the variable name. Braces are necessary when catenating a variable value with a string:

```
$ PREF=counter
$ WAY=$PREFclockwise
$ FAKE=$PREFfeit
$ echo $WAY $FAKE

$
```

The preceding example does not work as planned. Only a blank line is output because, although the symbols **PREFclockwise** and **PREFfeit** are valid variable names, they are not set. By default the shell evaluates an unset variable as an empty (null) string and displays this value (bash) or generates an error message (tcsh). To achieve the intent of these statements, refer to the **PREF** variable using braces:

```
$ PREF=counter
$ WAY=${PREF}clockwise
$ FAKE=${PREF}feit
$ echo $WAY $FAKE
counterclockwise counterfeit
```

The Bourne Again Shell refers to the arguments on its command line by position, using the special variables **$1**, **$2**, **$3**, and so forth up to **$9**. If you wish to refer to arguments past the ninth argument, you must use braces: **${10}**. The name of the command is held in **$0** (page 565).

unset: REMOVES A VARIABLE

Unless you remove a variable, it exists as long as the shell in which it was created exists. To remove the *value* of a variable but not the variable itself, set the value to null (use **set person =** in tcsh):

```
$ person=
$ echo $person

$
```

You can remove a variable with the unset builtin. To remove the variable **person**, give the following command:

```
$ unset person
```

VARIABLE ATTRIBUTES

This section discusses attributes and explains how to assign them to variables.

readonly: MAKES THE VALUE OF A VARIABLE PERMANENT

You can use the readonly builtin (not in tcsh) to ensure that the value of a variable cannot be changed. The next example declares the variable **person** to be readonly. You must assign a value to a variable *before* you declare it to be readonly; you cannot change its value after the declaration. When you attempt to unset or change the value of a readonly variable, the shell displays an error message:

```
$ person=jenny
$ echo $person
jenny
$ readonly person
$ person=helen
bash: person: readonly variable
```

If you use the readonly builtin without an argument, it displays a list of all readonly shell variables. This list includes keyword variables that are automatically set as readonly as well as keyword or user-created variables that you have declared as readonly. See "Listing variable attributes" on page 283 for an example (**readonly** and **declare –r** produce the same output).

declare AND typeset: ASSIGN ATTRIBUTES TO VARIABLES

The declare and typeset builtins (two names for the same command, neither of which is available in tcsh) set attributes and values for shell variables. Table 8-3 lists five of these attributes.

Table 8-3 Variable attributes (typeset or declare)

Attribute	Meaning
–a	Declares a variable as an array (page 558)
–f	Declares a variable to be a function name (page 314)

Table 8-3 Variable attributes (typeset or declare) (continued)

–i	Declares a variable to be of type integer (page 284)
–r	Makes a variable readonly; also readonly (page 282)
–x	Exports a variable (makes it global); also export (page 560)

The following commands declare several variables and set some attributes. The first line declares **person1** and assigns it a value of **alex**. This command has the same effect with or without the word **declare**.

```
$ declare person1=alex
$ declare -r person2=jenny
$ declare -rx person3=helen
$ declare -x person4
```

The readonly and export builtins are synonyms for the commands **declare –r** and **declare –x**, respectively. It is legal to declare a variable without assigning a value to it, as the preceding declaration of the variable **person4** illustrates. This declaration makes **person4** available to all subshells (makes it global). Until an assignment is made to the variable, it has a null value.

You can list the options to declare separately in any order. The following is equivalent to the preceding declaration of **person3**:

```
$ declare -x -r person3=helen
```

Use the **+** character in place of **–** when you want to remove an attribute from a variable. You cannot remove a readonly attribute however. After the following command is given, the variable **person3** is no longer exported but it is still readonly.

```
$ declare +x person3
```

You can also use typeset instead of declare.

Listing variable attributes

Without any arguments or options, the declare builtin lists all shell variables. The same list is output when you run set (page 568) without any arguments.

If you use a declare builtin with options but no variable names as arguments, the command lists all shell variables that have the indicated attributes set. For example, the option **–r** with declare gives a list of all readonly shell variables. This list is the same as that produced by a **readonly** command without any arguments. After the declarations in the preceding example have been given, the results are as follows:

```
$ declare -r
declare -ar BASH_VERSINFO='([0]="2" [1]="05b" [2]="0" [3]="1" ... )'
declare -ir EUID="500"
declare -ir PPID="936"
declare -r SHELLOPTS="braceexpand:emacs:hashall:histexpand:history:..."
declare -ir UID="500"
declare -r person2="jenny"
declare -rx person3="helen"
```

The first five entries are keyword variables that are automatically declared as read-only. Some of these variables are stored as integers (**–i**). The **–a** option indicates that **BASH_VERSINFO** is an array variable; the value of each element of the array is listed to the right of an equal sign.

Integer By default the values of variables are stored as strings. When you perform arithmetic on a string variable, the shell converts the variable into a number, manipulates it, and then converts it back to a string. A variable with the integer attribute is stored as an integer. Assign the integer attribute as follows:

```
$ typeset -i COUNT
```

KEYWORD VARIABLES

Keyword variables either are inherited or are declared and initialized by the shell when it starts. You can assign values to these variables from the command line or from a startup file. Typically you want these variables to apply to all subshells you start as well as to your login shell. For those variables not automatically exported by the shell, you must use export (bash, page 560) or setenv (tcsh, page 353) to make them available to child shells.

HOME: YOUR HOME DIRECTORY

By default your home directory is your working directory when you log in. Your home directory is determined when you establish your account; its name is stored in the NetInfo database (page 441).

When you log in, the shell inherits the pathname of your home directory and assigns it to the variable **HOME**. When you give a **cd** command without an argument, cd makes the directory whose name is stored in **HOME** the working directory:

```
$ pwd
/Users/alex/laptop
$ echo $HOME
/Users/alex
$ cd
$ pwd
/Users/alex
```

This example shows the value of the **HOME** variable and the effect of the cd builtin. After you execute cd without an argument, the pathname of the working directory is the same as the value of **HOME**: your home directory.

Tilde (~) The shell uses the value of **HOME** to expand pathnames that use the shorthand tilde (~) notation (page 78) to denote a user's home directory. The following example uses echo to display the value of this shortcut and then uses ls to list the files in Alex's **laptop** directory, which is a subdirectory of his home directory:

```
$ echo ~
/Users/alex
$ ls ~/laptop
tester     count      lineup
```

PATH: WHERE THE SHELL LOOKS FOR PROGRAMS

When you give the shell an absolute or relative pathname rather than a simple filename as a command, it looks in the specified directory for an executable file with the specified filename. If the file with the pathname you specified does not exist, the shell reports **command not found**. If the file exists as specified but you do not have execute permission for it, or in the case of a shell script you do not have read and execute permission for it, the shell reports **Permission denied**.

If you give a simple filename as a command, the shell searches through certain directories for the program you want to execute. It looks in several directories for a file that has the same name as the command and that you have execute permission for (a compiled program) or read and execute permission for (a shell script). The **PATH** shell variable controls this search.

The default value of **PATH** is determined when bash or tcsh is compiled. It is not set in a startup file, although it may be modified there. Normally the default specifies that the shell search several system directories used to hold common commands and then search the working directory. These system directories include **/bin** and **/usr/bin** and other directories appropriate to the local system. When you give a command, if the shell does not find the executable—and, in the case of a shell script, readable—file named by the command in any of the directories listed in **PATH**, the shell generates one of the aforementioned error messages.

Working directory

The **PATH** variable specifies the directories in the order the shell should search them. Each directory must be separated from the next by a colon. The following command sets **PATH** so that a search for an executable file starts with the **/usr/local/bin** directory. If it does not find the file in this directory, the shell first looks in **/bin**, and then in **/usr/bin**. If the search fails in those directories, the shell looks in the **bin** director, a subdirectory of the user's home directory. Finally the shell looks in the working directory. Exporting **PATH** makes its value accessible to subshells:

```
$ export PATH=/usr/local/bin:/bin:/usr/bin:~/bin:
```

A null value in the string indicates the working directory. In the preceding example, a null value (nothing between the colon and the end of the line) appears as the last element of the string. The working directory is represented by a leading colon (not recommended; see the following security tip), a trailing colon (as in the example), or two colons next to each other anywhere in the string. You can also represent the working directory explicitly with a period (**.**).

See "**PATH**" on page 359 for a tcsh example. Because Mac OS X stores many executable files in directories named **bin** (*binary*), users typically put their own executable files in their own **~/bin** directories. If you put your own **bin** directory at the end of your **PATH**, as in the preceding example, the shell looks there for any commands that it cannot find in directories listed earlier in **PATH**.

PATH and security

security Do not put the working directory first in **PATH** when security is a concern. If you are running as Superuser, you should *never* put the working directory first in **PATH**. It is common for Superuser **PATH** to omit the working directory entirely. You can always execute a file in the working directory by prepending **./** to the name: **./ls**.

Putting the working directory first in **PATH** can create a security hole. Most people type **ls** as the first command when entering a directory. If the owner of a directory places an executable file named **ls** in the directory, and the working directory appears first in a user's **PATH**, the user giving an **ls** command from the directory executes the ls program in the working directory instead of the system ls utility, possibly with undesirable results.

If you want to add directories to **PATH**, you can reference the old value of the **PATH** variable while you are setting **PATH** to a new value (but see the preceding security tip). The following command adds **/usr/X11R6/bin** to the beginning of the current **PATH** and **/usr/local/bin** and the working directory to the end:

```
$ PATH=/usr/X11R6/bin:$PATH:/usr/local/bin
```

MAIL: WHERE YOUR MAIL IS KEPT

The **MAIL** variable (**mail** under tcsh) contains the pathname of the file that holds your mail (your *mailbox*, usually **/var/mail/*name***, where ***name*** is your login name). If **MAIL** is set and **MAILPATH** (next) is not set, the shell informs you when mail arrives in the file specified by **MAIL**. In a graphical environment you can unset **MAIL** so that the shell does not display mail reminders in a terminal emulator window (assuming you are using a graphical mail program). Most Mac OS X systems do not use local files for incoming mail; mail is typically kept on a remote mail server instead. The **MAIL** variable and other mail-related shell variables do not do anything unless you have a local mail server.

The **MAILPATH** variable (not available under tcsh) contains a list of filenames separated by colons. If this variable is set, the shell informs you when any one of the files is modified (for example, when mail arrives). You can follow any of the filenames in the list with a question mark (**?**), followed by a message. The message replaces the **you have mail** message when you get mail while you are logged in.

The **MAILCHECK** variable (not available under tcsh) specifies how often, in seconds, the shell checks for new mail. The default is 60 seconds. If you set this variable to zero, the shell checks before each prompt.

PS1: USER PROMPT (PRIMARY)

The default Bourne Again Shell prompt is a dollar sign (**$**). When you run bash as **root**, you may have a pound sign (**#**) prompt. The **PS1** variable (**prompt** under tcsh, page 360) holds the prompt string that the shell uses to let you know that it is waiting for a command. When you change the value of **PS1** or **prompt**, you change the appearance of your prompt.

You can customize the prompt displayed by **PS1**. For example, the assignment

```
$ PS1="[\u@\h \W \!]$ "
```

displays the following prompt:

[user@host directory event]$

where ***user*** is the username, ***host*** is the hostname up to the first period, ***directory*** is the basename of the working directory, and ***event*** is the event number of the current command.

If you are working on more than one system, it can be helpful to incorporate the system name into your prompt. For example, you might change the prompt to the name of the system you are using, followed by a colon and a SPACE (a SPACE at the end of the prompt makes the commands that you enter after the prompt easier to read):

```
$ PS1="$(hostname): "
bravo.example.com: echo test
test
bravo.example.com:
```

Use the following command under tcsh:

```
tcsh $ set prompt = "`hostname`: "
```

The first example that follows changes the prompt to the name of the local host, a SPACE, and a dollar sign (or, if the user is running as **root**, a pound sign). The second example changes the prompt to the time followed by the name of the user. The third example changes the prompt to the one used in this book (a pound sign for **root** and a dollar sign otherwise):

```
$ PS1='\h \$ '
bravo $

$ PS1='\@ \u $ '
09:44 PM alex $

$ PS1='\$ '
$
```

Table 8-4 describes some of the symbols you can use in **PS1**. For a complete list of special characters you can use in the prompt strings, open the bash man page and search for the second occurrence of **PROMPTING** (give the command **/PROMPTING** and then press **n**).

Table 8-4 PS1 symbols

Symbol	Display in prompt
\$	**#** if the user is running as **root**; otherwise, **$**
\w	Pathname of the working directory

Table 8-4 PS1 symbols (continued)

\W	Basename of the working directory
\!	Current event (history) number (page 299)
\d	Date in Weekday Month Date format
\h	Machine hostname, without the domain
\H	Full machine hostname, including the domain
\u	Username of the current user
\@	Current time of day in 12-hour, AM/PM format
\T	Current time of day in 12-hour HH:MM:SS format
\A	Current time of day in 24-hour HH:MM format
\t	Current time of day in 24-hour HH:MM:SS format

PS2: USER PROMPT (SECONDARY)

Prompt String 2 is a secondary prompt that the shell stores in **PS2** (not under tcsh). On the first line of the next example, an unclosed quoted string follows echo. The shell assumes that the command is not finished and, on the second line, gives the default secondary prompt (>). This prompt indicates that the shell is waiting for the user to continue the command line. The shell waits until it receives the quotation mark that closes the string and then executes the command:

```
$ echo "demonstration of prompt string
> 2"
demonstration of prompt string
2
$ PS2="secondary prompt: "
$ echo "this demonstrates
secondary prompt: prompt string 2"
this demonstrates
prompt string 2
```

The second command changes the secondary prompt to **secondary prompt:** followed by a SPACE. A multiline echo demonstrates the new prompt.

PS3: MENU PROMPT

PS3 holds the menu prompt for the **select** control structure (page 552).

PS4: DEBUGGING PROMPT

PS4 holds the bash debugging symbol (page 536).

IFS: SEPARATES INPUT FIELDS (WORD SPLITTING)

The **IFS** (Internal Field Separator) shell variable (not under tcsh) specifies the characters that you can use to separate arguments on a command line and has the

Be careful when changing IFS

caution Changing **IFS** has a variety of side effects so work cautiously. You may find it useful to first save the value of **IFS** before changing it; you can easily then restore the original value if you get unexpected results. Alternatively, you can fork a new shell with a **bash** command before experimenting with **IFS**; if you get into trouble, you can **exit** back to the old shell, where **IFS** is working properly. You can also set **IFS** to its default value with the following command:

```
$ IFS=' \t\n'
```

default value of SPACE TAB NEWLINE. Regardless of the value of **IFS**, you can always use one or more SPACE or TAB characters to separate arguments on the command line, provided that these characters are not quoted or escaped. When you assign **IFS** character values, these characters can also separate fields but only if they undergo expansion. This type of interpretation of the command line is called *word splitting*.

The following example demonstrates how setting **IFS** can affect the interpretation of a command line:

```
$ a=w:x:y:z
$ cat $a
cat: w:x:y:z: No such file or directory
$ IFS=":"
$ cat $a
cat: w: No such file or directory
cat: x: No such file or directory
cat: y: No such file or directory
cat: z: No such file or directory
```

The first time cat is called, the shell expands the variable **a**, interpreting the string **w:x:y:z** as a single word to be used as the argument to cat. The cat utility cannot find a file named **w:x:y:z** and reports an error for that filename. After **IFS** is set to a colon (**:**), the shell expands the variable **a** into four words, each of which is an argument to cat. Now cat reports an error for four separate files: **w**, **x**, **y**, and **z**. Word splitting based on the colon (**:**) takes place only *after* the variable **a** is expanded.

The shell splits all *expanded* words on a command line according to the separating characters found in **IFS**. When there is no expansion, there is no splitting. Consider the following commands:

```
$ IFS="p"
$ export VAR
```

Although **IFS** is set to **p**, the **p** on the **export** command line is not expanded so the word **export** is not split.

The next example uses variable expansion in an attempt to produce an **export** command:

```
$ IFS="p"
$ aa=export
$ echo $aa
ex ort
```

This time expansion occurs so that the character **p** in the token **export** is interpreted as a separator as the preceding echo command shows. Now when you try to use the

value of the **aa** variable to export the **VAR** variable, the shell parses the **$aa VAR** command line as **ex ort VAR**. The effect is that the command line starts the ex editor with two filenames: **ort** and **VAR**.

```
$ $aa VAR
2 files to edit
"ort" [New File]
Entering Ex mode.  Type "visual" to go to Normal mode.
:q
E173: 1 more file to edit
:q
$
```

If you unset **IFS**, only SPACEs and TABs work as field separators.

CDPATH: BROADENS THE SCOPE OF cd

The **CDPATH** variable (**cdpath** under tcsh) allows you to use a simple filename as an argument to the cd builtin to change the working directory to a directory other than a child of the working directory. If you have several directories you like to work out of, this variable can speed things up and save you the tedium of using cd with longer pathnames to switch among them.

When **CDPATH** or **cdpath** is not set and you specify a simple filename as an argument to cd, cd searches the working directory for a subdirectory with the same name as the argument. If the subdirectory does not exist, cd displays an error message. When **CDPATH** or **cdpath** is set, cd searches for an appropriately named subdirectory in the directories in the **CDPATH** list. If cd finds one, that directory becomes the working directory. With **CDPATH** or **cdpath** set, you can use cd and a simple filename to change the working directory to a child of any of the directories listed in **CDPATH** or **cdpath**.

The **CDPATH** or **cdpath** variable takes on the value of a colon-separated list of directory pathnames (similar to the **PATH** variable). It is usually set in the **~/.bash_profile** (bash) or **~/.tcshrc** (tcsh) startup file with a command line such as the following:

```
export CDPATH=$HOME:$HOME/literature
```

Use the following format for tcsh:

```
setenv cdpath $HOME\:$HOME/literature
```

These commands cause cd to search your home directory, the **literature** directory, and then the working directory when you give a cd command. If you do not include the working directory in **CDPATH** or **cdpath**, cd searches the working directory if the search of all the other directories in **CDPATH** or **cdpath** fails. If you want cd to search the working directory first (which you should never do when you are logged in as **root**—refer to the tip on page 286), include a null string, represented by two colons (**::**), as the first entry in **CDPATH**:

```
export CDPATH=::$HOME:$HOME/literature
```

If the argument to the cd builtin is an absolute pathname—one starting with a slash (**/**)—the shell does not consult **CDPATH** or **cdpath**.

KEYWORD VARIABLES: A SUMMARY

Table 8-5 lists the bash keyword variables.

Table 8-5 bash keyword variables

Variable	Value
BASH_ENV	The pathname of the startup file for noninteractive shells (page 260)
CDPATH	The cd search path (page 290)
COLUMNS	The width of the display used by **select** (page 551)
FCEDIT	The name of the editor that fc uses by default (page 298)
HISTFILE	The pathname of the file that holds the history list (default: **~/.bash_history**; page 295)
HISTFILESIZE	The maximum number of entries saved in **HISTFILE** (default: 500; page 295)
HISTSIZE	The maximum number of entries saved in the history list (default: 500; page 295)
HOME	The pathname of the user's home directory (page 284); used as the default argument for cd and in tilde expansion (page 78)
IFS	Internal Field Separator (page 288); used for word splitting (page 328)
INPUTRC	The pathname of the Readline startup file (default: **~/.inputrc**; page 308)
LANG	The locale category when that category is not specifically set with a **LC_*** variable
LC_*	A group of variables that specify locale categories including **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, and **LC_NUMERIC**; use the locale builtin to display a complete list with values
LINES	The height of the display used by **select** (page 551)
MAIL	The pathname of the file that holds a user's mail (page 286)
MAILCHECK	How often, in seconds, bash checks for mail (page 286)
MAILPATH	A colon-separated list of file pathnames that bash checks for mail in (page 286)
PATH	A colon-separated list of directory pathnames that bash looks for commands in (page 285)
PROMPT_COMMAND	A command that bash executes just before it displays the primary prompt
PS1	Prompt String 1; the primary prompt (default: **'\s–\v\$ '**; page 286)
PS2	Prompt String 2; the secondary prompt (default: **'> '**; page 288)
PS3	The prompt issued by **select** (page 551)
PS4	The bash debugging symbol (page 536)
REPLY	Holds the line that read accepts (page 572); also used by **select** (page 551)

SPECIAL CHARACTERS

Table 8-6 lists most of the characters that are special to the bash and tcsh shells.

Table 8-6 Shell special characters

Character	Use
NEWLINE	Initiates execution of a command (page 269)
;	Separates commands (page 269)
( )	Groups commands (page 271) for execution by a subshell or identifies a function (page 314)
&	Executes a command in the background (pages 131 and 270)
|	Sends output through a pipe (page 270)
>	Redirects standard output (page 122)
>>	Appends standard output (page 127)
<	Redirects standard input (page 124)
<<	Here document (page 553)
*	Any string of zero or more characters in an ambiguous file reference (page 134)
?	Any single character in an ambiguous file reference (page 134)
\	Quotes the following character (page 40)
'	Quotes a string, preventing all substitution (page 40)
"	Quotes a string, allowing only variable and command substitution (pages 40 and 280)
` ... `	Performs command substitution (page 327)
[ ]	Character class in an ambiguous file reference (page 136)
$	References a variable (page 278)
. (dot builtin)	Executes a command (only at the beginning of a line, page 261)
#	Begins a comment (page 268)
{ }	Used to surround the contents of a function (page 314)
: (null builtin)	Returns *true* (page 579)
&& (Boolean AND)	Executes command on right only if command on left succeeds (returns a zero exit status, page 590)
|| (Boolean OR)	Executes command on right only if command on left fails (returns a nonzero exit status; page 590)

Table 8-6 Shell special characters (continued)

!	(Boolean NOT)	Reverses exit status of a command
$()	(not in tcsh)	Performs command substitution (preferred form; page 327)
[ ]		Evaluates an arithmetic expression (page 325)

PROCESSES

A *process* is the execution of a command by Mac OS X. The shell that starts when you log in is a command, or a process, like any other. When you give the name of a Mac OS X utility on the command line, you initiate a process. When you run a shell script, another shell process is started and additional processes are created for each command in the script. Depending on how you invoke the shell script, the script is run either by the current shell or, more typically, by a subshell (child) of the current shell. A process is not started when you run a shell builtin, such as cd.

PROCESS STRUCTURE

fork system call

Like the file structure, the process structure is hierarchical, with parents, children, and even a *root*. A parent process *forks* a child process, which in turn can fork other processes. (The term *fork* indicates that, as with a fork in the road, one process turns into two. Initially the two forks are identical except that one is identified as the parent and one as the child. You can also use the term *spawn;* the words are interchangeable.) The operating system routine, or *system call,* that creates a new process is named **fork**.

When Mac OS X begins execution when a system is started, it starts **launchd**, a single process called a *spontaneous process,* with PID number 1. This process holds the same position in the process structure as the root directory does in the file structure: It is the ancestor of all processes that the system and users work with. When the system is in multiuser mode, **launchd** runs additional processes that display the login window and may also display **login:** prompts on terminals or accept network logins. When a user responds to one of these prompts, the process that authenticates the user then starts a user environment such as the Finder for a graphical login, or the user's shell process, for a textual login.

PROCESS IDENTIFICATION

PID number

Mac OS X assigns a unique PID (process identification) number at the inception of each process. As long as a process exists, it keeps the same PID number. During one session the same process is always executing the login shell. When you fork a new process—for example, when you use an editor—the PID number of the new (child) process is different from that of its parent process. When you return to the login shell, it is still being executed by the same process and has the same PID number as when you logged in.

The following example shows that the process running the shell forked (is the parent of) the process running ps (page 133). When you call it with the **–l** option, ps displays a full listing of information about each process. The line of the ps display with **bash** in the **COMMAND** column refers to the process running the shell. The column headed by **PID** identifies the PID number. The column headed **PPID** identifies the PID number of the *parent* of the process. From the PID and PPID columns you can see that the process running the shell (PID 6476) is the parent of the process running sleep (PID 11040). The parent PID number of sleep is the same as the PID number of the shell (6476).

```
$ sleep 10 &
[1] 11040
$ ps -l
  UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT  TT       TIME COMMAND
  502  6476  6472   0  31  0    27800    864 -      Ss    p5    0:00.13 -bash
  502 11040  6476   0  31  0    27232    672 -      S     p5    0:00.01 sleep 10
```

Refer to page 819 for more information on ps and the columns it displays with the **–l** option. A second pair of **sleep** and **ps –l** commands shows that the shell is still being run by the same process but that it forked another process to run sleep:

```
$ sleep 10 &
[1] 11046
$ ps -l
  UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT  TT       TIME COMMAND
  502  6476  6472   0  31  0    27800    864 -      Ss    p5    0:00.14 -bash
  502 11046  6476   0  31  0    27232    364 -      S     p5    0:00.00 sleep 10
```

EXECUTING A COMMAND

fork and sleep

When you give the shell a command, it usually forks (spawns) a child process to execute the command. While the child process is executing the command, the parent process *sleeps*. While a process is sleeping, it does not use any computer time but remains inactive, waiting to wake up. When the child process finishes executing the command, it tells its parent of its success or failure via its exit status and then dies. The parent process (which is running the shell) wakes up and prompts for another command.

Background process

When you run a process in the background by ending a command with an ampersand (&), the shell forks a child process without going to sleep and without waiting for the child process to run to completion. The parent process, which is executing the shell, reports the job number and PID number of the child and prompts for another command. The child process runs in the background, independent of its parent.

Builtins

Although the shell forks a process to run most of the commands you give it, some commands are built into the shell. The shell does not need to fork a process to run builtins. For more information refer to "Builtins" on page 138.

Variables

Within a given process, such as your login shell or a subshell, you can declare, initialize, read, and change variables. By default however, a variable is local to a process. When a process forks a child process, the parent does not pass the value of a

variable to the child. You can make the value of a variable available to child processes (global) by using the export builtin under bash (page 560) or the setenv builtin under tcsh (page 353).

HISTORY

The history mechanism, a feature adapted from the C Shell, maintains a list of recently issued command lines, also called *events,* providing a quick way to reexecute any of the events in the list. This mechanism also enables you to execute variations of previous commands and to reuse arguments from them. You can replicate complicated commands and arguments that you used earlier in this login session or in a previous one and enter a series of commands that differ from one another in minor ways. The history list also serves as a record of what you have done. It can prove helpful when you have made a mistake and are not sure what you did or when you want to keep a record of a procedure that involved a series of commands.

The history builtin (both in bash and tcsh) displays the history list. If it does not, read on—you need to set some variables.

history can help track down mistakes

tip When you have made a command line mistake (not an error within a script or program) and are not sure what you did wrong, look at the history list to review your recent commands. Sometimes this list can help you figure out what went wrong and how to fix things.

VARIABLES THAT CONTROL HISTORY

The TC Shell's history mechanism is similar to bash's but uses different variables and has other differences. See page 342 for more information.

The value of the **HISTSIZE** variable determines the number of events preserved in the history list during a session. A value in the range of 100 to 1,000 is normal.

When you exit from the shell, the most recently executed commands are saved in the file given by the **HISTFILE** variable (the default is **~/.bash_history**). The next time you start the shell, this file initializes the history list. The value of the **HISTFILESIZE** variable determines the number of lines of history saved in **HISTFILE** (not necessarily the same as **HISTSIZE**). **HISTSIZE** holds the number of events remembered during a session, **HISTFILESIZE** holds the number remembered between sessions, and the file designated by **HISTFILE** holds the history list. See Table 8-7.

Table 8-7 History variables

Function	Variable	Default
Maximum number of events saved during a session	**HISTSIZE**	500 events
Location of the history file	**HISTFILE**	**~/.bash_history**
Maximum number of events saved between sessions	**HISTFILESIZE**	500 events

Event number The Bourne Again Shell assigns a sequential *event number* to each command line. You can display this event number as part of the bash prompt by including **\!** in **PS1** (page 286). Examples in this section show numbered prompts when they help to illustrate the behavior of a command.

Give the following command manually or place it in **~/.bash_profile** (to affect future sessions) to establish a history list of the 100 most recent events:

```
$ HISTSIZE=100
```

The following command causes bash to save the 100 most recent events across login sessions:

```
$ HISTFILESIZE=100
```

After you set **HISTFILESIZE**, you can log out and log in again, and the 100 most recent events from the previous login session will appear in your history list.

Give the command **history** to display the events in the history list. The list of events is ordered with oldest events at the top of the list. A tcsh history list includes the time the command was executed. The following history list includes a command to modify the bash prompt so that it displays the history event number. The last event in the history list is the **history** command that displayed the list.

```
32 $ history | tail
   23  PS1="\! bash$ "
   24  ls -l
   25  cat temp
   26  rm temp
   27  vim memo
   28  lpr memo
   29  vim memo
   30  lpr memo
   31  rm memo
   32  history | tail
```

As you run commands and your history list becomes longer, it may run off the top of the screen when you use the history builtin. Pipe the output of history through less to browse through it, or give the command **history 10** to look at the ten most recent commands.

REEXECUTING AND EDITING COMMANDS

You can reexecute any event in the history list. This feature can save you time, effort, and aggravation. Not having to reenter long command lines allows you to reexecute events more easily, quickly, and accurately than you could if you had to retype the entire command line. You can recall, modify, and reexecute previously executed events in three ways: You can use the fc builtin (covered next); the exclamation point commands (page 299); or the Readline Library, which uses a one-line vi- or emacs-like editor to edit and execute events (page 304).

Which method to use?

tip If you are more familiar with vi or emacs and less familiar with the C or TC Shell, use fc or the Readline Library. If you are more familiar with the C or TC Shell and less familiar with vi and emacs, use the exclamation point commands. If it is a toss-up, try the Readline Library; it will benefit you in other areas of Mac OS X more than learning the exclamation point commands will.

fc: DISPLAYS, EDITS, AND REEXECUTES COMMANDS

The fc (fix command) builtin (not in tcsh) enables you to display the history list and to edit and reexecute previous commands. It provides many of the same capabilities as the command line editors.

VIEWING THE HISTORY LIST

When you call fc with the **–l** option, it displays commands from the history list. Without any arguments, **fc –l** lists the 16 most recent commands in a numbered list, with the oldest appearing first:

```
$ fc -l
1024    cd
1025    view calendar
1026    vim letter.adams01
1027    vim letter.adams01
1028    lpr letter.adams01
1029    cd ../memos
1030    ls
1031    rm *0405
1032    fc -l
1033    cd
1034    pwd
1035    ls
...
```

The fc builtin can take zero, one, or two arguments with the **–l** option. The arguments specify the part of the history list to be displayed:

*fc –l [**first** [**last**]]*

The fc builtin lists commands beginning with the most recent event that matches ***first***. The argument can be an event number, the first few characters of the command line, or a negative number, which is taken to be the ***n***th previous command. If you provide ***last***, fc displays commands from the most recent event that matches ***first*** through the most recent event that matches ***last***. The next command displays the history list from event 1027 through event 1030:

```
$ fc -l 1027 1030
1027    vim letter.adams01
1028    lpr letter.adams01
1029    cd ../memos
1030    ls
```

The following command lists the most recent event that begins with **view** through the most recent command line that begins with **whereis**:

```
$ fc -l view pwd
1025     view calendar
1026     vim letter.adams01
1027     vim letter.adams01
1028     lpr letter.adams01
1029     cd ../memos
1030     ls
1031     rm *0405
1032     fc -l
1033     cd
1034     pwd
```

To list a single command from the history list, use the same identifier for the first and second arguments. The following command lists event 1027:

```
$ fc -l 1027 1027
1027     vim letter.adams01
```

EDITING AND REEXECUTING PREVIOUS COMMANDS

You can use fc to edit and reexecute previous commands.

*fc [–e **editor**] [**first** [**last**]]*

When you call fc with the **–e** option followed by the name of an editor, fc calls the editor with event(s) in the Work buffer. Without ***first*** and ***last***, fc defaults to the most recent command. The next example invokes the vi(m) editor to edit the most recent command:

```
$ fc -e vi
```

The fc builtin uses the stand-alone vi(m) editor. If you set the **FCEDIT** variable, you do not need to use the **–e** option to specify an editor on the command line. Because the value of **FCEDIT** has been changed to **/usr/bin/emacs** and fc has no arguments, the following command edits the most recent command with the emacs editor:

```
$ export FCEDIT=/usr/bin/emacs
$ fc
```

If you call it with a single argument, fc invokes the editor on the specified command. The following example starts the editor with event 21 in the Work buffer. When you exit from the editor, the shell executes the command:

```
$ fc 21
```

Again you can identify commands with numbers or by specifying the first few characters of the command name. The following example calls the editor to work on events from the most recent event that begins with the letters **vim** through event 206:

```
$ fc vim 206
```

Clean up the fc buffer

caution When you execute an fc command, the shell executes whatever you leave in the editor buffer, possibly with unwanted results. If you decide you do not want to execute a command, delete everything from the buffer before you exit from the editor.

REEXECUTING COMMANDS WITHOUT CALLING THE EDITOR

You can reexecute previous commands without going into an editor. If you call fc with the **–s** option, it skips the editing phase and reexecutes the command. The following example reexecutes event 1028:

```
$ fc -s 1028
lpr letter.adams01
```

The next example reexecutes the previous command:

```
$ fc -s
```

When you reexecute a command you can tell fc to substitute one string for another. The next example substitutes the string **john** for the string **adams** in event 1028 and executes the modified event:

```
$ fc -s adams=john 1028
lpr letter.john01
```

USING AN EXCLAMATION POINT (!) TO REFERENCE EVENTS

The C Shell history mechanism uses an exclamation point to reference events and is available under bash and tcsh. It is frequently more cumbersome to use than fc but nevertheless has some useful features. For example, the **!!** command reexecutes the previous event, and the **!$** token represents the last word on the previous command line.

You can reference an event by using its absolute event number, its relative event number, or the text it contains. All references to events, called event designators, begin with an exclamation point (**!**). One or more characters follow the exclamation point to specify an event.

You can put history events anywhere on a command line. To escape an exclamation point so that it is treated literally instead of as the start of a history event, precede it with a backslash (\) or enclose it within single quotation marks.

EVENT DESIGNATORS

An event designator specifies a command in the history list. See Table 8-8 on page 301 for a list of event designators.

!! reexecutes the previous event

You can always reexecute the previous event by giving a **!!** command. In the following example, event 45 reexecutes event 44:

```
44 $ ls -l text
-rw-rw-r--   1 alex alex 45 Apr 30 14:53 text
45 $ !!
ls -l text
-rw-rw-r--   1 alex alex 45 Apr 30 14:53 text
```

The **!!** command works whether or not your prompt displays an event number. As this example shows, when you use the history mechanism to reexecute an event, the shell displays the command it is reexecuting.

!*n* event number

A number following an exclamation point refers to an event. If that event is in the history list, the shell executes it. Otherwise, the shell displays an error message. A negative number following an exclamation point references an event relative to the current event.For example, the command **!–3** refers to the third preceding event. After you issue a command, the relative event number of a given event changes (event **–3** becomes event **–4**). Both of the following commands reexecute event 44:

```
51 $ !44
ls -l text
-rw-rw-r--   1 alex alex 45 Nov 30 14:53 text
52 $ !-8
ls -l text
-rw-rw-r--   1 alex alex 45 Nov 30 14:53 text
```

!*string* event text

When a string of text follows an exclamation point, the shell searches for and executes the most recent event that *began* with that string. If you enclose the string between question marks, the shell executes the most recent event that *contained* that string. The final question mark is optional if a RETURN would immediately follow it.

```
68 $ history 10
   59  ls -l text*
   60  tail text5
   61  cat text1 text5 > letter
   62  vim letter
   63  cat letter
   64  cat memo
   65  lpr memo
   66  pine jenny
   67  ls -l
   68  history
69 $ !l
ls -l
...
70 $ !lpr
lpr memo
71 $ !?letter?
cat letter
...
```

Table 8-8 Event designators

Designator	Meaning
!	Starts a history event unless followed immediately by SPACE, NEWLINE, **=**, or **(**.
!!	The previous command.
!***n***	Command number ***n*** in the history list.
!–***n***	The ***n***th preceding command.
!***string***	The most recent command line that started with ***string***.
!?***string***[?]	The most recent command that contained ***string***. The last **?** is optional.
!#	The current command (as you have it typed so far).
!{***event***}	The ***event*** is an event designator. The braces isolate ***event*** from the surrounding text. For example, **!{–3}3** is the third most recently executed command followed by a **3**.

optional

WORD DESIGNATORS

A *word designator* specifies a word or series of words from an event. Table 8-9 on page 302 lists word designators.

The words are numbered starting with 0 (the first word on the line—usually the command), continuing with 1 (the first word following the command), and going through ***n*** (the last word on the line).

To specify a particular word from a previous event, follow the event designator (such as **!14**) with a colon and the number of the word in the previous event. For example, **!14:3** specifies the third word following the command from event 14. You can specify the first word following the command (word number 1) by using a caret (**^**) and the last word by using a dollar sign (**$**). You can specify a range of words by separating two word designators with a hyphen.

```
72 $ echo apple grape orange pear
apple grape orange pear
73 $ echo !72:2
echo grape
grape
74 $ echo !72:^
echo apple
apple
75 $ !72:0 !72:$
echo pear
pear
76 $ echo !72:2-4
echo grape orange pear
grape orange pear
77 $ !72:0-$
echo apple grape orange pear
apple grape orange pear
```

As the next example shows, **!$** refers to the last word of the previous event. You can use this shorthand to edit, for example, a file you just displayed with cat:

```
$ cat report.718
...
$ vim !$
vim report.718
...
```

If an event contains a single command, the word numbers correspond to the argument numbers. If an event contains more than one command, this correspondence does not hold true for commands after the first. In the following example event 78 contains two commands separated by a semicolon so that the shell executes them sequentially; the semicolon is word number 5.

```
78 $ !72 ; echo helen jenny barbara
echo apple grape orange pear ; echo helen jenny barbara
apple grape orange pear
helen jenny barbara
79 $ echo !78:7
echo helen
helen
80 $ echo !78:4-7
echo pear ; echo helen
pear
helen
```

Table 8-9 Word designators

Designator	Meaning
n	The ***n****th* word. Word 0 is normally the command name.
^	The first word (after the command name).
$	The last word.
m*–*n	All words from word number ***m*** through word number ***n***; ***m*** defaults to 0 if you omit it (0–***n***).
n*	All words from word number ***n*** through the last word.
*****	All words except the command name. The same as **1***.
%	The word matched by the most recent ?***string***? search.

MODIFIERS

On occasion you may want to change an aspect of an event you are reexecuting. Perhaps you entered a complex command line with a typo or incorrect pathname or you want to specify a different argument. You can modify an event or a word of an event by putting one or more modifiers after the word designator, or after the event designator if there is no word designator. Each modifier must be preceded by a colon (**:**).

Substitute modifier

The *substitute modifier* is more complex than the other modifiers. The following example shows the substitute modifier correcting a typo in the previous event:

```
$ car /Users/jenny/memo.0507 /Users/alex/letter.0507
bash: car: command not found
$ !!:s/car/cat
cat /Users/jenny/memo.0507 /Users/alex/letter.0507
...
```

The substitute modifier has the following syntax:

[g]s/old/new/

where ***old*** is the original string (not a regular expression), and ***new*** is the string that replaces ***old***. The substitute modifier substitutes the first occurrence of ***old*** with ***new***. Placing a **g** before the **s** (as in *gs/old/new/*) causes a global substitution, replacing all occurrences of ***old***. The **/** is the delimiter in the examples but you can use any character that is not in either ***old*** or ***new***. The final delimiter is optional if a RETURN would immediately follow it. As with the vim Substitute command, the history mechanism replaces an ampersand (&) in ***new*** with ***old***. The shell replaces a null old string (*s//new/*) with the previous old string or string within a command that you searched for with *?string?*.

Quick substitution

An abbreviated form of the substitute modifier is *quick substitution*. Use it to reexecute the most recent event while changing some of the event text. The quick substitution character is the caret (^). For example, the command

```
$ ^old^new^
```

produces the same results as

```
$ !!:s/old/new/
```

Thus substituting **cat** for **car** in the previous event could have been entered as

```
$ ^car^cat
cat /Users/jenny/memo.0507 /Users/alex/letter.0507
...
```

You can omit the final caret if it would be followed immediately by a RETURN. As with other command line substitutions, the shell displays the command line as it appears after the substitution.

Other modifiers

Modifiers (other than the substitute modifier) perform simple edits on the part of the event that has been selected by the event designator and the optional word designators. You can use multiple modifiers, each preceded by a colon (**:**).

The following series of commands uses ls to list the name of a file, repeats the command without executing it (**p** modifier), and repeats the last command, removing the last part of the pathname (**h** modifier) again without executing it:

```
$ ls /etc/postfix/aliases
/etc/postfix/aliases
$ !!:p
ls /etc/postfix/aliases
$ !!:h:p
ls /etc/postfix
$
```

Table 8-10 lists event modifiers other than the substitute modifier.

Table 8-10 Modifiers

Modifier	Function
e (extension)	Removes all but the filename extension
h (head)	Removes the last part of a pathname
p (print-not)	Displays the command, but does not execute it
q (quote)	Quotes the substitution to prevent further substitutions on it
r (root)	Removes the filename extension
t (tail)	Removes all elements of a pathname except the last
x	Like **q** but quotes each word in the substitution individually

THE READLINE LIBRARY

Command line editing under the Bourne Again Shell is implemented through the *Readline Library,* which is available to any application written in C. Any application that uses the Readline Library supports line editing that is consistent with that provided by bash. Programs that use the Readline Library, including bash, read **~/.inputrc** (page 308) for key binding information and configuration settings. The **––noediting** command line option turns off command line editing in bash.

vi mode

You can choose one of two editing modes when using the Readline Library in bash: emacs or vi(m). Both modes provide many of the commands available in the stand-alone versions of the vi(m) and emacs editors. You can also use the ARROW keys to move around. Up and down movements move you backward and forward through the history list. In addition, Readline provides several types of interactive word completion (page 306). The default mode is emacs; you can switch to vi mode with the following command:

```
$ set -o vi
```

emacs mode

The next command switches back to emacs mode:

```
$ set -o emacs
```

vi EDITING MODE

Before you start make sure you are in vi mode.

When you enter bash commands while in vi editing mode, you are in Input mode (page 149). As you enter a command, if you discover an error before you press RETURN, you can press ESCAPE to switch to vi Command mode. This setup is different from the stand-alone vi(m) editor's initial mode. While in Command mode you can use many vi(m) commands to edit the command line. It is as though you were using

vi(m) to edit a copy of the history file with a screen that has room for only one command. When you use the **k** command or the UP ARROW to move up a line, you access the previous command. If you then use the **j** command or the DOWN ARROW to move down a line, you will return to the original command. To use the **k** and **j** keys to move between commands you must be in Command mode; you can use the ARROW keys in both Command and Input modes.

The stand-alone editor starts in Command mode

tip The stand-alone vim editor starts in Command mode, whereas the command line vi(m) editor starts in Input mode. If commands display characters and do not work properly, you are in Input mode. Press ESCAPE and enter the command again.

In addition to cursor-positioning commands, you can use the search-backward (**?**) command followed by a search string to look *back* through your history list for the most recent command containing that string. If you have moved back in your history list, use a forward slash (**/**) to search *forward* toward your most recent command. Unlike the search strings in the stand-alone vi(m) editor, these search strings cannot contain regular expressions. You can, however, start the search string with a caret (**^**) to force the shell to locate commands that start with the search string. As in vi(m), pressing **n** after a successful search looks for the next occurrence of the same string.

You can also access events in the history list by using event numbers. While you are in Command mode (press ESCAPE), enter the event number followed by a **G** to go to the command with that event number.

When you use **/**, **?**, or **G** to move to a command line, you are in Command mode, not Input mode. Now you can edit the command as you like or press RETURN to execute it.

Once the command you want to edit is displayed, you can modify the command line using vi(m) Command mode editing commands such as **x** (delete character), **r** (replace character), **~** (change case), and **.** (repeat last change). To change to Input mode, use an Insert (**i**, **I**), Append (**a**, **A**), Replace (**R**), or Change (**c**, **C**) command. You do not have to return to Command mode to run a command; simply press RETURN, even if the cursor is in the middle of the command line.

Refer to page 191 for a summary of vim commands.

emacs EDITING MODE

Unlike the vi(m) editor, emacs is modeless. You need not switch between Command mode and Input mode because most emacs commands are control characters (page 208), allowing emacs to distinguish between input and commands. Like vi(m), the emacs command line editor provides commands for moving the cursor on the command line and through the command history list and for modifying part or all of a command. The emacs command line editor commands differ in a few cases from the commands in the stand-alone emacs editor.

In emacs you perform cursor movement by using both CONTROL and ESCAPE commands. To move the cursor one character backward on the command line, press CONTROL-B. Press CONTROL-F to move one character forward. As in vi, you may precede these movements with counts. To use a count you must first press ESCAPE; otherwise, the numbers you type will appear on the command line.

Like vi(m), emacs provides word and line movement commands. To move backward or forward one word on the command line, press ESCAPE **b** or ESCAPE **f**. To move several words by using a count, press ESCAPE followed by the number and the appropriate escape sequence. To get to the beginning of the line, press CONTROL-A; to the end of the line, press CONTROL-E; and to the next instance of the character ***c***, press CONTROL-X CONTROL-F followed by ***c***.

You can add text to the command line by moving the cursor to the correct place and typing the desired text. To delete text, move the cursor just to the right of the characters that you want to delete and press the erase key (page 25) once for each character you want to delete.

CONTROL-D can terminate your screen session

If you want to delete the character directly under the cursor, press CONTROL-D. If you enter CONTROL-D at the beginning of the line, it may terminate your shell session.

If you want to delete the entire command line, type the line kill character (page 25). You can type this character while the cursor is anywhere in the command line. If you want to delete from the cursor to the end of the line, use CONTROL-K.

Refer to page 243 for a summary of emacs commands.

READLINE COMPLETION COMMANDS

You can use the TAB key to complete words you are entering on the command line. This facility, called *completion*, works in both vi and emacs editing modes and is similar to the completion facility available in tcsh. Several types of completion are possible, and which one you use depends on which part of a command line you are typing when you press TAB.

COMMAND COMPLETION

If you are typing the name of a command (the first word on the command line), pressing TAB results in *command completion*. That is, bash looks for a command whose name starts with the part of the word you have typed. If no command starts with what you have entered, bash beeps. If there is one such command, bash completes the command name for you. If there is more than one choice, bash does nothing in vi mode and beeps in emacs mode. Pressing TAB a second time causes bash to display a list of commands whose names start with the prefix you typed and allows you to finish typing the command name.

In the following example, the user types **bz** and presses TAB. The shell beeps (the user is in emacs mode) to indicate that several commands start with the letters **bz**. The

user enters another TAB to cause the shell to display a list of commands that start with **bz** followed by the command line as the user had entered it so far:

```
$ bz ⟶ TAB (beep) ⟶ TAB
bzcat         bzdiff        bzip2         bzless
bzcmp         bzgrep        bzip2recover  bzmore
$ bz■
```

Next the user types **c** and presses TAB twice. The shell displays the two commands that start with **bzc**. The user types **a** followed by TAB and the shell then completes the command because only one command starts with **bzca**.

```
$ bzc ⟶ TAB (beep) ⟶ TAB
bzcat  bzcmp
$ bzca ⟶ TAB ⟶t ■
```

PATHNAME COMPLETION

Pathname completion, which also uses TABs, allows you to type a portion of a pathname and have bash supply the rest. If the portion of the pathname that you have typed is sufficient to determine a unique pathname, bash displays that pathname. If more than one pathname would match it, bash completes the pathname up to the point where there are choices so that you can type more.

When you are entering a pathname, including a simple filename, and press TAB, the shell beeps (if the shell is in emacs mode—in vi mode there is no beep). It then extends the command line as far as it can.

```
$ cat films/dar ⟶ TAB (beep) cat films/dark_■
```

In the **films** directory every file that starts with **dar** has **k_** as the next characters, so bash cannot extend the line further without making a choice among files. You are left with the cursor just past the _ character. At this point you can continue typing the pathname or press TAB twice. In the latter case bash beeps, displays your choices, redisplays the command line, and again leaves the cursor just after the _ character.

```
$ cat films/dark_ ⟶ TAB (beep) ⟶ TAB
dark_passage  dark_victory
$ cat films/dark_■
```

When you add enough information to distinguish between the two possible files and press TAB, bash displays the unique pathname. If you enter **p** followed by TAB after the _ character, the shell completes the command line:

```
$ cat films/dark_p ⟶ TAB ⟶ assage
```

Because there is no further ambiguity, the shell appends a SPACE so you can finish typing the command line or just press RETURN to execute the command. If the complete pathname is that of a directory, bash appends a slash (/) in place of a SPACE.

VARIABLE COMPLETION

When typing a variable name, pressing TAB results in *variable completion*, where bash tries to complete the name of the variable. In case of an ambiguity, pressing TAB twice displays a list of choices:

```
$ echo $HO ⟶ TAB ⟶ TAB
$HOME       $HOSTNAME   $HOSTTYPE
$ echo $HOM ⟶ TAB ⟶ E
```

Pressing RETURN executes the command

caution Pressing RETURN causes the shell to execute the command regardless of where the cursor is on the command line.

.inputrc: CONFIGURING READLINE

The Bourne Again Shell and other programs that use the Readline Library read the file specified by the **INPUTRC** environment variable to obtain initialization information. If **INPUTRC** is not set, these programs read the **~/.inputrc** file. They ignore lines of **.inputrc** that are blank or that start with a pound sign (**#**).

VARIABLES

You can set variables in **.inputrc** to control the behavior of the Readline Library using the following syntax:

set variable value

Table 8-11 lists some variables and values you can use. See **Readline Variables** in the bash man or info page for a complete list.

Table 8-11 Readline variables

Variable	Effect
editing-mode	Set to **vi** to start Readline in **vi** mode. Set to **emacs** to start Readline in emacs mode (the default). Similar to the **set –o vi** and **set –o emacs** shell commands (page 304).
horizontal-scroll-mode	Set to **on** to cause long lines to extend off the right edge of the display area. Moving the cursor to the right when it is at the right edge of the display area shifts the line to the left so you can see more of the line. You can shift the line back by moving the cursor back past the left edge. The default value is **off**, which causes long lines to wrap onto multiple lines of the display.
mark-directories	Set to **off** to cause Readline not to place a slash (/) at the end of directory names it completes. Normally it is **on**.
mark-modified-lines	Set to **on** to cause Readline to precede modified history lines with an asterisk. The default value is **off**.

KEY BINDINGS

You can specify bindings that map keystroke sequences to Readline commands, allowing you to change or extend the default bindings. As in emacs, the Readline Library includes many commands that are not bound to a keystroke sequence. To use an unbound command, you must map it using one of the following forms:

keyname: command_name
"keystroke_sequence": command_name

In the first form, you spell out the name for a single key. For example, CONTROL-U would be written as **control-u**. This form is useful for binding commands to single keys.

In the second form, you specify a string that describes a sequence of keys that will be bound to the command. You can use the emacs-style backslash escape sequences to represent the special keys CONTROL (**\C**), META (**\M**), and ESCAPE (**\e**). Specify a backslash by escaping it with another backslash: ****. Similarly, a double or single quotation mark can be escaped with a backslash: **\"** or **\'**.

The **kill-whole-line** command, available in emacs mode only, deletes the current line. Put the following command in **.inputrc** to bind the **kill-whole-line** command (which is unbound by default) to the keystroke sequence CONTROL-R.

```
control-r: kill-whole-line
```

bind Give the command **bind –P** to display a list of all Readline commands. If a command is bound to a key sequence, that sequence is shown. Commands you can use in vi mode start with **vi**. For example, **vi-next-word** and **vi-prev-word** move the cursor to the beginning of the next and previous words, respectively. Commands that do not begin with **vi** are generally available in emacs mode.

Use **bind –q** to determine which key sequence is bound to a command:

```
$ bind -q kill-whole-line
kill-whole-line can be invoked via "\C-r".
```

You can also bind text by enclosing it within double quotation marks (emacs mode only):

```
"QQ": "The Mac OS X Operating System"
```

This command causes bash to insert the string **The Mac OS X Operating System** when you type **QQ**.

CONDITIONAL CONSTRUCTS

You can conditionally select parts of the **.inputrc** file using the **$if** directive. The syntax of the conditional construct is

*$if [**test**[=**value**]]*
commands
[$else
commands
$endif

where ***test*** is **mode**, **term**, or **bash**. If ***test*** equals ***value*** or if ***test*** is *true*, this structure executes the first set of ***commands***. If ***test*** does not equal ***value*** or if ***test*** is *false*, it executes the second set of ***commands*** if they are present or exits from the structure if they are not present.

The power of the **$if** directive lies in the three types of tests it can perform.

1. You can test to see which mode is currently set.

   ```
   $if mode=vi
   ```

   The preceding test is *true* if the current Readline mode is **vi** and *false* otherwise. You can test for **vi** or **emacs**.

2. You can test the type of terminal.

   ```
   $if term=xterm
   ```

   The preceding test is *true* if the **TERM** variable is set to **xterm**. You can test for any value of **TERM**.

3. You can test the application name.

   ```
   $if bash
   ```

   The preceding test is *true* when you are running bash and not another program that uses the Readline Library. You can test for any application name.

These tests can customize the Readline Library based on the current mode, the type of terminal, and the application you are using. They give you a great deal of power and flexibility when using the Readline Library with bash and other programs.

The following commands in **.inputrc** cause CONTROL-Y to move the cursor to the beginning of the next word regardless of whether bash is in vi or emacs mode:

```
$ cat ~/.inputrc
set editing-mode vi
$if mode=vi
       "\C-y": vi-next-word
    $else
       "\C-y": forward-word
$endif
```

Because bash reads the preceding conditional construct when it is started, you must set the editing mode in **.inputrc**. Changing modes interactively using set will not change the binding of CONTROL-Y.

For more information on the Readline Library, open the bash man page and give the command **/^READLINE**, which searches for the word **READLINE** at the beginning of a line.

If Readline commands do not work, log out and log in again

tip The Bourne Again Shell reads **~/.inputrc** when you log in. After you make changes to this file, you should log out and log in again before testing the changes.

ALIASES

An *alias* is a (usually short) name that the shell translates into another (usually longer) name or (complex) command. Aliases allow you to define new commands by substituting a string for the first token of a simple command. They are typically placed in the **~/.bashrc** (bash) or **~/.tcshrc** (tcsh) startup files so that they are available to interactive subshells.

Under bash the syntax of the alias builtin is

*alias [**name**[=**value**]]*

Under tcsh the syntax is

*alias [**name**[**value**]]*

In the bash syntax there are no SPACEs around the equal sign. If ***value*** contains SPACEs or TABs, you must enclose ***value*** between quotation marks. Unlike aliases under tcsh, a bash alias does not accept an argument from the command line in ***value***. Use a bash function (page 314) when you need to use an argument.

An alias does not replace itself, which avoids the possibility of infinite recursion in handling an alias such as the following:

```
$ alias ls='ls -F'
```

You can nest aliases. Aliases are disabled for noninteractive shells (that is, shell scripts). To see a list of the current aliases, give the command **alias**. To view the alias for a particular name, use alias followed by the name and nothing else. You can use the unalias builtin to remove an alias.

When you give an alias builtin without any arguments, the shell displays a list of all defined aliases:

```
$ alias
alias ll='ls -l'
alias l='ls -ltr'
alias ls='ls -F'
alias zap='rm -i'
```

Give an **alias** command to see which aliases are in effect. You can delete the aliases you do not want from the appropriate startup file.

SINGLE VERSUS DOUBLE QUOTATION MARKS IN ALIASES

The choice of single or double quotation marks is significant in the alias syntax when the alias includes variables. If you enclose ***value*** within double quotation marks, any variables that appear in ***value*** are expanded when the alias is created. If you enclose ***value*** within single quotation marks, variables are not expanded until the alias is used. The following example illustrates the difference.

The **PWD** keyword variable holds the pathname of the working directory. Alex creates two aliases while he is working in his home directory. Because he uses double quotation marks when he creates the **dirA** alias, the shell substitutes the value of the working directory when he creates this alias. The **alias dirA** command displays the **dirA** alias and shows that the substitution has already taken place:

```
$ echo $PWD
/Users/alex
$ alias dirA="echo Working directory is $PWD"
$ alias dirA
alias dirA='echo Working directory is /Users/alex'
```

When Alex creates the **dirB** alias, he uses single quotation marks, which prevent the shell from expanding the **$PWD** variable. The **alias dirB** command shows that the **dirB** alias still holds the unexpanded **$PWD** variable:

```
$ alias dirB='echo Working directory is $PWD'
$ alias dirB
alias dirB='echo Working directory is $PWD'
```

After creating the **dirA** and **dirB** aliases, Alex uses cd to make **cars** his working directory and gives each of the aliases as commands. The alias that he created with double quotation marks displays the name of the directory that he created the alias in as the working directory (which is wrong) and the **dirB** alias displays the proper name of the working directory:

```
$ cd cars
$ dirA
Working directory is /Users/alex
$ dirB
Working directory is /Users/alex/cars
```

How to prevent the shell from invoking an alias

tip The shell checks only simple, unquoted commands to see if they are aliases. Commands given as relative or absolute pathnames and quoted commands are not checked. When you want to give a command that has an alias but do not want to use the alias, precede the command with a backslash, specify the command's absolute pathname, or give the command as ***./command***.

EXAMPLES OF ALIASES

The following alias allows you to type **r** to repeat the previous command or **r abc** to repeat the last command line that began with **abc**:

```
$ alias r='fc -s'
```

If you use the command **ls –ltr** frequently, you can create an alias that substitutes **ls –ltr** when you give the command **l**:

```
$ alias l='ls -ltr'
$ l
total 41
-rw-r--r--  1 alex   alex   30015 Mar  1 2004 flute.ps
-rw-r-----  1 alex   alex    3089 Feb 11 2005 XTerm.ad
-rw-r--r--  1 alex   alex     641 Apr  1 2005 fixtax.icn
```

```
-rw-r--r--  1 alex   alex     484 Apr  9 2005 maptax.icn
drwxrwxr-x  2 alex   alex    1024 Aug  9 17:41 Tiger
drwxrwxr-x  2 alex   alex    1024 Sep 10 11:32 testdir
-rwxr-xr-x  1 alex   alex     485 Oct 21 08:03 floor
drwxrwxr-x  2 alex   alex    1024 Oct 27 20:19 Test_Emacs
```

Another common use of aliases is to protect yourself from mistakes. The following example substitutes the interactive version of the rm utility when you give the command **zap**:

```
$ alias zap='rm -i'
$ zap f*
rm: remove 'fixtax.icn'? n
rm: remove 'flute.ps'? n
rm: remove 'floor'? n
```

The **–i** option causes rm to ask you to verify each file that would be deleted, to help you avoid accidentally deleting the wrong file. You can also alias rm with the **rm –i** command: **alias rm='rm –i'**.

The aliases in the next example cause the shell to substitute **ls –l** each time you give an **ll** command and **ls –F** when you use **ls:**

```
$ alias ls='ls -F'
$ alias ll='ls -l'
$ ll
total 41
drwxrwxr-x  2 alex   alex    1024 Oct 27 20:19 Test_Emacs/
drwxrwxr-x  2 alex   alex    1024 Aug 9 17:41 Tiger/
-rw-r-----  1 alex   alex    3089 Feb 11 2005 XTerm.ad
-rw-r--r--  1 alex   alex     641 Apr 1 2005 fixtax.icn
-rw-r--r--  1 alex   alex    30015 Mar 1 2004 flute.ps
-rwxr-xr-x  1 alex   alex     485 Oct 21 08:03 floor*
-rw-r--r--  1 alex   alex     484 Apr 9 2005 maptax.icn
drwxrwxr-x  2 alex   alex    1024 Sep 10 11:32 testdir/
```

The **–F** option causes ls to print a slash (**/**) at the end of directory names and an asterisk (*****) at the end of the names of executable files. In this example, the string that replaces the alias **ll** (**ls –l**) itself contains an alias (**ls**). When it replaces an alias with its value, the shell looks at the first word of the replacement string to see whether it is an alias. In the preceding example, the replacement string contains the alias **ls**, so a second substitution occurs to produce the final command **ls –F –l**. (To avoid a *recursive plunge,* the **ls** in the replacement text, although an alias, is not expanded a second time.)

When given a list of aliases without the ***=value*** or ***value*** field, the alias builtin responds by displaying the value of each defined alias. The alias builtin reports an error if an alias has not been defined:

```
$ alias ll l ls zap wx
alias ll='ls -l'
alias l='ls -ltr'
alias ls='ls -F'
alias zap='rm -i'
bash: alias: wx: not found
```

You can avoid alias substitution by preceding the aliased command with a backslash (\):

```
$ \ls
Test_Emacs XTerm.ad  flute.ps  maptax.icn
Tiger    fixtax.icn  floor     testdir
```

Because the replacement of an alias name with the alias value does not change the rest of the command line, any arguments are still received by the command that gets executed:

```
$ ll f*
-rw-r--r--  1 alex   alex     641 Apr  1 2005 fixtax.icn
-rw-r--r--  1 alex   alex   30015 Mar  1 2004 flute.ps
-rwxr-xr-x  1 alex   alex     485 Oct 21 08:03 floor*
```

You can remove an alias with the unalias builtin. When the **zap** alias is removed, it is no longer displayed with the alias builtin and its subsequent use results in an error message:

```
$ unalias zap
$ alias
alias ll='ls -l'
alias l='ls -ltr'
alias ls='ls -F'
$ zap maptax.icn
bash: zap: command not found
```

FUNCTIONS

A shell function (tcsh does not have functions) is similar to a shell script in that it stores a series of commands for execution at a later time. However, because the shell stores a function in the computer's main memory (RAM) instead of in a file on the disk, the shell can access it more quickly than the shell can access a script. The shell also preprocesses (parses) a function so that it starts up more quickly than a script. Finally the shell executes a shell function in the same shell that called it. If you define too many functions, the overhead of starting a subshell (as when you run a script) can become unacceptable.

You can declare a shell function in the **~/.bash_profile** startup file, in the script that uses it, or directly from the command line. You can remove functions with the unset builtin. The shell does not keep functions once you log out.

Removing variables and functions

tip If you have a shell variable and a function with the same name, using unset removes the shell variable. If you then use unset again with the same name, it removes the function.

The syntax that declares a shell function is

[function] ***function-name*** *()*
{
    commands
}

where the word *function* is optional, ***function-name*** is the name you use to call the function, and ***commands*** comprise the list of commands the function executes when you call it. The ***commands*** can be anything you would include in a shell script, including calls to other functions.

The first brace ({) can appear on the same line as the function name. Aliases and variables are expanded when a function is read, not when it is executed. You can use the **break** statement (page 543) within a function to terminate its execution.

Shell functions are useful as a shorthand as well as to define special commands. The following function starts a process named **process** in the background, with the output normally displayed by **process** being saved in **.process.out**:

```
start_process() {
process > .process.out 2>&1 &
}
```

The next example shows how to create a simple function that displays the date, a header, and a list of the people who are using the system. This function runs the same commands as the **whoson** script described on page 265. In this example the function is being entered from the keyboard. The greater-than (**>**) signs are secondary shell prompts (**PS2**); do not enter them.

```
$ function whoson ()
> {
>   date
>   echo "Users Currently Logged On"
>   who
> }

$ whoson
Sun Aug  7 15:44:58 PDT 2005
Users Currently Logged On
zach     console  May 5 22:18
zach     ttyp1    May 5 22:20
zach     ttyp2    May 5 22:21 (bravo.example.co)
```

Functions in startup files

If you want to have the **whoson** function always be available without having to enter it each time you log in, put its definition in **~/.bash_profile**. Then run **.bash_profile**, using the **.** (dot) command to put the changes into effect immediately:

```
$ cat ~/.bash_profile
export TERM=vt100
stty kill '^u'
whoson ()
{
    date
    echo "Users Currently Logged On"
    who
}
$ . ~/.bash_profile
```

You can specify arguments when you call a function. Within the function these arguments are available as positional parameters (page 564). The following example shows the **arg1** function entered from the keyboard.

```
$ arg1 ( ) {
> echo "$1"
> }
$ arg1 first_arg
first_arg
```

See the function **switch** () on page 261 for another example of a function. "Functions" on page 561 discusses the use of local and global variables within a function.

optional The following function allows you to export variables using tcsh syntax. The env builtin lists all environment variables and their values and verifies that **setenv** worked correctly:

```
$ cat .bash_profile
...
# setenv - keep tcsh users happy
function setenv()
{
    if [ $# -eq 2 ]
        then
            eval $1=$2
            export $1
        else
            echo "Usage: setenv NAME VALUE" 1>&2
    fi
}
$ . ~/.bash_profile
$ setenv TCL_LIBRARY /usr/local/lib/tcl
$ env | grep TCL_LIBRARY
TCL_LIBRARY=/usr/local/lib/tcl
```

eval The **$#** special parameter (page 564) takes on the value of the number of command line arguments. This function uses the eval builtin to force bash to scan the command **$1=$2** *twice*. Because **$1=$2** begins with a dollar sign (**$**), the shell treats the entire string as a single token—a command. With variable substitution performed, the command name becomes **TCL_LIBRARY=/usr/local/lib/tcl**, which results in an error. Using eval, a second scanning splits the string into the three desired tokens, and the correct assignment occurs. See page 374 for more information on eval.

CONTROLLING bash FEATURES AND OPTIONS

This section explains how to control bash features and options using command line options and the set and shopt builtins.

COMMAND LINE OPTIONS

Two kinds of command line options are available: short and long. Short options consist of a hyphen followed by a letter; long options have two hyphens followed by multiple characters. Long options must appear before short options on a command line that calls bash. Table 8-12 lists some commonly used command line options.

Table 8-12 Command line options

Option	Explanation	Syntax
Help	Displays a usage message.	**––help**
No edit	Prevents users from using the Readline Library (page 304) to edit command lines in an interactive shell.	**––noediting**
No profile	Prevents reading these startup files (page 259): **/etc/profile**, **~/.bash_profile**, **~/.bash_login**, and **~/.profile**.	**––noprofile**
No rc	Prevents reading the **~/.bashrc** startup file (page 260). This option is on by default if the shell is called as **sh**.	**––norc**
POSIX	Runs bash in POSIX mode.	**––posix**
Version	Displays bash version information and exits.	**––version**
Login	Causes bash to run as though it were a login shell.	**–l** (lowercase "l")
shopt	Runs a shell with the ***opt*** shopt option (page 318). A **–O** (uppercase "O") sets the option; **+O** unsets it.	[±]**O** [***opt***]
End of options	On the command line, signals the end of options. Subsequent tokens are treated as arguments even if they begin with a hyphen (**–**).	**––**

SHELL FEATURES

You can control the behavior of the Bourne Again Shell by turning features on and off. Different features use different methods to turn features on and off. The set builtin controls one group of features, while the shopt builtin controls another group. You can also control many features from the command line you use to call bash.

Features, options, variables?

tip To avoid confusing terminology, this book refers to the various shell behaviors that you can control as *features.* The bash info page refers to them as "options" and "values of variables controlling optional shell behavior."

set ±o: TURNS SHELL FEATURES ON AND OFF

The set builtin (there is a set builtin in tcsh, but it works differently), when used with the **–o** or **+o** option, enables, disables, and lists certain bash features. For example, the following command turns on the **noclobber** feature (page 125):

```
$ set -o noclobber
```

You can turn this feature off (the default) by giving the command

```
$ set +o noclobber
```

The command **set –o** without an option lists each of the features controlled by set followed by its state (on or off). The command **set +o** without an option lists the same features in a form that you can use as input to the shell. Table 8-13 lists bash features.

shopt: TURNS SHELL FEATURES ON AND OFF

The shopt (shell option) builtin (not available in tcsh) enables, disables, and lists certain bash features that control the behavior of the shell. For example, the following command causes bash to include filenames that begin with a period (**.**) when it expands ambiguous file references (the **–s** stands for *set*):

```
$ shopt -s dotglob
```

You can turn this feature off (the default) by giving the command (the **–u** stands for *unset*)

```
$ shopt -u dotglob
```

The shell displays how a feature is set if you give the name of the feature as the only argument to shopt:

```
$ shopt dotglob
dotglob         off
```

The command **shopt** without any options or arguments lists the features controlled by shopt and their state. The command **shopt –s** without an argument lists the features controlled by shopt that are set or on. The command **shopt –u** lists the features that are unset or off. Table 8-13 lists bash features.

Setting set ±o features using shopt

tip You can use shopt to set/unset features that are otherwise controlled by **set ±o**. Use the regular shopt syntax with **–s** or **–u** and include the **–o** option. For example, the following command turns on the **noclobber** feature:

```
$ shopt -o -s noclobber
```

Table 8-13 bash features

Feature	Description	Syntax	Alternate syntax
allexport	Automatically exports all variables and functions that you create or modify after giving this command.	**set –o allexport**	**set –a**
braceexpand	Causes bash to perform brace expansion (the default; page 323).	**set –o braceexpand**	**set –B**
cdspell	Corrects minor spelling errors in directory names used as arguments to cd.	**shopt –s cdspell**	
cmdhist	Saves all lines of a multiline command in the same history entry, adding semicolons as needed.	**shopt –s cmdhist**	
dotglob	Causes shell special characters (wildcards; page 133) in an ambiguous file reference to match a leading period in a filename. By default special characters do not to match a leading period. You must always specify the filenames **.** and **..** explicitly because no pattern ever matches them.	**shopt –s dotglob**	
emacs	Specifies emacs editing mode for command line editing (the default; page 305).	**set –o emacs**	
errexit	Causes bash to exit when a simple command (not a control structure) fails.	**set –o errexit**	**set –e**
execfail	Causes a shell script to continue running when it cannot find the file that is given as an argument to exec. By default a script terminates when exec cannot find the file that is given as its argument.	**shopt –s execfail**	
expand_aliases	Causes aliases (page 311) to be expanded (by default it is on for interactive shells and off for noninteractive shells).	**shopt –s expand_alias**	
hashall	Causes bash to remember where commands it has found using **PATH** (page 285) are located (default).	**set –o hashall**	**set –h**
histappend	Causes bash to append the history list to the file named by **HISTFILE** (page 295) when the shell exits. By default bash overwrites this file.	**shopt –s histappend**	

Table 8-13 bash features (continued)

histexpand	Causes the history mechanism (which uses exclamation points; page 299) to work (default). Turn this feature off to turn off history expansion.	**set –o histexpand**	**set –H**
history	Enable command history (on by default; page 295).	**set –o history**	
ignoreeof	Specifies that bash must receive ten EOF characters before it exits. Useful on noisy dial-up lines.	**set –o ignoreeof**	
monitor	Enables job control (on by default, page 272).	**set –o monitor**	**set –m**
nocaseglob	Causes ambiguous file references (page 133) to match filenames without regard to case (off by default).	**shopt –s nocaseglob**	
noclobber	Helps prevent overwriting files (off by default; page 125).	**set –o noclobber**	**set –C**
noglob	Disables pathname expansion (off by default; page 133).	**set –o noglob**	**set –f**
notify	With job control (page 272) enabled, reports the termination status of background jobs immediately. The default behavior is to display the status just before the next prompt.	**set –o notify**	**set –b**
nounset	Displays an error and exits from a shell script when you use an unset variable in an interactive shell. The default is to display a null value for an unset variable.	**set –o nounset**	**set –u**
nullglob	Causes bash to expand ambiguous file references (page 133) that do not match a filename to a null string. By default bash passes these file references without expanding them.	**shopt –s nullglob**	
posix	Runs bash in POSIX mode.	**set –o posix**	
verbose	Displays command lines as bash reads them.	**set –o verbose**	**set –v**
vi	Specifies vi editing mode for command line editing (page 304).	**set –o vi**	

Table 8-13 bash features (continued)

xpg_echo	Causes the echo builtin to expand backslash escape sequences without the need for the **–e** option (page 548).	**shopt –s xpg_echo**	
xtrace	Turns on shell debugging (page 536).	**set –o xtrace**	**set –x**

PROCESSING THE COMMAND LINE

Whether you are working interactively or running a shell script, bash needs to read a command line before it can start processing it—bash always reads at least one line before processing a command. Some bash builtins, such as **if** and **case**, as well as functions and quoted strings, span multiple lines. When bash recognizes a command that covers more than one line, it reads the entire command before processing it. In interactive sessions bash prompts you with the secondary prompt (**PS2**, **>** by default; page 288) as you type each line of a multiline command until it recognizes the end of the command:

```
$ echo 'hi
> end'
hi
end
$ function hello () {
> echo hello there
> }
$
```

After reading a command line, bash applies history expansion and alias substitution to the line.

HISTORY EXPANSION

"Reexecuting and Editing Commands" on page 296 discusses the commands you can give to modify and reexecute command lines from the history list. History expansion is the process that bash uses to turn a history command into an executable command line. For example, when you give the command **!!**, history expansion changes that command line so it is the same as the previous one. History expansion is turned on by default for interactive shells; **set +o histexpand** turns it off. History expansion does not apply to noninteractive shells (shell scripts).

ALIAS SUBSTITUTION

Aliases (page 311) substitute a string for the first word of a simple command. By default aliases are turned on for interactive shells and off for noninteractive shells. Give the command **shopt –u expand_aliases** to turn aliases off.

PARSING AND SCANNING THE COMMAND LINE

After processing history commands and aliases, bash does not execute the command immediately. One of the first things the shell does is to *parse* (isolate strings of characters in) the command line into tokens or words. The shell then scans each token for special characters and patterns that instruct the shell to take certain actions. These actions can involve substituting one word or words for another. When the shell parses the following command line, it breaks it into three tokens (**cp**, **~/letter**, and **.**):

```
$ cp ~/letter .
```

After separating tokens and before executing the command, the shell scans the tokens and performs *command line expansion*.

COMMAND LINE EXPANSION

In both interactive and noninteractive use, the shell transforms the command line using *command line expansion* before passing the command line to the program being called. You can use a shell without knowing much about command line expansion, but you can use what a shell has to offer to a better advantage with an understanding of this topic. This section covers Bourne Again Shell command line expansion; TC Shell command line expansion is covered starting on page 341.

The Bourne Again Shell scans each token for the various types of expansion and substitution in the following order. Most of these processes expand a word into a single word. Only brace expansion, word splitting, and pathname expansion can change the number of words in a command (except for the expansion of the variable **"$@"**—page 566).

1. Brace expansion (page 323)
2. Tilde expansion (page 324)
3. Parameter and variable expansion (page 325)
4. Arithmetic expansion (page 325)
5. Command substitution (page 327)
6. Word splitting (page 328)
7. Pathname expansion (page 329)
8. Process substitution (page 330)

Quote removal

After bash finishes with the preceding list, it removes from the command line single quotation marks, double quotation marks, and backslashes that are not a result of an expansion. This process is called *quote removal*.

ORDER OF EXPANSION

The order in which bash carries out these steps affects the interpretation of commands. For example, if you set a variable to a value that looks like the instruction

for output redirection and then enter a command that uses the variable's value to perform redirection, you might expect bash to redirect the output.

```
$ SENDIT="> /tmp/saveit"
$ echo xxx $SENDIT
xxx > /tmp/saveit
$ cat /tmp/saveit
cat: /tmp/saveit: No such file or directory
```

In fact, the shell does *not* redirect the output—it recognizes input and output redirection before it evaluates variables. When it executes the command line, the shell checks for redirection and, finding none, evaluates the **SENDIT** variable. After replacing the variable with **> /tmp/saveit**, bash passes the arguments to echo, which dutifully copies its arguments to standard output. No **/tmp/saveit** file is created.

The following sections provide more detailed descriptions of the steps involved in command processing. Keep in mind that double and single quotation marks cause the shell to behave differently when performing expansions. Double quotation marks permit parameter and variable expansion but suppress other types of expansion. Single quotation marks suppress all types of expansion.

BRACE EXPANSION

Brace expansion, which originated in the C Shell, provides a convenient way to specify filenames when pathname expansion does not apply. Although brace expansion is almost always used to specify filenames, the mechanism can be used to generate arbitrary strings; the shell does not attempt to match the brace notation with the names of existing files.

Brace expansion is turned on in interactive and noninteractive shells by default; you can turn it off with **set +o braceexpand**. The shell also uses braces to isolate variable names (page 281).

The following example illustrates how brace expansion works. The ls command does not display any output because there are no files in the working directory. The echo builtin displays the strings that the shell generates with brace expansion. In this case the strings do not match filenames (there are no files in the working directory.)

```
$ ls
$ echo chap_{one,two,three}.txt
chap_one.txt chap_two.txt chap_three.txt
```

The shell expands the comma-separated strings inside the braces in the echo command into a SPACE-separated list of strings. Each string from the list is prepended with the string **chap_**, called the *preamble,* and appended with the string **.txt**, called the *postscript*. Both the preamble and the postscript are optional. The left-to-right order of the strings within the braces is preserved in the expansion. For the shell to treat the left and right braces specially and for brace expansion to occur, at least one comma and no unquoted whitespace characters must be inside the braces. You can nest brace expansions.

Brace expansion is useful when there is a long preamble or postscript. The following example copies the four files **main.c**, **f1.c**, **f2.c**, and **tmp.c** located in the **/usr/local/src/C** directory to the working directory:

```
$ cp /usr/local/src/C/{main,f1,f2,tmp}.c .
```

You can also use brace expansion to create directories with related names:

```
$ ls -F
file1  file2  file3
$ mkdir vrs{A,B,C,D,E}
$ ls -F
file1  file2  file3  vrsA/  vrsB/  vrsC/  vrsD/  vrsE/
```

The **–F** option causes ls to display a slash (/) after a directory and an asterisk (*) after an executable file.

If you tried to use an ambiguous file reference instead of braces to specify the directories, the result would be different (and not what you wanted):

```
$ rmdir vrs*
$ mkdir vrs[A-E]
$ ls -F
file1  file2  file3  vrs[A-E]/
```

An ambiguous file reference matches the names of existing files. Because it found no filenames matching **vrs[A–E]**, bash passed the ambiguous file reference to mkdir, which created a directory with that name. Page 136 has a discussion of brackets in ambiguous file references.

TILDE EXPANSION

Chapter 4 (page 78) showed a shorthand notation to specify your home directory or the home directory of another user. This section provides a more detailed explanation of *tilde expansion*.

The tilde (~) is a special character when it appears at the start of a token on a command line. When it sees a tilde in this position, bash looks at the following string of characters—up to the first slash (/) or to the end of the word if there is no slash—as a possible login name. If this possible login name is null (that is, if the tilde appears as a word by itself or if it is immediately followed by a slash), the shell substitutes the value of the **HOME** variable for the tilde. The following example demonstrates this expansion, where the last command copies the file named **letter** from Alex's home directory to the working directory:

```
$ echo $HOME
/Users/alex
$ echo ~
/Users/alex
$ echo ~/letter
/Users/alex/letter
$ cp ~/letter .
```

If the string of characters following the tilde forms a valid login name, the shell substitutes the path of the home directory associated with that login name for the tilde and name. If it is not null and not a valid login name, the shell does not make any substitution:

```
$ echo ~jenny
/Users/jenny
$ echo ~root
/var/root
$ echo ~xx
~xx
```

Tildes are also used in directory stack manipulation (page 275). In addition, **~+** is a synonym for **PWD** (the name of the working directory), and **~–** is a synonym for **OLDPWD** (the name of the previous working directory).

PARAMETER AND VARIABLE EXPANSION

On a command line a dollar sign (**$**) that is not followed by an open parenthesis introduces parameter or variable expansion. *Parameters* include command line, or positional, parameters (page 564) and special parameters (page 562). *Variables* include user-created variables (page 279) and keyword variables (page 284). The bash man and info pages do not make this distinction, however.

Parameters and variables are not expanded if they are enclosed within single quotation marks or if the leading dollar sign is escaped (preceded with a backslash). If they are enclosed within double quotation marks, the shell expands parameters and variables.

ARITHMETIC EXPANSION

The shell performs *arithmetic expansion* by evaluating an arithmetic expression and replacing it with the result. See page 355 for information on arithmetic expansion under tcsh. Under bash the syntax for arithmetic expansion is

*$((**expression**))*

The shell evaluates ***expression*** and replaces *$((**expression**))* with the result of the evaluation. This syntax is similar to the syntax used for command substitution [*$(...)*] and performs a parallel function. You can use *$((**expression**))* as an argument to a command or in place of any numeric value on a command line.

The rules for forming ***expression*** are the same as those found in the C programming language; all standard C arithmetic operators are available (see Table 13-8 on page 588). Arithmetic in bash is done using integers. Unless you use variables of type integer (page 284) or actual integers, however, the shell must convert string-valued variables to integers for the purpose of the arithmetic evaluation.

You do not need to precede variable names within *expression* with a dollar sign (**$**). In the following example, an arithmetic expression determines how many years are left until age 60:

```
$ cat age_check
#!/bin/bash
echo -n "How old are you? "
read age
echo "Wow, in $((60-age)) years, you'll be 60!"

$ age_check
How old are you? 55
Wow, in 5 years, you'll be 60!
```

You do not need to enclose the *expression* within quotation marks because bash does not perform filename expansion on it. This feature makes it easier for you to use an asterisk (*****) for multiplication, as the following example shows:

```
$ echo There are $((60*60*24*365)) seconds in a non-leap year.
There are 31536000 seconds in a non-leap year.
```

Fewer dollar signs ($)

tip When you use variables within **$((** and **))**, the dollar signs that precede individual variable references are optional:

```
$ x=23 y=37
$ echo $((2*$x + 3*$y))
157
$ echo $((2*x + 3*y))
157
```

The next example uses wc, cut, arithmetic expansion, and command substitution to estimate the number of pages required to print the contents of the file **letter.txt**. The output of the wc utility (page 888) used with the **–l** option is the number of lines in the file, in columns 1 through 4, followed by a SPACE and the name of the file (the first command following). The cut utility (page 699) with the **–c1-4** option extracts the first four columns.

```
$ wc -l letter.txt
351 letter.txt
$ wc -l letter.txt | cut -c1-4
351
```

The dollar sign and single parenthesis instruct the shell to perform command substitution; the dollar sign and double parentheses indicate arithmetic expansion:

```
$ echo $(( $(wc -l letter.txt | cut -c1-4)/66 + 1))
6
```

The preceding example sends standard output from wc to standard input of cut via a pipe. Because of command substitution, the output of both commands replaces the commands between the **$(** and the matching) on the command line. Arithmetic

expansion then divides this number by 66, the number of lines on a page. A 1 is added at the end because the integer division results in any remainder being discarded.

Another way to get the same result without using cut is to redirect the input to wc instead of having wc get its input from a file you name on the command line. When you redirect its input, wc does not display the name of the file:

```
$ wc -l < letter.txt
    351
```

It is common practice to assign the result of arithmetic expansion to a variable:

```
$ numpages=$(( $(wc -l < letter.txt)/66 + 1))
```

let builtin

The let builtin (not available in tcsh) evaluates arithmetic expressions just as the $(( )) syntax does. The following command is equivalent to the preceding one:

```
$ let "numpages=$(wc -l < letter.txt)/66 + 1"
```

The double quotation marks keep the SPACEs (both those you can see and those that result from the command substitution) from separating the expression into separate arguments to let. The value of the last expression determines the exit status of let. If the value of the last expression is 0, the exit status of let is 1; otherwise, the exit status is 0.

You can give multiple arguments to let on a single command line:

```
$ let a=5+3 b=7+2
$ echo $a $b
8 9
```

When you refer to variables when doing arithmetic expansion with let or $(( )), the shell does not require you to begin the variable name with a dollar sign ($). Nevertheless, it is a good practice to do so, as in most places you must include this symbol.

COMMAND SUBSTITUTION

Command substitution replaces a command with the output of that command. The preferred syntax for command substitution under bash follows:

$(command)

Under bash you can also use the following syntax, which is the only syntax allowed under tcsh:

`` `command` ``

The shell executes ***command*** within a subshell and replaces ***command***, along with the surrounding punctuation, with standard output of ***command***.

In the following example, the shell executes pwd and substitutes the output of the command for the command and surrounding punctuation. Then the shell passes the output of the command, which is now an argument, to echo, which displays it.

```
$ echo $(pwd)
/Users/alex
```

The next script assigns the output of the pwd builtin to the variable **where** and displays a message containing the value of this variable:

```
$ cat where
where=$(pwd)
echo "You are using the $where directory."
$ where
You are using the /Users/jenny directory.
```

Although it illustrates how to assign the output of a command to a variable, this example is not realistic. You can more directly display the output of pwd without using a variable:

```
$ cat where2
echo "You are using the $(pwd) directory."
$ where2
You are using the /Users/jenny directory.
```

The following command uses find to locate files with the name **README** in the directory tree with its root at the working directory. This list of files is standard output of find and becomes the list of arguments to ls.

```
$ ls -l $(find . -name README -print)
```

The next command line shows the older **`command`** syntax:

```
$ ls -l `find . -name README -print`
```

One advantage of the newer syntax is that it avoids the rather arcane rules for token handling, quotation mark handling, and escaped back ticks within the old syntax. Another advantage of the new syntax is that it can be nested, unlike the old syntax. For example, you can produce a long listing of all **README** files whose size exceeds the size of **./README** with the following command:

```
$ ls -l $(find . -name README -size +$(echo $(cat ./README | wc -c)c ) -print )
```

Try giving this command after giving a **set –x** command (page 536) to see how bash expands it. If there is no **README** file, you just get the output of **ls –l**.

For additional scripts that use command substitution, see pages 532, 549, and 579.

$((Versus $(

tip The symbols **$((** constitute a separate token. They introduce an arithmetic expression, not a command substitution. Thus, if you want to use a parenthesized subshell (page 271) within **$()**, you must insert a SPACE between the **$(** and the next **(**.

WORD SPLITTING

The results of parameter and variable expansion, command substitution, and arithmetic expansion are candidates for word splitting. Using each character of **IFS**

(page 288) as a possible delimiter, bash splits these candidates into words or tokens. If **IFS** is unset, bash uses its default value (SPACE-TAB-NEWLINE). If **IFS** is null, bash does not split words.

PATHNAME EXPANSION

Pathname expansion (page 133), also called *filename generation* or *globbing,* is the process of interpreting ambiguous file references and substituting the appropriate list of filenames. Unless **noglob** (page 320) is set, the shell performs this function when it encounters an ambiguous file reference—a token containing any of the unquoted characters *****, **?**, [, or]. If bash cannot locate any files that match the specified pattern, the token with the ambiguous file reference is left alone. The shell does not delete the token or replace it with a null string but rather passes it to the program as is (except see **nullglob** on page 320). The TC Shell generates an error message.

In the first echo command in the following example, the shell expands the ambiguous file reference **tmp*** and passes three tokens (**tmp1**, **tmp2**, and **tmp3**) to echo. The echo builtin displays the three filenames it was passed by the shell. After rm removes the three **tmp*** files, the shell finds no filenames that match **tmp*** when it tries to expand it. Thus it passes the unexpanded string to the echo builtin, which displays the string it was passed.

```
$ ls
tmp1 tmp2 tmp3
$ echo tmp*
tmp1 tmp2 tmp3
$ rm tmp*
$ echo tmp*
tmp*
```

By default the same command causes the TC Shell to display an error message:

```
tcsh $ echo tmp*
echo: No match
```

A period that either starts a pathname or follows a slash (**/**) in a pathname must be matched explicitly unless you have set **dotglob** (page 319). The option **nocaseglob** (page 320) causes ambiguous file references to match filenames without regard to case.

Quotation marks

Putting double quotation marks around an argument causes the shell to suppress pathname and all other expansion except parameter and variable expansion. Putting single quotation marks around an argument suppresses all types of expansion. The second echo command in the following example shows the variable **$alex** between double quotation marks, which allow variable expansion. As a result the shell expands the variable to its value: **sonar**. This expansion does not occur in the third echo command, which uses single quotation marks. Because neither single nor double quotation marks allow pathname expansion, the last two commands display the unexpanded argument **tmp***.

```
$ echo tmp* $alex
tmp1 tmp2 tmp3 sonar
$ echo "tmp* $alex"
tmp* sonar
$ echo 'tmp* $alex'
tmp* $alex
```

The shell distinguishes between the value of a variable and a reference to the variable and does not expand ambiguous file references if they occur in the value of a variable. As a consequence you can assign to a variable a value that includes special characters, such as an asterisk (*).

Levels of expansion

In the next example, the working directory has three files whose names begin with **letter**. When you assign the value **letter*** to the variable **var**, the shell does not expand the ambiguous file reference because it occurs in the value of a variable (in the assignment statement for the variable). No quotation marks surround the string **letter***; context alone prevents the expansion. After the assignment the set builtin (with the help of grep) shows the value of **var** to be **letter***.

The three **echo** commands demonstrate three levels of expansion. When **$var** is quoted with single quotation marks, the shell performs no expansion and passes the character string **$var** to echo, which displays it. When you use double quotation marks, the shell performs variable expansion only and substitutes the value of the **var** variable for its name, preceded by a dollar sign. No pathname expansion is performed on this command because double quotation marks suppress it. In the final command, the shell, without the limitations of quotation marks, performs variable substitution and then pathname expansion before passing the arguments to echo.

```
$ ls letter*
letter1  letter2  letter3
$ var=letter*
$ set | grep var
var='letter*'
$ echo '$var'
$var
$ echo "$var"
letter*
$ echo $var
letter1 letter2 letter3
```

PROCESS SUBSTITUTION

A special feature of the Bourne Again Shell is the ability to replace filename arguments with processes. An argument with the syntax ***<(command)*** causes ***command*** to be executed and the output written to a named pipe (FIFO). The shell replaces that argument with the name of the pipe. If that argument is then used as the name of an input file during processing, the output of ***command*** is read. Similarly an argument with the syntax ***>(command)*** is replaced by the name of a pipe that ***command*** reads as standard input.

The following example uses sort (page 837) with the **–m** (merge, which works correctly only if the input files are already sorted) option to combine two word lists into a single list. Each word list is generated by a pipe that extracts words matching a pattern from a file and sorts the words in that list.

```
$ sort -m -f <(grep "[^A-Z]..$" memo1 | sort) <(grep ".*aba.*" memo2 |sort)
```

CHAPTER SUMMARY

The shell is both a command interpreter and a programming language. As a command interpreter, the shell executes commands you enter in response to its prompt. As a programming language, the shell executes commands from files called shell scripts. When you start a shell, it typically runs one or more startup files.

Running a shell script

Assuming that the file holding a shell script is in the working directory, there are three basic ways to execute the shell script from the command line.

1. Type the simple filename of the file that holds the script.
2. Type a relative pathname, including the simple filename preceded by **./**.
3. Type **bash** or **tcsh** followed by the name of the file.

Technique 1 requires that the working directory be in the **PATH** variable. Techniques 1 and 2 require that you have execute and read permission for the file holding the script. Technique 3 requires that you have read permission for the file holding the script.

Job control

A job is one or more commands connected by pipes. You can bring a job running in the background into the foreground by using the fg builtin. You can put a foreground job into the background by using the bg builtin, provided that you first suspend the job by pressing the suspend key (typically CONTROL-Z). Use the jobs builtin to see which jobs are running or suspended.

Variables

The shell allows you to define variables. You can declare and initialize a variable by assigning a value to it; you can remove a variable declaration by using unset. Variables are local to a process unless they are exported using the export (bash) or setenv (tcsh) builtin to make them available to child processes. Variables you declare are called *user-created* variables. The shell also defines called *keyword* variables. Within a shell script you can work with the command line (*positional*) parameters the script was called with.

Process

Each process has a unique identification (PID) number and is the execution of a single Mac OS X command. When you give it a command, the shell forks a new (child) process to execute the command, unless the command is built into the shell (page 138). While the child process is running, the shell is in a state called sleep. By ending a command line with an ampersand (&), you can run a child process in the background and bypass the sleep state so that the shell prompt returns immediately after you press RETURN. Each command in a shell script forks a separate process, each

of which may in turn fork other processes. When a process terminates, it returns its exit status to its parent process. An exit status of zero signifies success and nonzero signifies failure.

History — The history mechanism, a feature adapted from the C Shell, maintains a list of recently issued command lines, also called *events,* that provides a way to reexecute previous commands quickly. There are several ways to work with the history list; one of the easiest is to use a command line editor.

Command line editors — When using an interactive Bourne Again Shell, you can edit your command line and commands from the history file, using either of the Bourne Again Shell's command line editors (vi[m] or emacs). When you use the vi(m) command line editor, you start in Input mode, unlike the way you normally enter vi(m). You can switch between Command and Input modes. The emacs editor is modeless and distinguishes commands from editor input by recognizing control characters as commands.

Aliases — An alias is a name that the shell translates into another name or (complex) command. Aliases allow you to define new commands by substituting a string for the first token of a simple command. The Bourne Again and TC Shells use different syntaxes to define an alias, but aliases in both shells work similarly.

Functions — A shell function is a series of commands that, unlike a shell script, are parsed prior to being stored in memory so that they run faster than shell scripts. Shell scripts are parsed at runtime and are stored on disk. A function can be defined on the command line or within a shell script. If you want the function definition to remain in effect across login sessions, you can define it in a startup file. Like the functions of a programming language, a shell function is called by giving its name followed by any arguments.

Shell features — There are several ways to customize the shell's behavior. You can use options on the command line when you call bash and you can use the bash set and shopt builtins to turn features on and off.

Command line expansion — When it processes a command line, the Bourne Again Shell may replace some words with expanded text. Most types of command line expansion are invoked by the appearance of a special character within a word (for example, a leading dollar sign denotes a variable). See Table 8-6 on page 292 for a list of special characters. The expansions take place in a specific order. Following the history and alias expansions, the common expansions are parameter and variable expansion, command substitution, and pathname expansion. Surrounding a word with double quotation marks suppresses all types of expansion except parameter and variable expansion. Single quotation marks suppress all types of expansion, as does quoting (escaping) a special character by preceding it with a backslash.

EXERCISES

1. Explain the following unexpected result:

```
$ whereis date
/bin/date
$ echo $PATH
.:/usr/local/bin:/usr/bin:/bin
$ cat > date
echo "This is my own version of date."
$ date
Tue May 24 11:45:49 PDT 2005
```

2. What are two ways you can execute a shell script when you do not have execute access permission for the file containing the script? Can you execute a shell script if you do not have read access permission for the file containing the script?
3. What is the purpose of the **PATH** variable?
   a. Set the **PATH** variable so that it causes the shell to search the following directories in order:
      - **/usr/local/bin**
      - **/usr/bin/X11**
      - **/usr/bin**
      - **/bin**
      - **/Developer/Tools**
      - The **bin** directory in your home directory
      - The working directory

   b. If there is a file named **doit** in **/usr/bin** and another file with the same name in your **~/bin**, which one will be executed? (Assume that you have execute permission for both files.)

   c. If your **PATH** variable is not set to search the working directory, how can you execute a program located there?

   d. Which command can you use to add the directory **/usr/sbin** to the end of the list of directories in **PATH**?
4. Assume that you have made the following assignment:

   ```
   $ person=jenny
   ```

   Give the output of each of the following commands:

   a. **echo $person**

   b. **echo '$person'**

   c. **echo "$person"**

5. The following shell script adds entries to a file named **journal-file** in your home directory. This script helps you keep track of phone conversations and meetings.

```
$ cat journal
# journal: add journal entries to the file
# $HOME/journal-file

file=$HOME/journal-file
date >> $file
echo -n "Enter name of person or group: "
read name
echo "$name" >> $file
echo >> $file
cat >> $file
echo "----------------------------------------------------" >> $file
echo >> $file
```

   a. What do you have to do to the script to be able to execute it?

   b. Why does the script use the read builtin (page 571) the first time it accepts input from the terminal and the cat utility the second time?

6. Assume that the **/Users/jenny/grants/biblios** and **/Users/jenny/biblios** directories exist. Give Jenny's working directory after she executes each sequence of commands given. Explain what happens in each case.

   a.

```
$ pwd
/Users/jenny/grants
$ CDPATH=$(pwd)
$ cd
$ cd biblios
```

   b.

```
$ pwd
/Users/jenny/grants
$ CDPATH=$(pwd)
$ cd $HOME/biblios
```

7. Name two ways you can identify the PID number of your login shell.

8. Give the following command:

```
$ sleep 30 | cat /etc/weekly
```

   Is there any output from sleep? Where does cat get its input from? What has to happen before the shell displays another prompt?

ADVANCED EXERCISES

9. Write a sequence of commands or a script that demonstrates that variable expansion occurs before pathname expansion.
10. Write a shell script that outputs the name of the shell that is executing it.
11. Explain the behavior of the following shell script:

```
$ cat quote_demo
twoliner="This is line 1.
This is line 2."
echo "$twoliner"
echo $twoliner
```

    a. How many arguments does each echo command see in this script? Explain.

    b. Redefine the **IFS** shell variable so that the output of the second echo is the same as the first.

12. Add the exit status of the previous command to your prompt so that it behaves similarly to the following:

```
$ [0] ls xxx
ls: xxx: No such file or directory
$ [1]
```

13. The dirname utility treats its argument as a pathname and writes to standard output the path prefix—that is, everything up to but not including the last component:

```
$ dirname a/b/c/d
a/b/c
```

    If you give **dirname** a simple filename (no **/** characters) as an argument, **dirname** writes a **.** to standard output:

```
$ dirname simple
.
```

    Implement dirname as a bash function. Make sure that it behaves sensibly when given such arguments as **/**.

14. Implement the basename utility, which writes the last component of its pathname argument to standard output, as a bash function. For example, given the pathname **a/b/c/d**, basename writes **d** to standard output:

```
$ basename a/b/c/d
d
```

15. The Mac OS X basename utility has an optional second argument. If you give the command **basename** ***path suffix***, basename removes the ***suffix*** and the prefix from ***path***:

```
$ basename src/shellfiles/prog.bash .bash
prog
$ basename src/shellfiles/prog.bash .c
prog.bash
```

Add this feature to the function you wrote for exercise 14.